Delft Students on Software Architecture
DESOSA 2017

van Deursen, Arie; Zaidman, Andy; Aniche, Maurício; Mairet, Valentine; Oever, Sander van den

Publication date
2017

Document Version
Final published version

Citation (APA)
van Deursen, A., Zaidman, A., Aniche, M., Mairet, V., & Oever, S. V. D. (2017). *Delft Students on Software Architecture: DESOSA 2017*. (Delft Students on Software Architecture: DESOSA; Vol. 3). Delft University of Technology. https://www.gitbook.com/book/delftswa/desosa-2017/details

Table of Contents

Delft Students on Software Architecture: DESOSA 2017

Arie van Deursen, **Andy Zaidman**, **Maurício Aniche**, **Valentine Mairet** and **Sander van den Oever**.
Delft University of Technology, The Netherlands, April 17, 2017, Version 1.0

We are proud to present *Delft Students on Software Architecture*, a collection of 21 architectural descriptions of open source software systems written by students from Delft University of Technology during a master-level course taking place in the spring of 2017.

In this course, teams of approximately 4 students could adopt a project of choice on GitHub. The projects selected had to be sufficiently complex and actively maintained (one or more pull requests merged per day). The systems are from a wide range of domains, including testing frameworks (Mockito, JUnit5), editors (Neovim, VSCode) and visualisation (Kibana).

During an 8-week period, the students spent one third of their time on this course, and engaged with these systems in order to understand and describe their software architecture.

Inspired by Brown and Wilsons' Architecture of Open Source Applications, we decided to organize each description as a chapter, resulting in the present online book.

This book is the third in volume the DESOSA series: The first DESOSA book resulted from the 2015 edition of the course, and contained architectural descriptions of ten (different) open source systems. The second DESOSA book followed a year later, including 21 architectural descriptions.

Recurring Themes

The chapters share several common themes, which are based on smaller assignments the students conducted as part of the course. These themes cover different architectural 'theories' as available on the web or in textbooks. The course used Rozanski and Woods' Software Systems Architecture, and therefore several of their architectural viewpoints and perspectives recur.

The first theme is outward looking, focusing on the use of the system. Thus, many of the chapters contain an explicit stakeholder analysis, as well as a description of the context in which the systems operate. These were based on available online documentation, as well as on an analysis of open and recently closed issues for these systems.

A second theme involves the development viewpoint, covering modules, layers, components, and their inter-dependencies. Furthermore, it addresses integration and testing processes used for the system under analysis.

A third recurring theme is *technical debt*. Large and long existing projects are commonly vulnerable to debt. The students assessed the current debt in the systems and provided proposals on resolving this debt where possible.

First-Hand Experience

Last but not least, the students tried to make themselves useful by contributing to the actual projects. Many pull requests have been opened, including documentation improvements (Scrapy #2636), bug fixes (Jupyter #2220), style / tooling fixes (yarn #2725) or even feature implementations (JabRef #2610, JUnit5 #723). With these contributions the students had the ability to interact with the community; they often discussed with other developers and architects of the systems. This provided them insights in the architectural trade-offs made in these systems.

The students have written a collaborative chapter on some of the contributions made during the course. It can be found in the dedicated contributions chapter.

Feedback

While we worked hard on the chapters to the best of our abilities, there might always be omissions and inaccuracies. We value your feedback on any of the material in the book. For your feedback, you can:

- Open an issue on our GitHub repository for this book.
- Offer an improvement to a chapter by posting a pull request on our GitHub repository.
- Contact @delftswa on Twitter.
- Send an email to Arie.vanDeursen at tudelft.nl.

Acknowledgments

We would like to thank:

- Our guest speakers: Nicolas Dintzner, Maikel Lobbezoo, Ali Niknam, Alex Nederlof, Felienne Hermans, Marcel Bakker and Marc Philipp.
- Valentine Mairet who created the front cover of this book.
- Michael de Jong and Alex Nederlof who were instrumental in the earlier editions of this

course.
- All open source developers who helpfully responded to the students' questions and contributions.
- The excellent gitbook toolset and gitbook hosting service making it easy to publish a collaborative book like this.

Further Reading

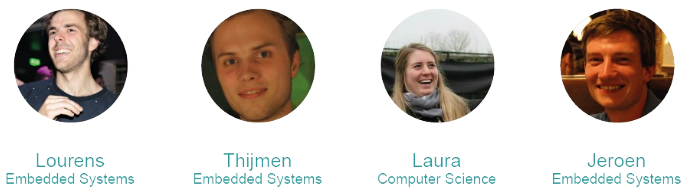
1. Arie van Deursen, Maurício Aniche, Joop Aué, Rogier Slag, Michael de Jong, Alex Nederlof, Eric Bouwers. A Collaborative Approach to Teach Software Architecture. 48th ACM Technical Symposium on Computer Science Education (SIGCSE), 2017.
2. Arie van Deursen, Alex Nederlof, and Eric Bouwers. Teaching Software Architecture: with GitHub! avandeursen.com, December 2013.
3. Arie van Deursen, Maurício Aniche, Joop Aué (editors). Delft Students on Software Architecture: DESOSA 2016, 2016.
4. Arie van Deursen and Rogier Slag (editors). Delft Students on Software Architecture: DESOSA 2015. delftswa.github.io, 2015.
5. Amy Brown and Greg Wilson (editors). The Architecture of Open Source Applications. Volumes 1-2, 2012.
6. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.

Copyright and License

Arduino - The Open-Source IDE

By Jeroen Overman, Lourens Pool, Laura Kreuk and Thijmen Ketel

Delft University of Technology



Abstract

Arduino is a fast and easy prototyping platform based around a small microcontroller and designed for students, artists, hobbyists and developers. The system was created by four engineers to make programming and prototyping easier and cheaper. While the development is open-source, the amount of documentation about contributing is limited. This results in a project where programming is not done through conventions or guidelines but with a strong focus on getting things to work. In this chapter Arduino is analysed through the people that are involved, the development process and the architectural design of the project. Furthermore, issues that slow down or stop the development process are discussed.

Introduction

When we look at the current state of technology it is clear that it is an amazing time to be alive. The speed of technological advancements has never been so high and electronics and gadgets can be found everywhere. What starts out as an idea can be made real in no time through the help of crowdfunding and fast prototyping. A big role in that last part is played by the subject of this chapter: Arduino.

Arduino is an Italian company specialized in making prototyping and tinkering with electronics accessible for the masses. Started by four engineers with the goal to make programming and prototyping easy and, most important, affordable. The system is originally based around Atmels 8-bit microcontroller platform but quickly evolved to be a staple in the projects of students, hobbyist, artists and even professionals. Arduino now has over 700 000 official boards in the hands of its users and almost the same amount of clones (copies of official boards) [1].

While the hardware is created by Arduino itself, the IDE (Integrated Development Environment) written in Java is an open-source effort. This easy to use app is created, added to and improved by a community of enthusiastic developers and users. This makes the Arduino project accessible and fuels innovation through a large pool of ideas from which the IDE can be improved. Prototypes, which are programmed in a C/C++ dialect, can realise innovating ideas like Moistly by easy integration through Arduino.

In this chapter the open-source project of the Arduino IDE is reviewed by four TU Delft students from the DESOSA (Delft Students on Software Architecture). The analysis is done through different aspects of the Arduino system, mostly based on the code, documentation and activity on the open-source Arduino repository on GitHub. Stakeholders, Architecture overview, the existence of Technical Debt, a development-view, and of course an explanation of the Arduino system and its evolution are some of the subjects analysed in this chapter.

Stakeholders

Many stakeholders are involved with Arduino, these are explained according to the most important stakeholders identified by Rozanski and Woods [2]. Arduino is a company that specializes in hardware boards which are plug and play programmable with their own IDE (Integrated Development Environment). This is reflected by structure of the company, which is divided into five different departments:

- Hardware
- Software
- Interaction Design
- Community Managers
- General Management

Table 1: Summary most important stakeholders.

Stakeholders	Description
Acquirers	The founders of Arduino are the ones that authorize funding for the products and system developement. The funding is generated by crowdfunding [11].
Assessors	Rules and regulations that apply to the Arduino product portfolio are handled by the General Management team.
Communicators	The Community Management team explains the system via documentation and training materials.
Users	The Arduino Suite is used by a lot of students, artists, hobbyist as well as by professionals who want a prototype working in no time.
Testers	All bugs in the software can be posted on the Arduino forum, emailed or issued on Github. Bugs and problems are picked up by the software and hardware staff employees of Arduino or the contributors on Github.
Developers	The people working on the code for the end product are both the hard- and software team.
Support staff	The website forum.arduino.cc plays a central role in staying in tough with users that report bugs as well as happily explaining nice projects that can be built with the products.
Suppliers	Intel and Atmel are the main suppliers for the hardware. Furthermore, multiple tools and packages are used for both the software and the hardware.
Maintainers	Software solutions are assessed by the Software development team in order to maintain the programming standards and architectural choices.

The four founders of Arduino are actively pushing and developing the long term strategy of the company [4].

Table 2: Founders of Arduino and their role within the company.

Founders	Role
Massimo Banzi	Is the CEO and has the most broad role. He gets advice from the three other founders.
David Cuartielles	Co-CEO and researcher on education projects, also maintains relationships with education policy makers in the European Union [11]. Educational use is a large share of the sales.
Tom Igoe	Responsible for user experience design in both software and hardware.
David Cuartielles	Focusses on the development of the Arduino software, mainly the IDE.

In most of the departments there is one person responsible for the communication to the stakeholders. One of the founders, Tom Igoe, advises on issues about the documentation, whereas Mikeal does this in the design team. Finally, the most important communicator is the Community Management team. The Community Management teaches Arduino and writes documentation and tutorials. They also speak at conferences and exhibitions.

Arduino has a team of engineers dedicated to both hard- and software. This chapter focuses on the software side of the development. The team is responsible for the end product and it works closely with the open source community on GitHub to constantly improve the product.

Anyone can join in designing the software and is encouraged to use the mailing list to get in touch with the software development team. In table 3 a small ranking of the most active contributors is shown in terms of number of commits. It does not reflect the most recently active contributors. It can be seen that most non-employees stay active for three years or so [5].

Table 3: Most active contributors in trems of number of commits.

Contributor	Roles	Commits	LOC++	LOC--	Active during
Christian Maglie	Software Engineer Arduino	1.220	1.449.973	1.334.641	2011 - present
David A. Mellis	Founder	1.034	1.100.356	1.557.719	2005 - 2015
tigoe	Contributor	330	1.024.956	25.437	2009 - 2013
Martino Facchin	Firmware Engineer Arduino	226	10.570	3.829.094	2015 - present
Matthijs Kooijman	Contributor	225	6.575	5.791	2013 - 2016
zeveland	Contributor	187	37.586	32.097	2011 - 2013
Thibaut Viard	Contributor	115	4.413.074	1.328.159	2011 - 2013

On the software side Arduino has the following suppliers. It uses the following packages: GNU avr-gcc toolchain, GCC ARM Embedded toolchain, avr-libc, avrdude, bossac, openOCD and code from Processing and Wiring. Arduino targets embedded applications, so the development of the Arduino IDE is also aided by the hardware design tools.

The management division takes responsibility for the strategy of the company. It works closely with all cores of the company: Hardware, Software, Interaction designers and Community Managers to create a clear and complete overview of the brand Arduino.

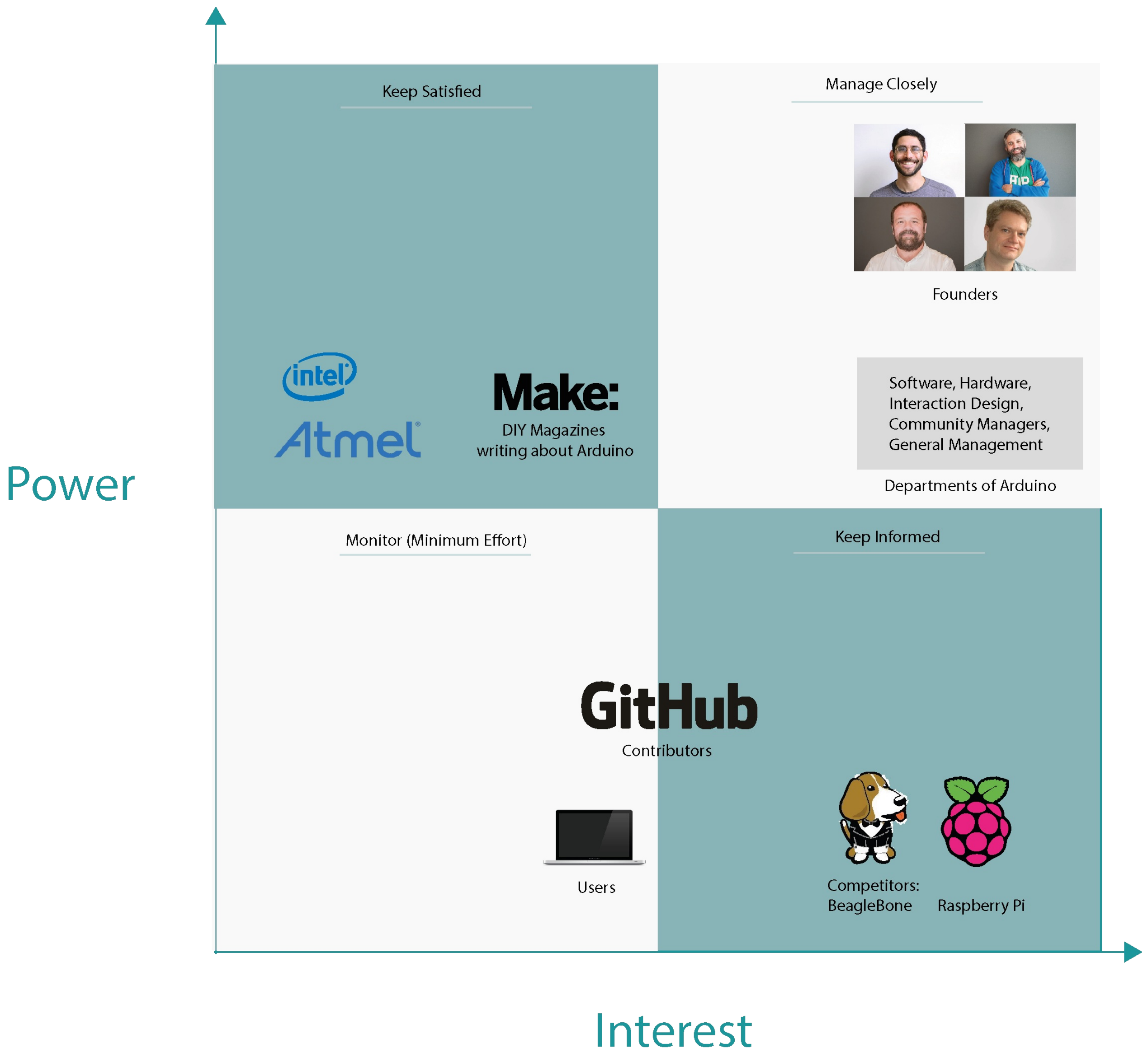


Figure 1: Power Interest Graph of the Arduino IDE.

Context view

The context view shows the different relations, dependencies and interactions Arduino has with its environment. Important for the context view are the people, systems and external entities with which the system interacts [2]. The context view can be seen in figure 2 and the relations will be explained shortly.

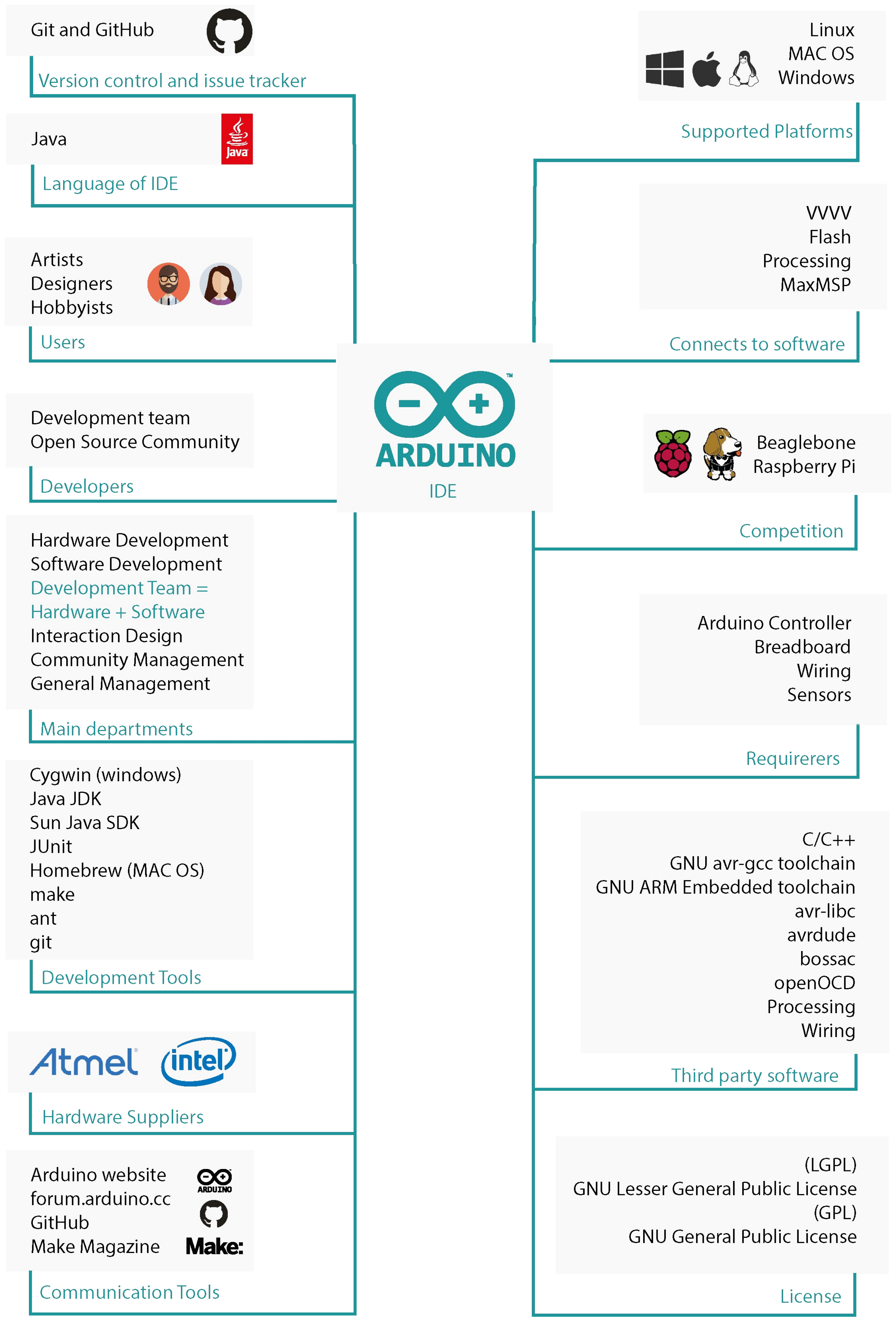


Figure 2: Arduino context model.

In order to understand the context model of Arduino, the role of the external entities with respect to Arduino is explained.

IDE

The core of Arduino is written in Java. The users that work with the Arduino software can write programs, called sketches, to upload to their Arduino board in a form of C or C++. The software of Arduino can be installed on multiple platforms: Linux, Mac OS and Windows. The users of Arduino are mostly artists, designers or hobbyists.

Communication Tools

Arduino uses Github and Git for version control to help developers collaborate and track issues related to the software. Contributions in the repository come from the Arduino software development team, as well as from the open-source community. In order to stay in contact with developers and users the following communication tools are used: the Arduino website, the Arduino forum and Github.

Arduino's community managers also need to maintain a good relationship with magazines like Make who write about Arduinos products and competitors.

Development Tools

When you are developing software for Arduino you can make your life easier by using at least the following tools: Cygwin, Java JDK and Homebrew [7].

Third party software The IDE can be connected to third party software: Flash, VVVV, Processing or Max/MSP. Finally, Arduino has some dependencies during run-time [8]. These can be found in the top right corner of figure 2.

Hardware Boards

When using the Arduino IDE you also need some hardware to start a project, for this an Arduino controller, breadboard, wiring and sensors are needed. Arduino has two main suppliers of the hardware: Intel and Atmel.

Some of the competitors of Arduino are Beaglebone and the Raspberry Pi. The Pi and Beaglebone are equipped with a faster processor and can be connected to the internet and HDMI displays out of the box. While the Arduino boards are using a simple processor and need another shield (hardware board) in order to be connected to the internet or a monitor.

Although Arduino lacks computational power compared to the Pi and the Beaglebone, it compensates this with its focus on simplicity and ease of use. The Arduino boards are also more suitable for hardware related projects. The smooth learning curve, extensive tutorials and lively community allows anyone to start programming.

Development view

This section gives an overview of the structure of the Arduino project from the perspective of the developers. Concerns like module organisation, standardization of design and codeline organisation are addressed. Due to the lack of technical documentation for the structure of the project, some parts cannot be described into depth.

Module organization

Modules are used to create an overview in the thousands of lines of code that the Arduino IDE has. These modules allow for a clear overview of a complex piece of software.

The Arduino repository does not have a specific naming convention that gives a clear overview of the underlying structure. Certain folder names (e.g. edazdarevic) make it unclear what it contains. In addition, not every folder contains an explanation file describing the role or function of the module.

The core modules are the GUI, IDE and hardware. In addition to these main components of the system there is also a module for building the software and a module that stores the libraries.

Table 4: Core modules and their role.

Module	Role
`app`	This folder contains the Graphical User-Interface (GUI)
`arduino-core`	This folder contains the main parts for the IDE
`build`	This directory is meant for building the Arduino IDE to test added code
`hardware`	This folder contains everything for the hardware that is supported by Arduino
`libraries`	The standard libraries that are used by Arduino are stored here

Module structure model

The module structure model gives an overview of the systems source files. It uses modules into which source files are collected and lists the dependencies between the modules.

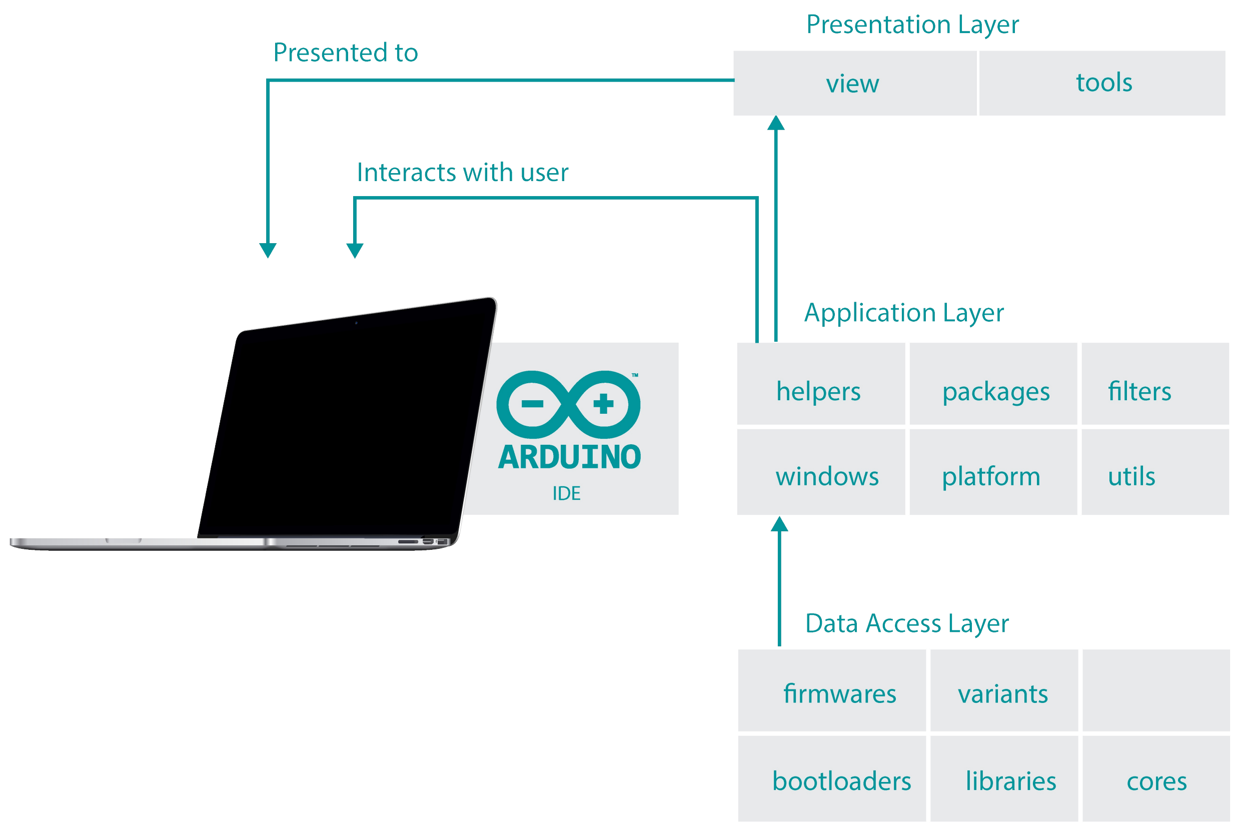


Figure 3: Module Structure Model.

As can be seen in the module structure model (figure 3) the Arduino system consists of three layers: the *presentation layer*, the *application layer* and the *data access layer*. The *presentation layer* consists of the classes that are used to output information to the user via the GUI. The user interacts with the *application layer*. Finally, the *data access layer* is responsible for accessing the data that is used in the system.

Standardization of design

Because Arduino is developed by a *team* of software developers, it is necessary to standardize some key aspects of the design. Furthermore, it is important for the development team to provide a well-designed, maintainable, and stable platform for the future. To achieve these goals, the following design standards are used:

- Emphasizing API design.
- Offering a small but easily-extendable core.
- Sacrificing (when necessary) *elegance of implementation*, in favour of *ease-of-use*.
- Emphasizing real use cases over theoretical possibilities.
- Focusing on the official Arduino hardware (Make sure official Arduino hardware works neatless instead of making a feature of a clone board work).
- Recognizing that documentation is as important as code.
- Trying to get things right the first time (even if this takes longer).

Main design decisions concerning Arduino:

- Uses Apache Ant as a building tool.
- Code is written in Java.
- Uses Github as main code repository and for version control.
- Uses a lot of code from Processing. This is an IDE for so called sketchbooks. This IDE enables users to learn programming in a visual context. .
- Based on same set of abstractions as Wiring, to make programming easier. Uses Wiring as an example for attaching microcontroller boards to software.
- Uses JUnit as a testing framework.

As mentioned before Arduino chooses ease-of-use over elegance of implementation. When inspecting the code of the Arduino IDE very few design patterns can be found.

Standardization of testing

The standardization of testing helps to speed up the testing process and ensures a consistent result for each newly released product. Arduino uses the following standardization of testing:

When a developer wants to contribute to Arduino, a pull-request should be made on GitHub. The Arduino bot makes an automated build of the IDE based on these pull-requests. The Arduino bot replies in the pull-request if the build is successful and links to files that should be tested. After this, the developers of Arduino will comment on the pull-request and discuss if there should be more changes.

The tests in the Arduino software are created using the Arduino test-suite library, they are developed as standard sketches. There are two kind of tests that are constructed for the software. Firstly, tests that are targeting specific issues, these should be made when a contributor fixes a specific Github issue in order to test the solution's correctness. Once the issue is integrated in the project, the test targeting the issue will be added to the automated test runs.

The other kind of tests that are constructed are the test-suite coverage tests. These tests are designed to test the functions in the code and its libraries, they run automatically and are version controlled. Furthermore there is a list of tests, made by identifying the features and issues of a new release. This list of tests should be passed before a new release can be made.

Despite the tests that are written, bugs exists in the Arduino project. A lot of bugs are found by users of the IDE and either communicated through the Arduino Forum or by means of a GitHub issue.

Arduino does not seem to advocate the use of tools that test the code. For example, to look for duplicate code, naming conventions, unused code, etc. Good analysis of the code can benefit in finding problems and bugs early on and should be advocated as much as possible.

Instrumentation

Inserting special code in order to log information, record system states or resource use about step executing is called instrumentation. This instrumentation is used to debug problems in the IDE [2]. Arduino uses the *Java logger* class to log this information.

In the GUI module, a class for a console logger and a class for a log formatter is found. Thus, the logging is done in two ways, directly to the console and saved in a file which can be checked when necessary.

Codeline organization

The codeline organization is the way that the source code is stored in a directory structure, managed via configuration management and how it is built and tested regularly [2]. In an open source project, it is important to organize the source code in a way that everyone can understand it and can add to it. An organized repository attracts enthusiastic programmers to start working on the code.

The source code of the Arduino IDE is spread over different directories also different Eclipse projects are present in the repository. This makes it hard for new developers to get familiar with the project. It is however easy to make your own build of the Arduino project. The Apache ANT automated software build tool is used to build the project. This works on all platforms where Arduino is supported.

Apache Ant is also configured to run the JUnit tests, which makes that easy to do because there is no needs to resolve all the external dependencies. The downside of using Apache Ant to build and test the project is that it is hard to debug the project using your own IDE. It is only possible to run the program en read the debug information. Step-by-step debugging is not easily achieved.

In figure 4 below, a simple overview of the repository is shown. The directories at the repository are in grey and the contents of the root folders are listed in green to the left:

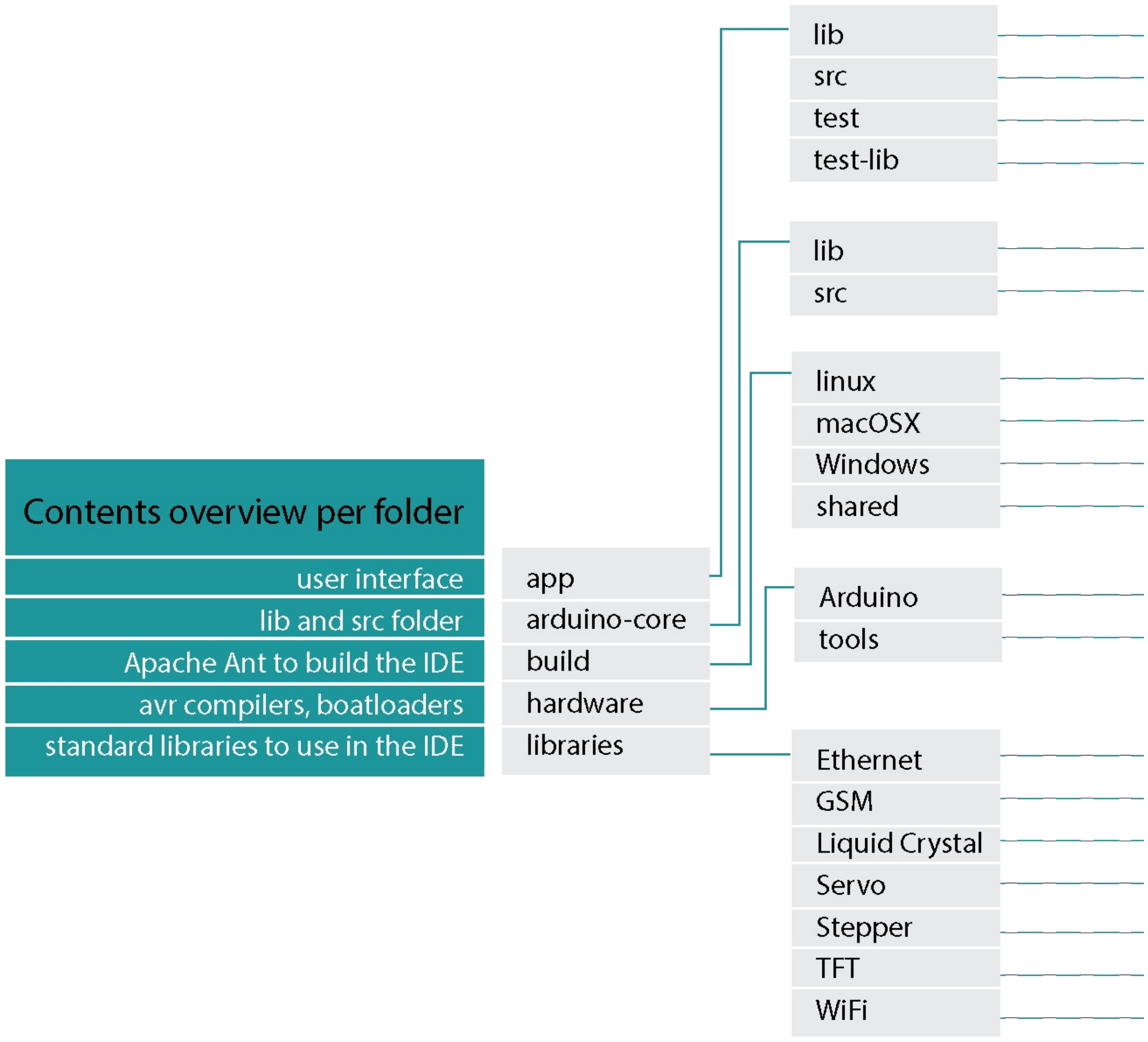


Figure 4: One level unfolding of Arduino repository

Technical debt

Suppose there is a piece of functionality that needs to be added to the system. There are two ways of doing this, the quick "hacky" way and the slow and tedious but more clear and secure way. A lot of the time when a feature, fix or improvement is added to a project, this is done in the first way. The problem with this way of coding is that code might be unclear or too complex and can leave a project hard to contribute to. So in the future when a new feature or improvement needs to be added or a bug needs fixing, the code might require a lot of adjustments to become functional. This problem, regarding nasty code, is called technical debt. This technical debt needs to be managed and kept at a reasonable level to make sure that the developers can keep on contributing to the project. Furthermore, avoiding technical debt can keep the cost for maintenance lower, increase productivity and prevent unwanted surprises. In this section the technical debt that exists in the Arduino project is analysed.

SonarQube analysis

One of the ways to find technical debt in a system is to use specialised software. The software that is used to analyse the debt in Arduino is called SonarQube. This tool is designed to analyse code for bugs, vulnerabilities and report on technical debt based on code smells (ugly code).

Overview & Assessment

When the tool is executed on the Arduino project, it produces an overview of the issues it found. This overview also contains a metric that judges the project on its possibility for successfull release: quality-gate. The overview can be seen in figure 5.

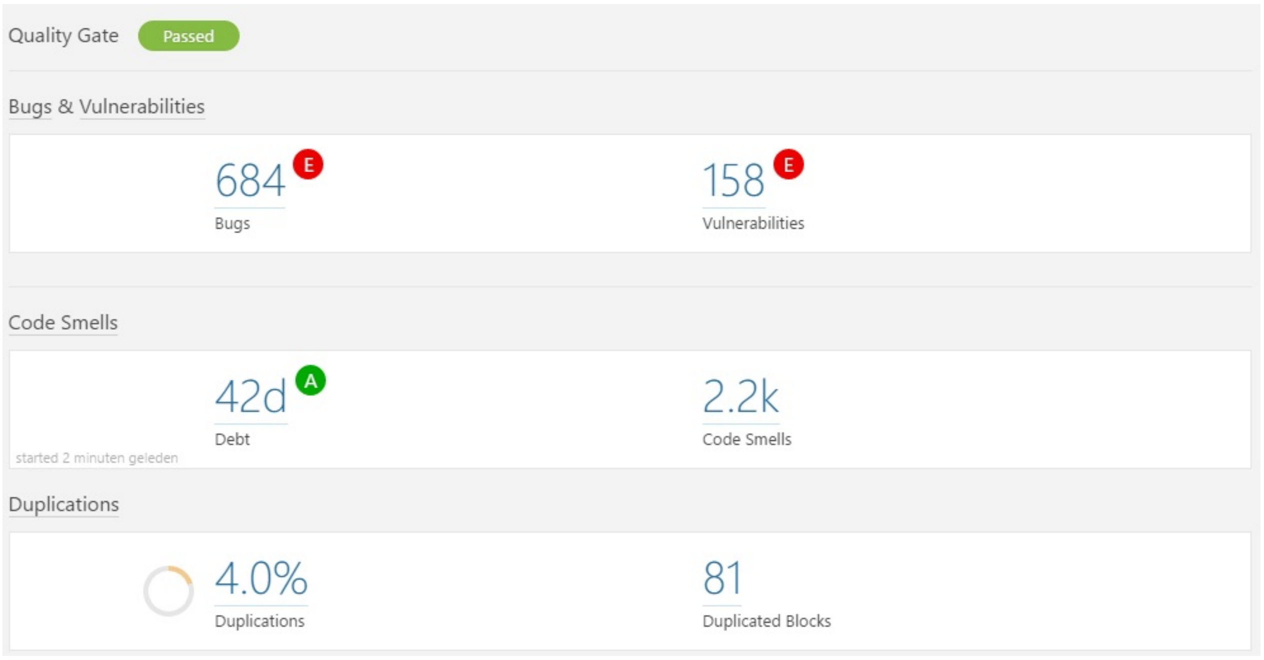


Figure 5: Overview produced by SonarQube of the Arduino project.

SonarQube reported a large amount of bugs, vulnerabilities and code smells during its analysis. The issues ranged from severe (*Blocker*) to harmless (*Info*) and affected almost every part of the system.

Table 5: Issues reported by SonarQube and their assessment.

Issue	Assessment
Bugs	The 684 bugs that were found by SonarQube were mostly harmless, they might block the productivity of the developer but not the behavior of the system. About 50 were actually more severe and could cause problems like deadlocks or resource leaks.
Vulnerabilities	Similar to the bugs, most of the 158 vulnerabilities flagged by SonarQube were harmless and could only impair the productivity of the developer. Four were wrongly labelled as hardcoded password which were not accurate.
Code smells	The number of code smells that were found was pretty high (2228). This meant that the code was confusing, complicated and generally ugly. No coding guidelines or conventions were followed and large parts of the code is not documented.

The total calculated time to resolve all the issues found by SonarQube would be 65 days (this includes 42 days for the technical debt and another 23 days for the bugs and vulnerabilities). It is, however, not necessary to resolve all bugs, vulnerabilities and code smells to reduce the technical debt to a lower level.

The analysis already gives a very big hint about the state of the project and the way Arduino is developed. No guidelines or conventions are followed and large parts of the code is not documented.

Manual analysis

The next step is to look at the technical debt of the project manually. Multiple sources can indicate technical debt.

Table 6: Summary of issues found by manual identification of technical debt.

Indicator	Assessment
Documentation	Documentation that is provided with the Arduino software on the repository is very limited. Most documentation on the repositories wiki are about the end product (programming the hardware). No contribution-guide or coding guidelines are available.
Defects	When looking at the issues that are labeled `bug` on the repository and the age of the issue, it indicates that technical debt exists. Some of the bugs are dated back to 2012 which is very old and gives an indication of technical debt.
Refactoring	Multiple commits can be found that refactor the code and even some issues can be found that request refactoring of the code. Most of the issues are fixed or closed, but it is hard draw a conclusion from this fact as the number of commits containing a refactor are 49 of the more than 6000.

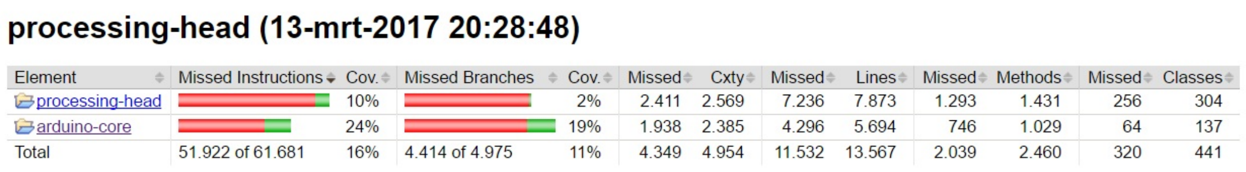
When looking at the indicators it does hint at technical debt. Especially documentation is an issue with Arduino. No clear guidelines and rules exist for contributing.

Testing debt

Testing debt is concered with the testing of altered and existing code. To ensure that functions will keep on working with new changes, the code needs to be tested well enough.

Test coverage

The test coverage of the Arduino project was measured by using the EclEmma plugin for Eclipse. The results of this coverage report can be seen in figure 6. The processing-head is the project that contains the graphical user interface which is based upon the Processing-project. Arduino-core is the core of the Arduino IDE.



processing-head (13-mrt-2017 20:28:48)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
processing-head		10%		2%	2.411	2.569	7.236	7.873	1.293	1.431	256	304
arduino-core		24%		19%	1.938	2.385	4.296	5.694	746	1.029	64	137
Total	51.922 of 61.681	16%	4.414 of 4.975	11%	4.349	4.954	11.532	13.567	2.039	2.460	320	441

Figure 6: Test coverage report of the Arduino project.

As can be seen the test coverage of both Eclipse projects are not that high. The total covered instructions are 10% for the processing-head and 24% for the arduino-core. The total coverage of the branches is 2% for the processing-head and 19% for the arduino-core. These results are very low as most project consider 70%-80% more reasonable. Most of the errors can be attributed to the documentation debt. Trying to build the Arduino project requires a lot of googling due to the lack of a clear guide.

A total of 57 test are run on the Arduino project, which is not a high number. This indicates that the Arduino project probably has testing debt. However, this also relates to the problem of documentation debt.

TODO's and FIXME's

To get an indication of the policy about technical debt at Arduino, multiple sources are investigated. If developers discuss technical debt on the Github repository (in issues or pull-requests) this is more general, other stakeholders than the developers can contribute in the discussion. Whereas if there are TODO's (marker for later work) and FIXME's (marker for broken code) in the source code this is a discussion between the developers of the software.

The amount of "TODO" markers that pop up in the code for Arduino is 94, while the amount of "FIXME" markers is a lot lower with 18\. A third marker (XXX) indicates that some part of the code needs attention but does work, which there are 22 of. The number of FIXME's compared to the amount of TODO's does give a positive indication of how broken code is handled at Arduino. Most of the FIXME's are, however, about adding documentation, which further indicates documentation debt.

Debt evolution

To get a better perspective about the evolution of the technical debt in the project over time, we looked at the different releases of Arduino. The project has a total of 73 releases since 2005 (version 0002), with a release every few months. The releases do not contain any changelogs until the release of version 1.0.5 in 2013\. Arduino does release versions for beta testing by the community to fix bugs early on. Checking the issue list of Arduino there are 4252 issues closed and 719 still open. 43 open issues are tagged with the label bug and the largest part of those are open more than a year already. These issues definitely contribute to technical debt in the future.

To visualise the debt evolution over time SonarQube is executed over six Arduino verions: 1.0 (first release), 1.5 (first large BETA), 1.6 (large overhaul), 1.8.0, 1.8.1 (latest release) and 1.8.2 (current development) [3]. Two parts are visualised: Bugs & Vulnerabilities (figure 7) and Technical debt in days (figure 8). Furthermore, the amount of code and duplicated code is visualised to give an indication of the size of the project (figure 9).

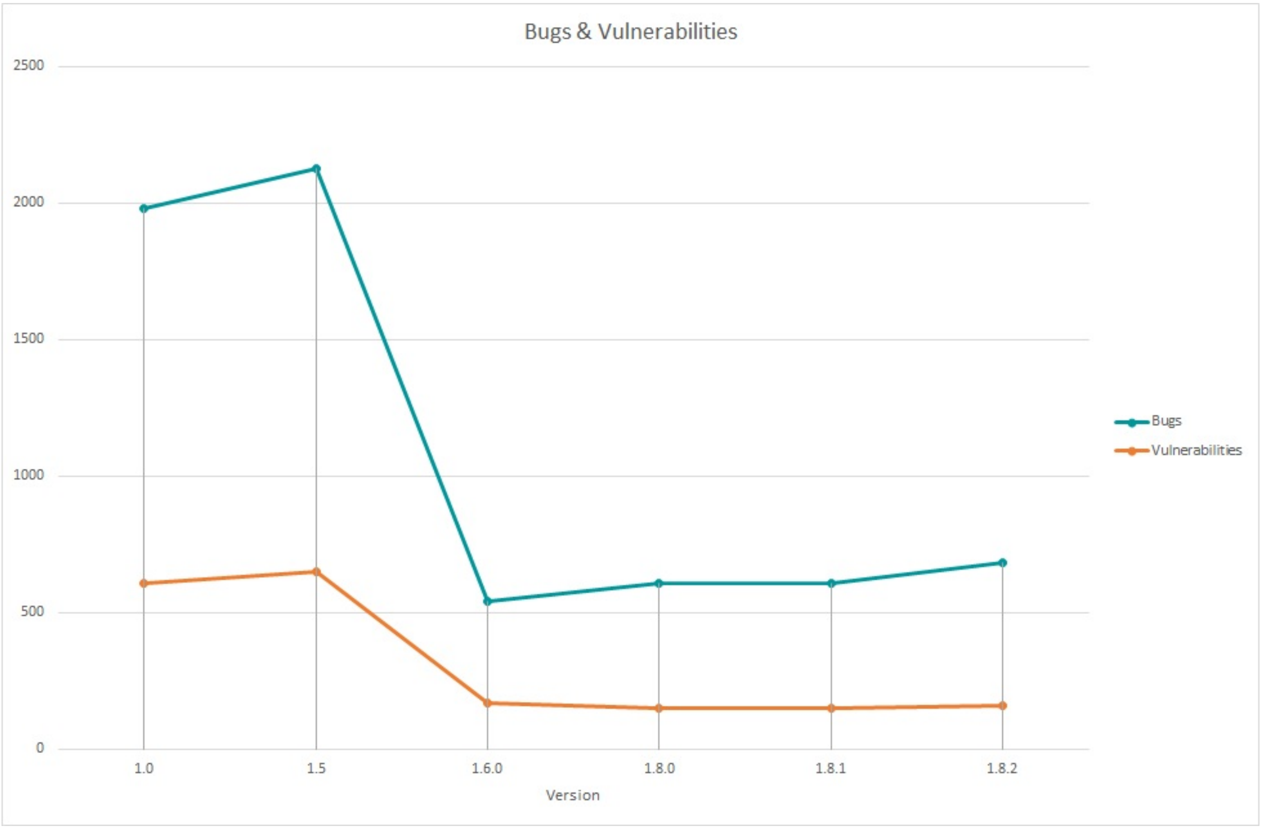


Figure 7: Amount of Bugs and Vulnerabilities over the different releases found by SonarQube.

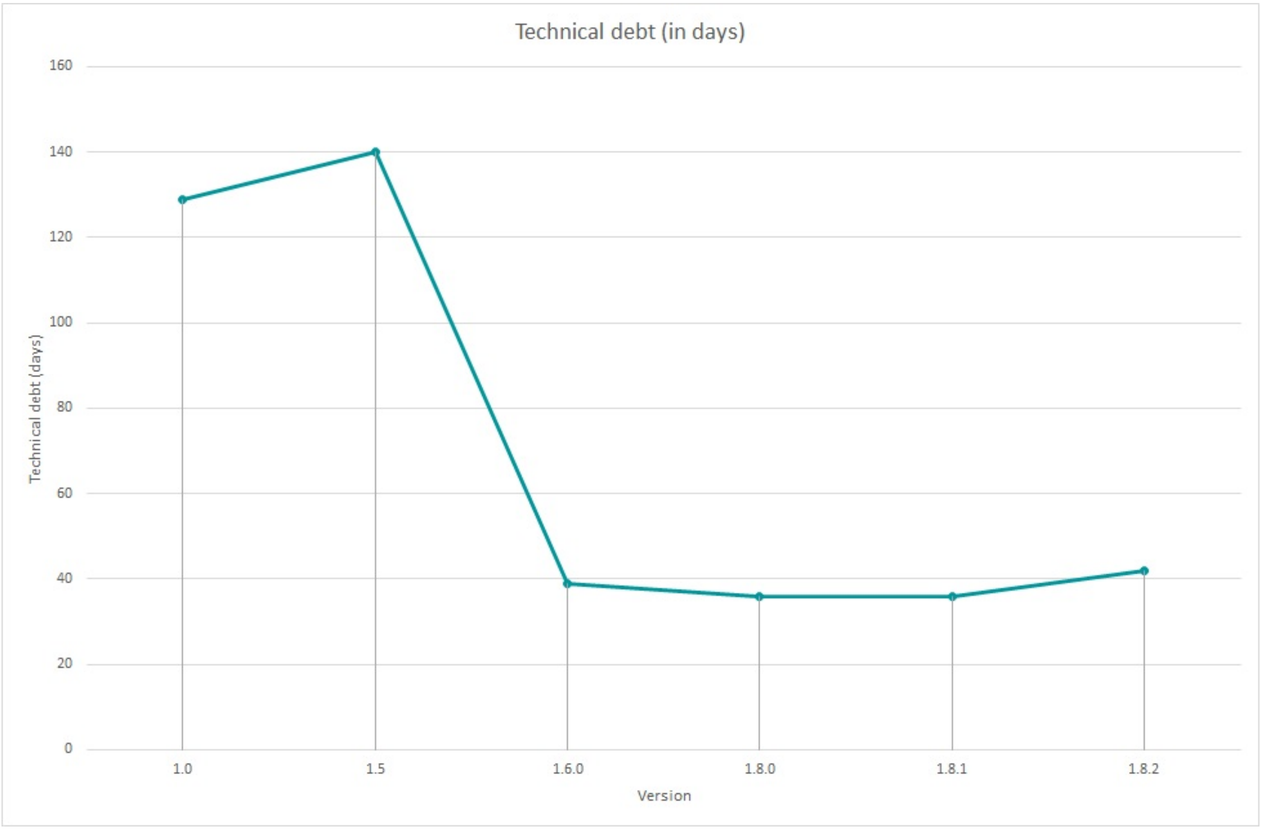


Figure 8: Amount of Technical debt in days over the different releases found by SonarQube.

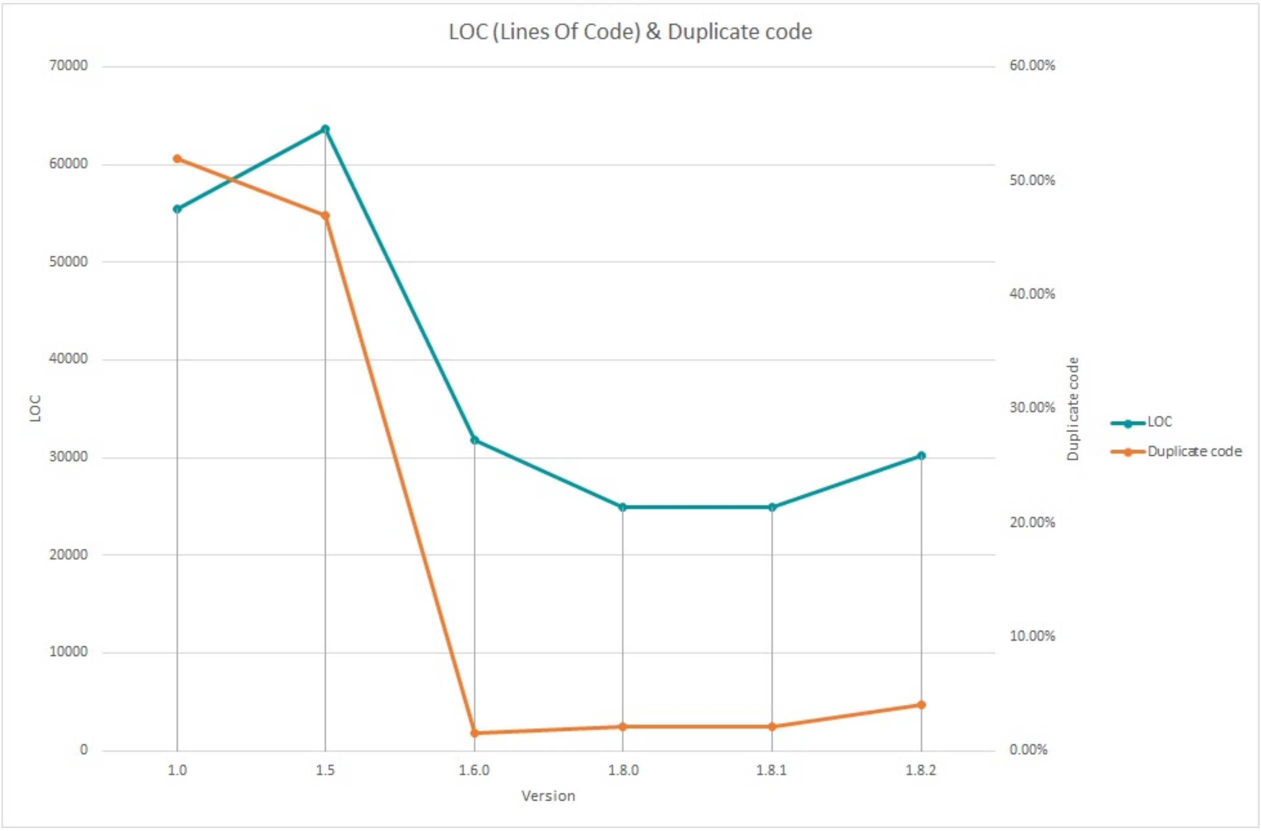


Figure 9: Lines of code and duplicate code over the different verions of Arduino.

The increase of debt after the first release can be seen, after which it drops off sharply with release 1.6\. The same can be said about the bugs and vulnerabilities, this is also attributed to the number of lines of code. The sharp decrease might be a refactoring of the code and project in a whole. The bugs, vulnerabilities and technical-debt is generally stable (but still quite substantial) over the last few releases. The size of the project is increasing again in the latest (unreleased) version.

It can be concluded that technical debt in Arduino has been present from the start and has evolved over time along with the project. The biggest issues concerning technical debt are the missing of documentation, the tests that are not working because the building of the project is not documented and multiple bugs that are old. There is no indication the developers actively think about technical debt, except for commenting on pull-requests. As long as there is almost no documentation the technical debt can only grow.

Deployment view

The deployment view defines the aspects of the system that are important after the system has been built. At this moment it needs to pass validation tests in order to transition it to live operation [2]. For the Arduino software this contains the runtime dependencies the system has and the hardware that is needed for the system to run.

Third-party dependencies: During runtime Arduino has several third-party software dependencies. However, the user does not see these dependencies and does not have to download anything else then the Arduino software. The dependencies are all built into the Arduino software.

Table 7: Third party dependencies during runtime

Third party dependencies	**Role**
C and C++	Language used for the programs that can be written.
GNU toolchain	Compiling and linking with a program stub the main() to an executable cyclic executive program.
avrdude	Converts the executable code into a text file in hexadecimal encoding.
Bossa	Flash programming utility for Atmel's SAM family of flash-based ARM microcontrollers.
AVR Libs	C library for use with GCC on Atmel AVR microcontrollers.
OpenOCD	Open on-chip debugger.
Java	Some standard Java JDK libraries are imported. For example a bug concerning the save-as window shows Arduino is dependent on Java updates [6].

Hardware needed: In order to work with Arduino several hardware tools are needed. Without this hardware the user would have no use of the Arduino software, only the combination of the software and the hardware is valuable. For the most basic project only an Arduino or Genuino board is needed. However, when the user wants to create a useful project the following components might be needed:

- Arduino boards
- Breadboard
- Set of resistors
- Jumper wires
- Diodes
- LEDs
- Buttons

This list is only an example of what can be needed. Based on the project an user wants to carry out, the user would need different supplies. Also, there are a lot of different versions of the de Arduino board which can be chosen from [9]. Arduino offers serveral basic kits containing useful components, see figure 10. An example of an Arduino project is making a simple Arduino alarm system, the hardware components you need for this are an Arduino boards, a ping sensor, a pieze buzzer and a LED strip light [10].

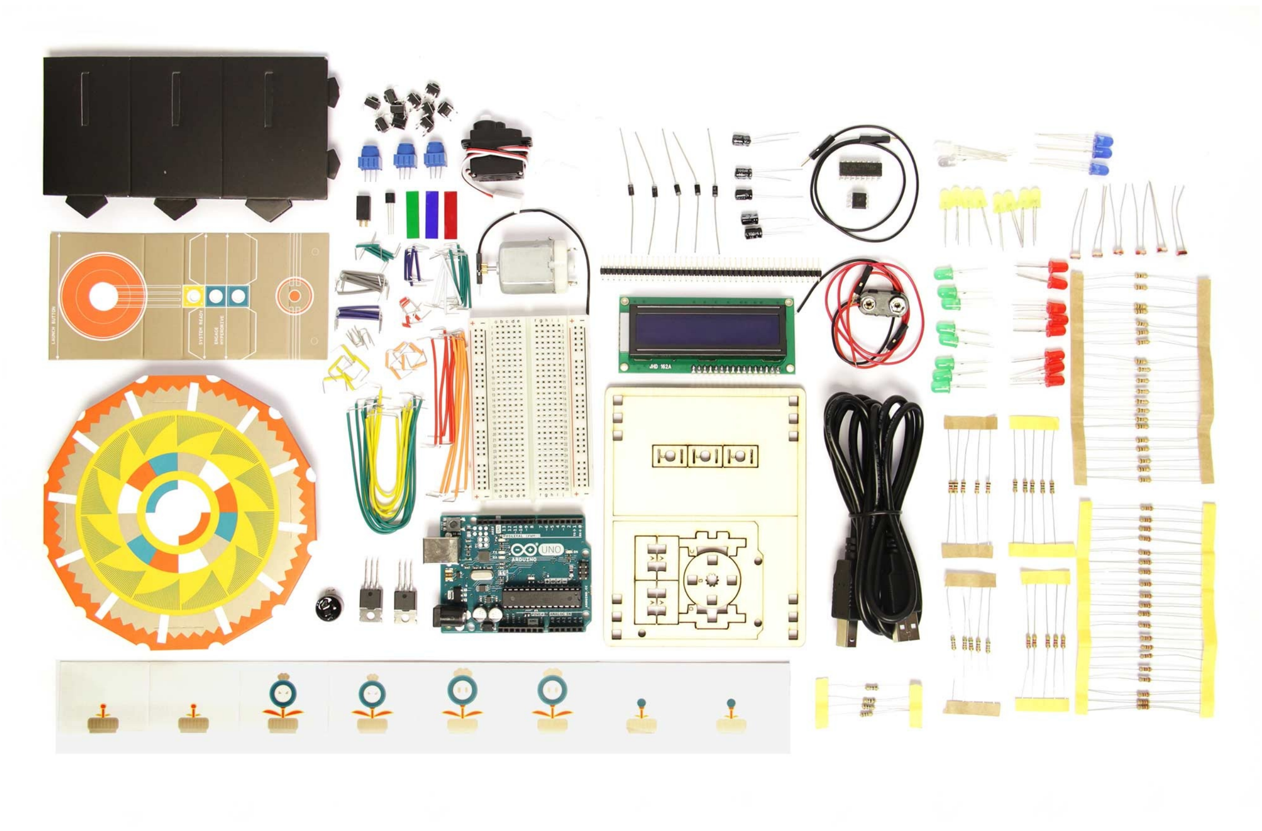


Figure 10: Example of hardware needed for Arduino, Arduino basic kit

Evolution perspective

The evolution perspective deals with concerns related to evolution during the lifetime of a system. This is relevant to Arduino because it is a system where a large amount of change needs to be handled [2].

Throughout the years, Arduino has evolved a lot. At this moment Arduino has 73 releases, which started with release 0002 in October 2005 and the latest release 1.8.1 in January 2017\. However the real first release is 0001, but this can't be found on the Github repository anymore. This is probably because it was an alpha version and "it's a terrible hack" as is described in version 0002\. In order to understand the changes in the various releases, the changelogs and release notes have been analysed. These were found on the Arduino release page on Github [3].

The changes to a new Arduino release are divided into four categories by the developers:

Core: Most changes in the core are related to the architecture of the project. The architecture is divided into three groups AVR, SAM and SAMD. The improvements are listed to be in one of these groups. The three architecture groups refer to the different microcontroller boards that can be used. Another important change in the core is the updating of libraries used by Arduino. Finally, there are also some changes in the core related to refactoring.

To sum up, the following labels are used in Core:

- AVR
- SAM
- SAMD
- Libraries
- Refactoring

IDE: The IDE is what the users work with and a lot of changes are made here regarding the GUI. These changes have a direct benefit for the users. The power of the users is substantial as stated in our power interest matrix.

Changes to the IDE are for example: improving the layout, updating the sketch build process, fixing command-line dependency issues and fixing cross-platform dependency problems. These problems are quite often posted by users on the Arduino Forum or on the Issue Tracker on GitHub.

Libraries: Arduino uses a lot of libraries to provide extra functionality in the sketches. (Recall that: Arduino uses sketch files in which the users write their code) The changes in the library category are thus updating to a new version, adding new libraries, bug fixes and adding support for libraries.

Firmware: This is the category that changes the least. Firmware needs to be maintained in order to upload code to the Arduino boards correctly. The changes to the firmware are often small and not noticed by the users of the IDE.

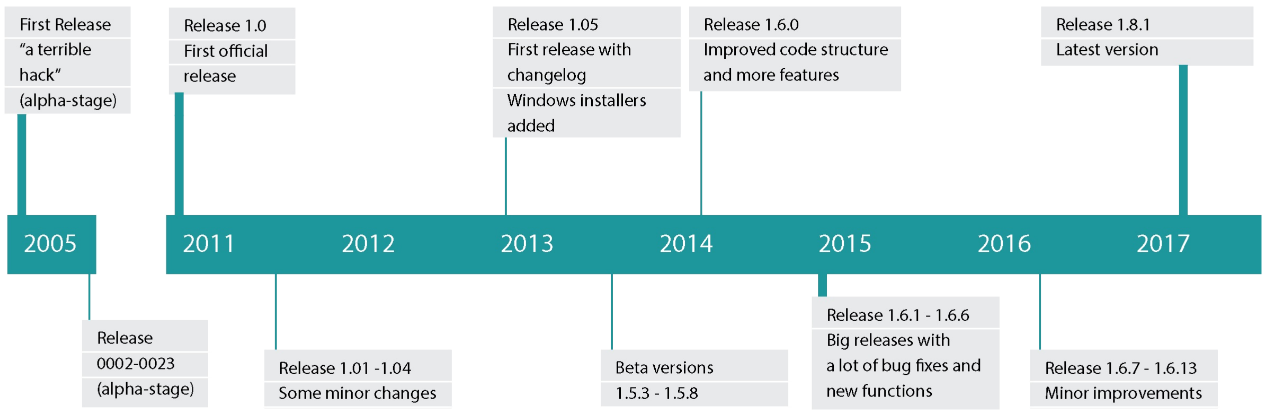


Figure 11: Releases of Arduino.

The releases of Arduino started in 2005 with a "hack just getting everything to work". It took them until 2011 to construct the first official release. After that a lot of releases followed. With a peak of big releases in 2015.

The next release is 1.8.3 which will probably contain the CONTRIBUTING.md file but surely will have the solution to the Save-as Bug since it is merged and assigned as a Milestone for Release 1.8.3.

Conclusion

This chapter summarised and analysed the ins and outs of the Arduino system and what keeps it running. Stakeholders in the project, company-wise as well as users and external developers are identified. The strategy is created with functionality, ease of use and cheap hardware in mind. This is found in the way the code is written: no guidelines or code conventions are used and focus on functionality.

During the analysis of the system it is found that, although the technical debt has decreased compared to the first few releases, there is still a lot of debt left. The single biggest problem that is encountered during the analysis is documentation. The repository only has minimal information on how to build Arduino and no information on how to contribute. There are no real code guidelines and the code only gets tested on functionality.

Some suggestions that could be made to help developers in the future include:

- Improve the documentation on coding, pull-requests, building and testing;
- Improve the test coverage of the project to reduce the chance of faulty functions or code breaking;
- Reduce the technical debt by fixing code smells and coding guidelines, this will improve readability of the code which in turn can improve productivity.

Despite the bad documentation and ugly code, the Arduino system is very popular. The simplicity and ease of use together with the low price make this platform very appealing. This is very visible on the forum as well as on the repository, both places get lots of traffic every day.

References

1. http://medea.mah.se/2013/04/arduino-faq. FAQ by Arduino founder David. (2017).
2. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
3. https://github.com/arduino/Arduino/releases. Arduino release page. (2017).

4. https://www.arduino.cc/en/Main/AboutUs. Arduino about page. (2017).
5. https://github.com/arduino/Arduino/graphs/contributors. GitHub contributors graph. (2017)
6. https://github.com/arduino/Arduino/issues/5191. Example dependency on Java. (2017).
7. https://github.com/arduino/Arduino/wiki/Building-Arduino. Description on how to install development tools. (2017).
8. https://github.com/arduino/Arduino/blob/master/README.md. README of Aruino, lists the third-party software dependencies. (2017).
9. https://www.arduino.cc/en/main/boards. Different Arduino board versions. (2017).
10. http://www.makeuseof.com/tag/how-to-make-a-simple-arduino-alarm-system/. Exampe of an Arduino project, how to make an Arduino alarm system. (2017).
11. https://pitchbook.com/profiles/arduino-profile-investors-funding-valuation-and-analysis. Company profile Arduino on Pitchbook. (2017).

Gradle: adaptable, fast automation for all

Lars van de Kamp, Ingmar Wever, Julian Hols & Hugo Bijmans

Abstract

Gradle is a build automation tool that builds upon the concepts of Apache Maven and Ant. The tool, written in Java and Groovy, allows packaging of software to be deployed on any platform. The open source project that started in 2007 now contains over 400.000 lines of code, almost 1400 plugins made by 266 different contributors. Gradle flourished and instantiated the company Gradle Inc. consisting of most of the core developers of the open source project. In this chapter, the overarching architecture of Gradle is analysed using different views and perspectives as defined by Rozanski and Woods [1]. It concludes that Gradle has a very up-to-date architecture and values proper design and testing to preserve long-term maintainability of the project. The chapter as a whole can serve as a helpful introduction to prospective developers looking to better understand the architecture to which they might contribute.

Table of contents

Introduction

Every year, the size of software projects increases rapidly. This increase demands a more flexible build strategy. In 2007, the founders of Gradle came up with an innovative way of performing software builds that suits the needs of these larger projects. They use a Groovy-based domain-specific-language (DSL) instead of XML in combination with directed acyclic graphs as the fundamentals of their system [2]. The Gradle build tool is able to determine which tasks to run in what order, and identify which sections remained stable and do not need to be rebuilt. By doing so, build time can be drastically reduced for small changes.

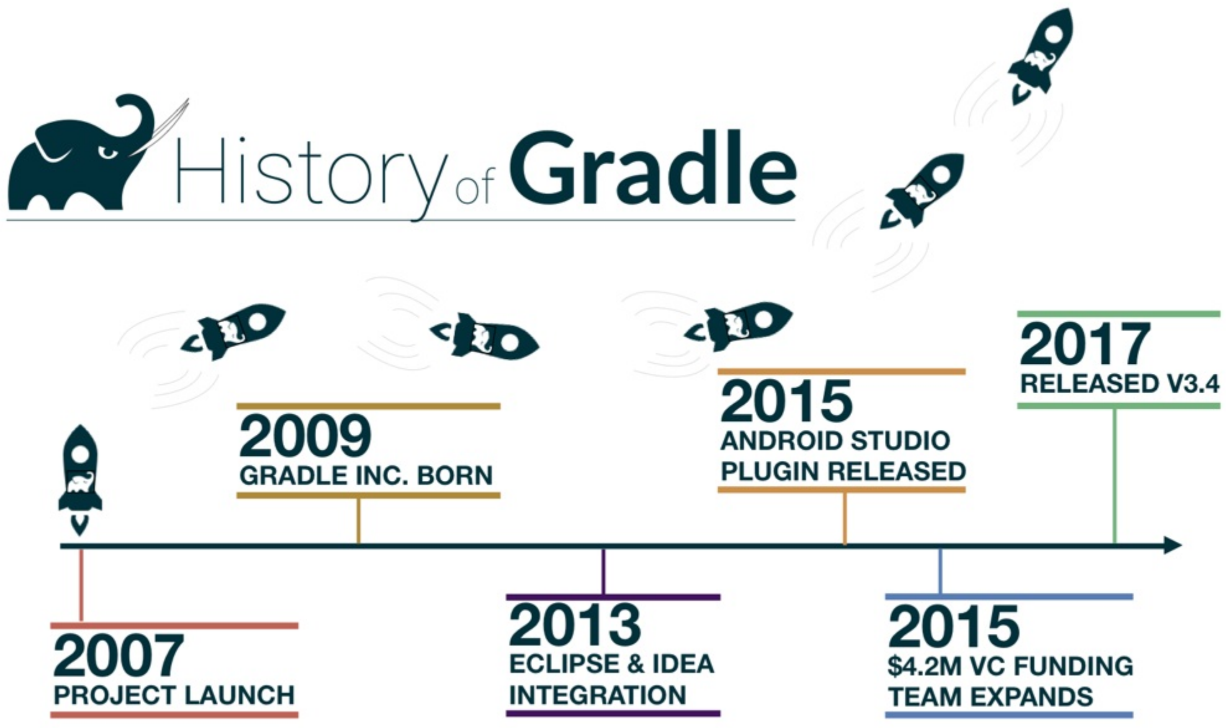

History of Gradle
2007
PROJECT LAUNCH
2009
GRADLE INC. BORN
2013
ECLIPSE & IDEA INTEGRATION
2015
ANDROID STUDIO PLUGIN RELEASED
2015
$4.2M VC FUNDING TEAM EXPANDS
2017
RELEASED V3.4

Figure 1: The history of Gradle

As shown in Figure 1 Gradle started as a two-man project in 2007, and has grown to become a tool used by industry renowned players like Google and Netflix [3]. Gradle Inc., the company that evolved from the project, is now the employer of a dozen developers all dedicated to improving the platform [4]. In 2015, the company received an investment of $4.2 million [5] to expand the company and improve its product. Nowadays, the Gradle build tool is used by a wide group of users, ranging from individual developers to major software development companies such as LinkedIn, PayPal and Adobe. In addition, an extensive user manual, video tutorials and integration with IDEs like IntelliJ IDEA and Eclipse make it easy for developers to integrate the tool into their workflow.

This chapter gives an overview of the overarching architecture of the Gradle project. It sets the scene by introducing the project and discussing its stakeholders. It then takes on different viewpoints and perspectives to analyse Gradle's performance, in addition to discussing the technical debt hidden in the depths of the codebase.

Stakeholders view

In a large open source project like Gradle, multiple types of stakeholders can be identified. Table 1 introduces the stakeholders involved in the Gradle project.

Stakeholder class	**Description**
Acquirers	Gradle Inc. sells cloud services, based on the Gradle project. Trueventures and DCVC are both investment companies with mostly financial influence and interest. Heavybit Industries provides a program to bring development products to market and scale the company, in return for a small percentage of equity.
Communicators	The developers in the core development team of Gradle are also the major communicators. Hans Dockter and Adam Murdoch are responsible for most of the user guide. [6]
Development & Testing	Most of the developers and testers are employees of Gradle Inc. However, individual GitHub contributions and extensions made by others in the industry are present as well. For example, Google [7] and Netflix [8] made plugins for Gradle.
Suppliers	Gradle is built upon Java and Groovy, heavily supported by its Spock testing framework and several code quality tools (e.g. Checkstyle, PMD, FindBugs and Codenarc).
Support	An active community of both users and moderators are part of the support staff. This support is mostly provided through the Gradle discuss page.
Users	Gradle has a very broad user base which can be separated into three categories: individual programmers, small projects needing insights into their builds, and companies which need insight into their software development. Since users are predominately developers, they can also be considered as system administrators of their own particular build.
Competitors	Other parties delivering similar services are for example Maven and Ant.
Maintainers	The in-house developers together with the Gradle community of users, contribute to the maintenance of the system. Overall, Benjamin Muschko and Cédric Champeau seem to invest the most effort in maintaining the project.

Table 1: Stakeholders of the Gradle project

The stakeholders are visualized in Figure 2, their respective power and interest in the Gradle project are shown in Figure 3.

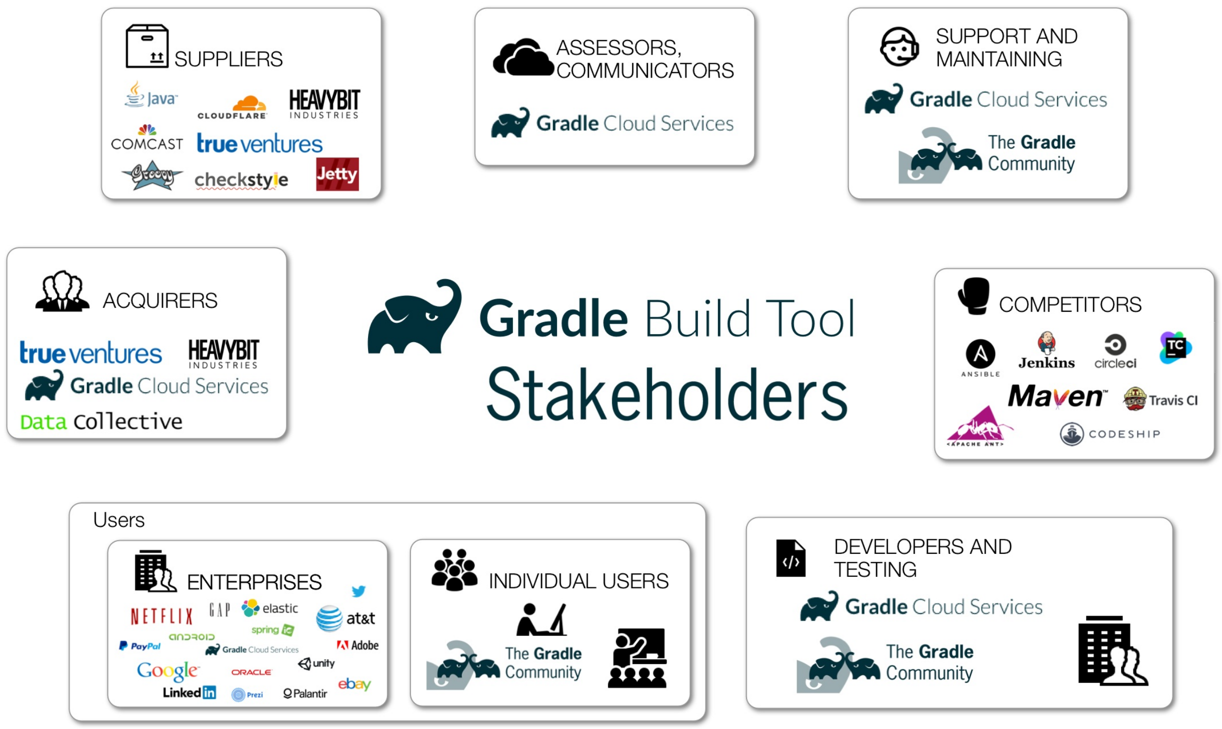


Figure 2: An overview of all the stakeholders involved in the Gradle project

Power-Interest grid

In the stakeholder analysis, many different actors with different roles, power and interest have been analysed and visualised in Figure 3. The main stakeholder of Gradle is Gradle Inc., which has the most power and interest in the platform. The company's key executives, Adam Murdoch and Hans Dockter, have the most influence and are considered to be the main architects of the platform. The venture capital companies that invested in Gradle have been assigned as having high power, but moderate interest. The diversified portfolios owned by these firms limits the interest in the company to only a moderate status.

Since Gradle is built upon Java and Groovy, these programming languages can be seen as more powerful actors than other suppliers, having very little interest. Heavybit industries can also be seen as a supplier, and since they possess a warrant and house the company, they are presumed to have more power and interest than most suppliers. The developers, testers and maintainers can be placed in the middle of the spectrum, all with some power and interest.

The two least powerful stakeholders are the individual users and the competitors. Whereas the last one has the highest interest, since every mistake made by Gradle could be in favour of them. Companies who use Gradle are considered to have more power than individual users, since they have the knowledge and manpower to put pressure on the development of Gradle, contributing themselves or by demanding extra features by the development team inside Gradle Inc.

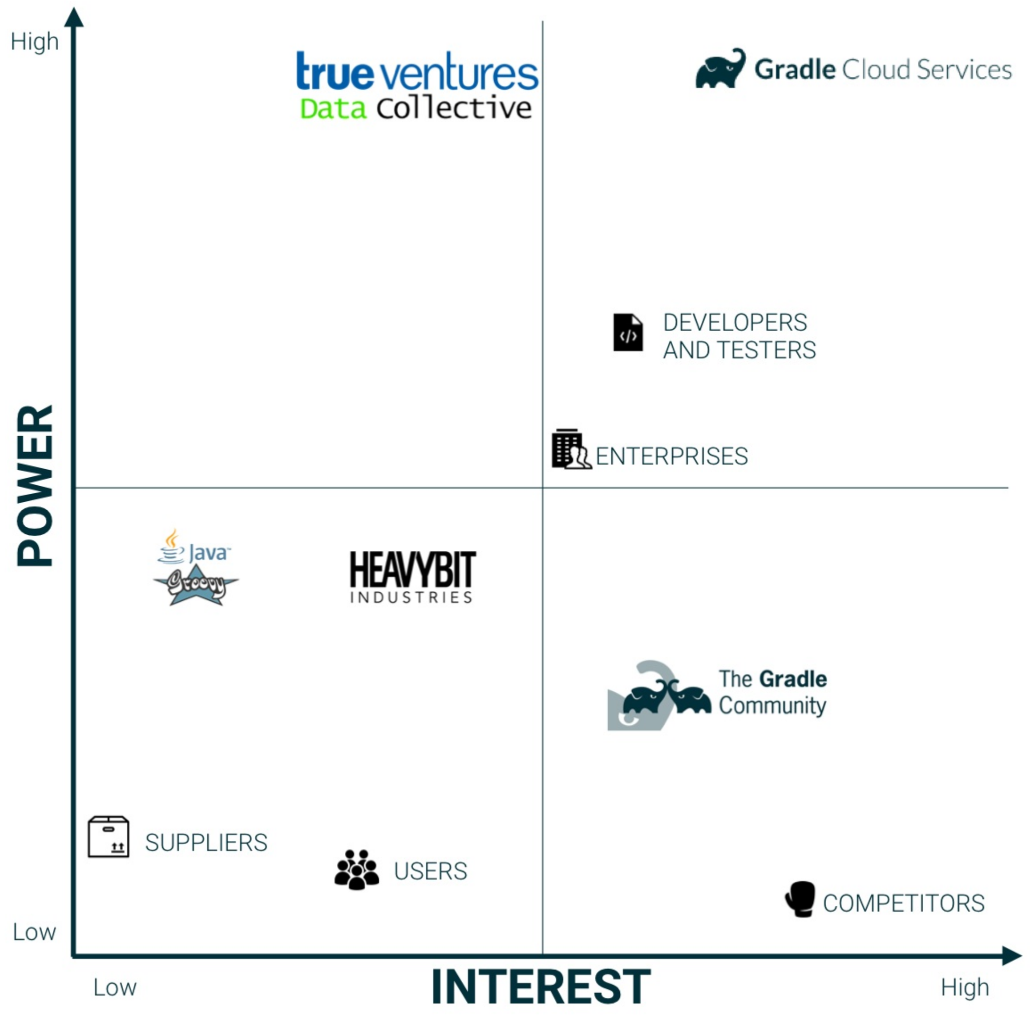


Figure 3: Power-Interest grid for the stakeholders of Gradle

Context view

The context view describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts) [1]. This section examines Gradle's scope, its dependencies on others and the interaction with other parties.

System scope & responsibilities

Gradle aims to build anything, automate everything and deliver faster [3]. The goal of the system is to create an environment which allows developers to code as flexible as they want, and to let them build their software as fast as possible. Since more players (competitors) are in the field, Gradle strives to deliver the best performance. Therefore, Gradle introduced incremental build strategies, allowing smart algorithms to speed up the building process. Thus, the scope of Gradle is to provide developers with a flexible and easy to use building tool to deliver their software faster.

External entities and interfaces

Gradle is a widely-used build tool out of which a company, Gradle Inc., emerged. As one can imagine, a software project like this cannot be developed without external libraries, tools and frameworks. On the other hand, many companies cannot develop their software without Gradle. These external relations are examined in this section. Below, these are elaborated upon and afterwards visualised in Figure 4.

- Written in Java and Groovy
- Windows, Linux and macOS are all supported
- Built by itself, using the Gradle Build Tool
- Supports many programming languages, enabled by their respective plugins [10]
- Plugins also allow editing in an IDE of choice
- Active development team of 30-40 core developers and more than 200 contributors from the open source community
- A diverse spectrum of users, from individual developers to major enterprises such as Google and Netflix
- Continuous integration using TeamCity CI
- A GitHub repository filled with code, plugins and many issues is used to host the code base [9]
- Communication and support is provided via Github, Gradle discuss and Twitter
- For testing purposes, the Spock Framework is used, which provides enterprise testing, behaviour driven testing and mocking and stubbing
- The project is licensed under the Apache License, Version 2.0, a free software license written by the Apache Software Foundation (ASF)

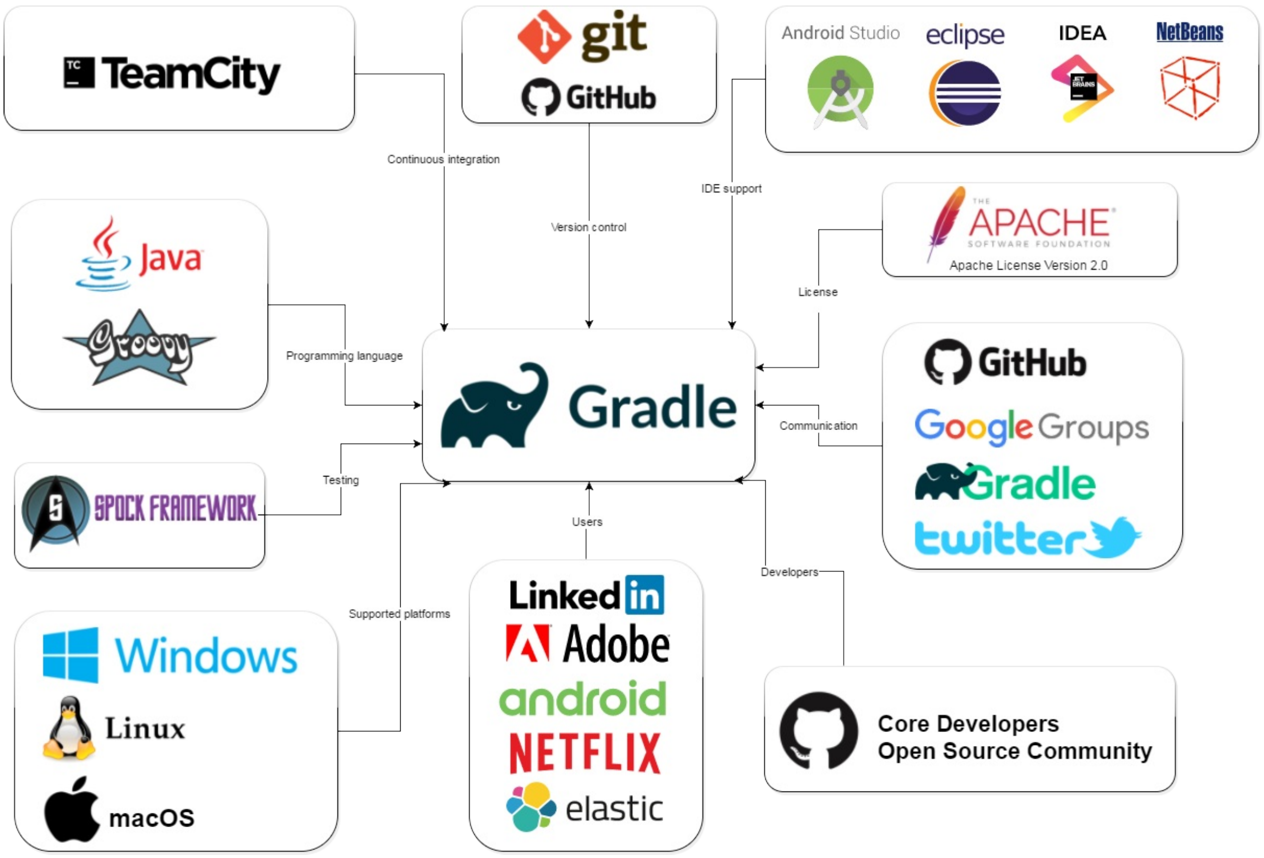


Figure 4: The Context View of Gradle

Evolution perspective

This section analyses the evolution of the Gradle Build Tool. The evolution perspective focuses on identifying the ability to be flexible in the face of inevitable change. As discussed by Rozanski and Woods, a flexible system should be able to deal with all possible types of changes that it may experience during its lifetime [1]. Therefore, the changes throughout the lifetime of the project are analysed, and the mechanisms in place to provide flexibility are discussed.

Gradle updates their current version number according to the semantic versioning convention [19]. Most Gradle releases can be categorised into two main categories: major updates that symbolize a new backwards compatibility baseline, and new versions containing novel features and bug fixes. The first one corresponds to the *major* indicator in the semantic version convention, the latter corresponds to the *minor* type. There might also be a third version number that represents a *patch*, which is incremented by small bug fixes that are merged into the Master branch.

The first type has only occurred three times in the history of Gradle. The latter on the other hand, has an average frequency between 6-8 weeks [20] despite efforts to decrease this to 4-6 weeks as stated in the release notes of Gradle 2.7 [21]. Figure 5 gives an overview of the different releases and mentions the changes with the largest magnitude of change.

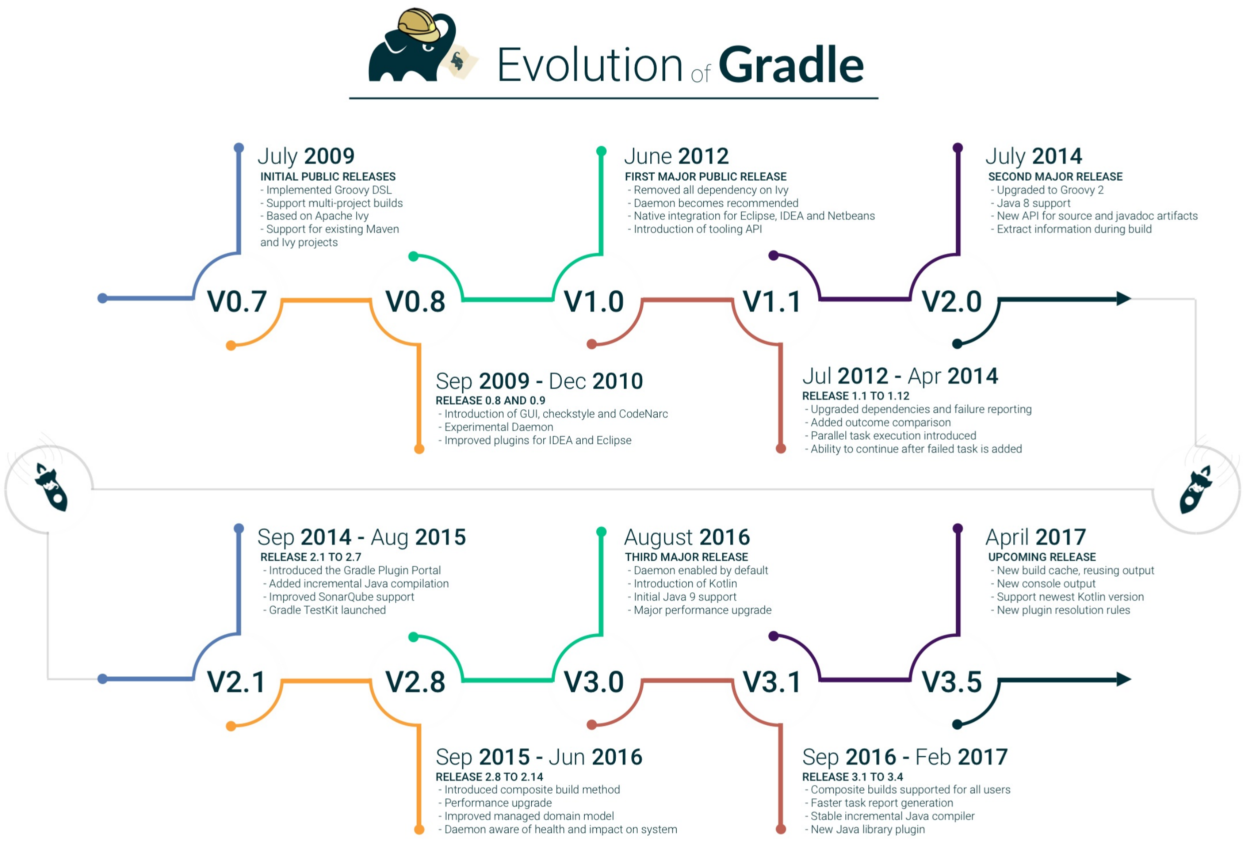

Evolution of Gradle
July 2009
INITIAL PUBLIC RELEASES
- Implemented Groovy DSL
- Support multi-project builds
- Based on Apache Ivy
- Support for existing Maven and Ivy projects
June 2012
FIRST MAJOR PUBLIC RELEASE
- Removed all dependency on Ivy
- Daemon becomes recommended
- Native integration for Eclipse, IDEA and Netbeans
- Introduction of tooling API
July 2014
SECOND MAJOR RELEASE
- Upgraded to Groovy 2
- Java 8 support
- New API for source and javadoc artifacts
- Extract information during build
V0.7
V0.8
V1.0
V1.1
V2.0
Sep 2009 - Dec 2010
RELEASE 0.8 AND 0.9
- Introduction of GUI, checkstyle and CodeNarc
- Experimental Daemon
- Improved plugins for IDEA and Eclipse
Jul 2012 - Apr 2014
RELEASE 1.1 TO 1.12
- Upgraded dependencies and failure reporting
- Added outcome comparison
- Parallel task execution introduced
- Ability to continue after failed task is added
Sep 2014 - Aug 2015
RELEASE 2.1 TO 2.7
- Introduced the Gradle Plugin Portal
- Added incremental Java compilation
- Improved SonarQube support
- Gradle TestKit launched
August 2016
THIRD MAJOR RELEASE
- Daemon enabled by default
- Introduction of Kotlin
- Initial Java 9 support
- Major performance upgrade
April 2017
UPCOMING RELEASE
- New build cache, reusing output
- New console output
- Support newest Kotlin version
- New plugin resolution rules
V2.1
V2.8
V3.0
V3.1
V3.5
Sep 2015 - Jun 2016
RELEASE 2.8 TO 2.14
- Introduced composite build method
- Performance upgrade
- Improved managed domain model
- Daemon aware of health and impact on system
Sep 2016 - Feb 2017
RELEASE 3.1 TO 3.4
- Composite builds supported for all users
- Faster task report generation
- Stable incremental Java compiler
- New Java library plugin

Figure 5: The Evolution Perspective of Gradle [24]

The main reason for Gradle to innovate and change the project structure is to improve the product. In every iteration, it attempts to add new features and improve performance and user experience significantly. They aim to stay flexible by using their plugin orientated architecture. Gradle has explicitly chosen to limit the features provided by the core module. As stated in the online user guide:

> All of the useful features, like the ability to compile Java code, are added by plugins. Plugins add new tasks (e.g. JavaCompile), domain objects (e.g. SourceSet), conventions (e.g. Java source is located at src/main/java) as well as extending core objects and objects from other plugins [22].

The described plugins can be implemented in any language, as long as the implementation ends up compiled as byte code[22]. The choice for a plugin based model improves several critical aspects of the software project. First of all, the overhead of maintaining similar logic across multiple projects is reduced. Furthermore, it enhances comprehensibility and organisation of the project due to a higher degree of modularization. Finally, it encapsulates imperative logic, allowing the build scripts to be as declarative as they can be [22].

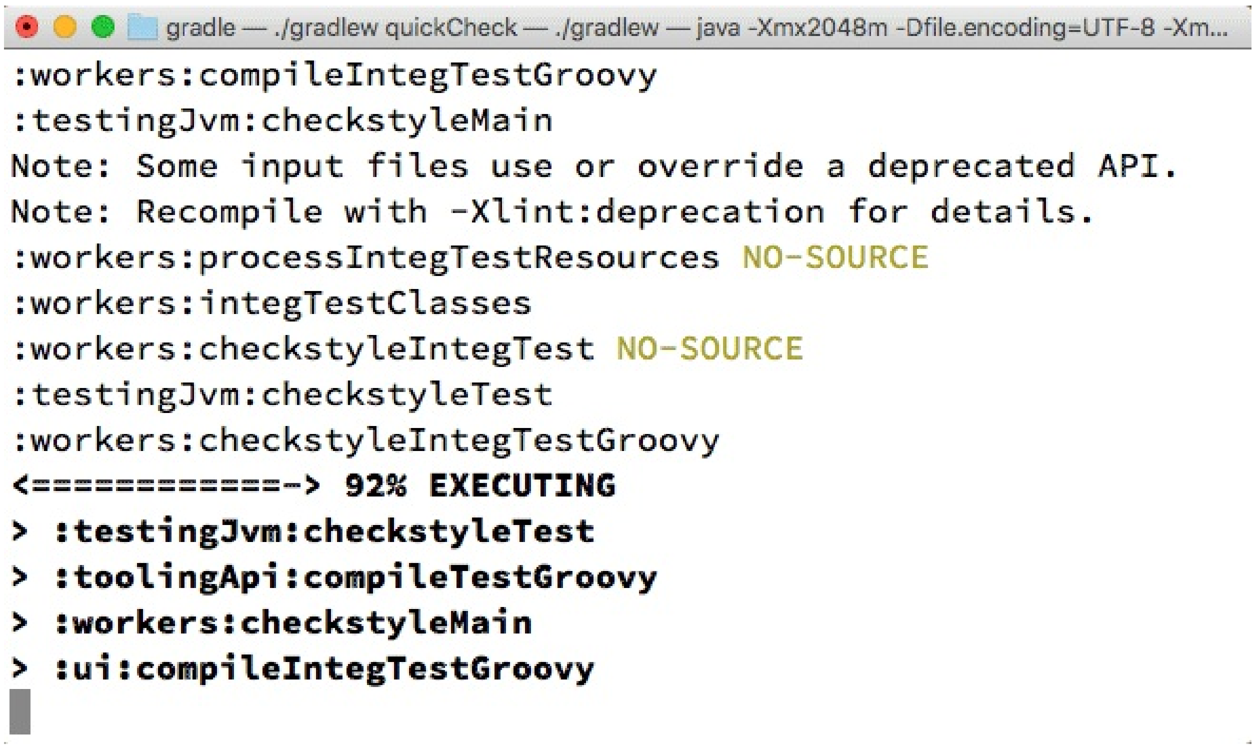
Most importantly, it seems that this strategy has allowed Gradle to be a front runner in the industry. The fast development of a strong and innovative core, combined with a very flexible plugin architecture will allow Gradle to keep moving in the future when facing inevitable change.

Deployment view

In the book Software Systems Architecture, Rozanski and Woods [1] define the deployment view as *"Describes the environment into which the system will be deployed, including the dependencies the system has on its runtime environment"*. Gradle will be deployed on the computer system of the users (developers), or on a continuous integration server on which a project will be built.

The Java Runtime Environment (JRE) is a third-party software requirement for Gradle, and needs to be installed on the system where the project will be utilised. The use of Java makes Gradle deployable on a huge variety of operating systems such as Windows, macOS and Linux as well as different hardware architectures such as x86, x64 and ARM. Java decouples Gradle from the operating systems' environment and hardware components and thus makes Gradle independent of the system and type of hardware on which it is deployed.

A particular issue may arise in the deployment due to the possibility that Gradle projects can be created using a different version of Gradle than the user might have installed. To aid the user in building a Gradle project independent of the original system environment on which the project was deployed, Gradle provides the option to make use of the Gradle Wrapper. The wrapper will take care of the installation of additional tools required, and will make sure to install the right version. A project that includes the Gradle Wrapper can be built on any system that has Java installed. The wrapper is invoked using the `./gradlew <task>` command and works by downloading and installing the version of Gradle the project requires and will thus take care of the deployment for the user. The wrapper is able to verify the downloaded files by calculating and comparing the expected file checksum against the actual one. This feature increases security and protects the software from tampering with the downloaded Gradle distribution. The Gradle Wrapper is the preferred way of starting a Gradle build [17] and is visualised in Figure 6, this also shows the parallelization of Gradle processes.



```
gradle — ./gradlew quickCheck — ./gradlew — java -Xmx2048m -Dfile.encoding=UTF-8 -Xm...
:workers:compileIntegTestGroovy
:testingJvm:checkstyleMain
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
:workers:processIntegTestResources NO-SOURCE
:workers:integTestClasses
:workers:checkstyleIntegTest NO-SOURCE
:testingJvm:checkstyleTest
:workers:checkstyleIntegTestGroovy
<=============-> 92% EXECUTING
> :testingJvm:checkstyleTest
> :toolingApi:compileTestGroovy
> :workers:checkstyleMain
> :ui:compileIntegTestGroovy
```

Figure 6: The Gradle wrapper in action

The Java Virtual Machine (JVM) requires a non-trivial initialization time. As a result, it can take a while to launch a Gradle build process. As a solution, Gradle provides the Gradle Daemon. The Gradle Daemon is a long-lived background process that will execute builds much quicker than invoking the Gradle process the normal way [18]. This is possible by avoiding the JVM start-up costs, as well as by maintaining a build cache that stores data about the project in memory. This build cache enables Gradle to make use of incremental builds to improve its performance. By using these incremental builds Gradle identifies input or output sections of the build process that have not changed. If these are present, Gradle can skip the build of that task and reuse the existing output from the previous build. The daemon is now enabled by default, but is recommended to be disabled for Continuous Integration (CI) and build servers.

Development view

This particular view highlights the concerns and interests of the developers and testers of the project. The different modules in the project have been identified, the file structure has been researched and important standardisations of processes are discussed.

Modules structure

The top layer division can be seen in Figure 7. It shows that the Gradle core and the plugins rely on different external dependencies. The plugins are dependent on the API provided by Gradle to connect to the core modules. All three of these different components are controlled by the Gradle build tool that manages the interaction between these subsections. Finally, the Gradle core also has internal dependencies which will be discussed next.

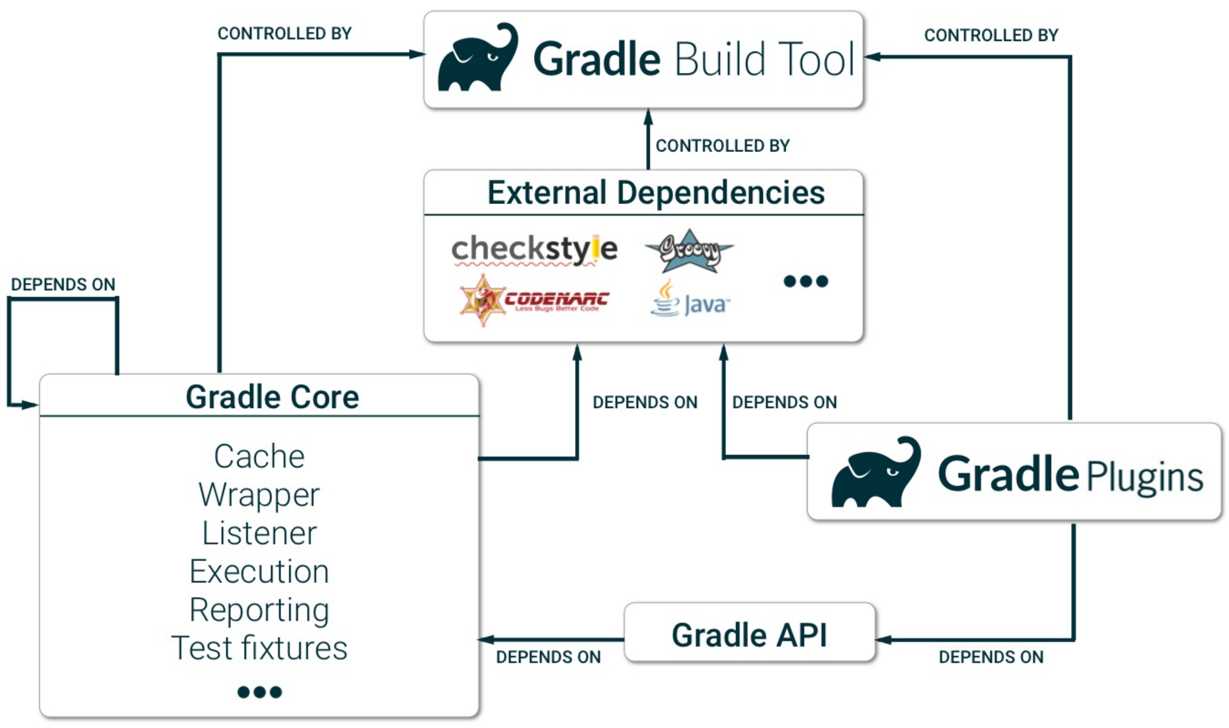


Figure 7: The module structure view of Gradle

A more detailed view of modules in the source code can be found in Figure 8. Due to the flexibility of the Gradle build tool, the core part of the source code is also very flexible, thus fragmented. The main building blocks are the `exhaustion` , `process` , `internal` and `initialization` blocks, which all use elements of the `cache` and `caching` modules to perform their tasks. Common processing parts, like `reporting` , `util` and the Gradle `API` , are also used by the main blocks of source code. The actual behaviour of those core building blocks is influenced by settings located in the `configuration` folder. Finally, all plugins are accessed through the `API` in order to work with the core source code.

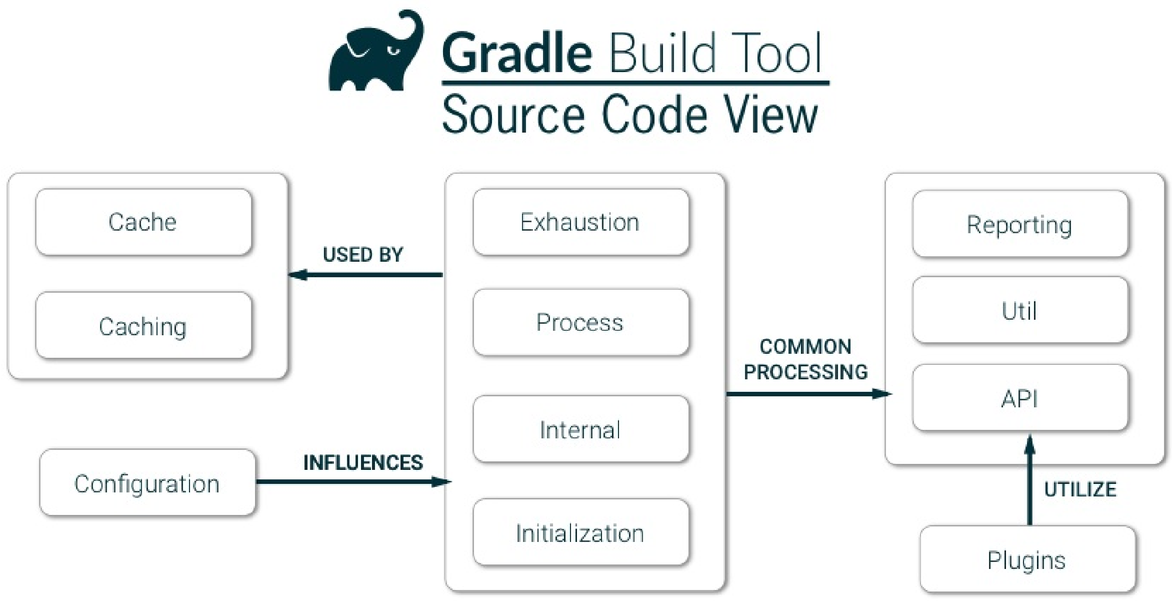


Figure 8: A more detailed view at the core source code of Gradle

Common processing

Just like any other large system, Gradle has separate code modules for tasks which are common to other modules. A few of Gradle's common used modules have been identified in the `subprojects/core` folder:

- A central, common logger is used. The Simple Logging Facade for Java (SLF4J) is used to keep track of all the logs made by Gradle. This is also used by the `-debug` option to allow the user to easily debug their code.
- Moreover, multiple utilities are made by the team to be used throughout the project. In the `src/util` folder, many utilities are listed, such as a `nameMatcher` , a `clock` and a `swapper` , which can be easily used by other modules.
- Because of Gradle's high dependency on external plugins, its API can also be considered as common process. The `tooling-api` provides the user with the needed dependencies on those plugins.

Standardization of design

Since Gradle is an open source platform, everyone is free to contribute to the repository on GitHub. Seeing as multiple contributors are influencing Gradle, the core developers have standardised aspects of the design of the system to make it as maintainable, reliable and technically cohesive as possible. There is not a lot of information available about the general design of the platform, but the core developers have made some guidelines for new contributors to keep the overall code quality high and the licences applicable. The most important aspects of contributing to Gradle are discussed in the `CONTRIBUTIONS.MD` file:

- Contributors have to use git and have a GitHub account to be able to fork the GitHub repository `gradle/gradle`
- Use a text editor or IDE (IntelliJ IDEA CE is recommended) to make changes
- Document changes in the User Guide and DSL Reference (under `subprojects/docs/src/docs`)
- Users are not allowed to put in their names in `@author` Javadoc field
- Write a solid commit message, explaining what has been done in the proposed change
- Sign the CLA, which defines the terms under which the intellectual property has been contributed to Gradle
- Make a pull request using the `PULL_REQUEST_TEMPLATE.md` and wait for their code to be reviewed by a Gradle core developer

There are no concrete guidelines of using design patterns in the development of Gradle. However, when examining the code, the *factory*, *builder*, *Singleton* or a combination of these design patterns have been identified many times. For example, the `GroovyCompilerFactory.java` and `DefaultDirectoryFileTreeFactory.java` are two classes built according to the *factory* design pattern.

The implementation workflow is flexible. Core Gradle developers are allowed to commit directly into the master, other contributors have to use pull requests. These pull requests are labelled by developers, assigned to issues (if applicable) and categorised before merging.

Standardisation of Testing

By standardising the test approaches, technologies and conventions, the overall testing process remains consistent and has a higher pace. In this section, Gradle's efforts to achieve this are reviewed.

The Gradle project uses its own Gradle build tool as its build tool of choice, which initiates all tests. All new contributions need to include two things in terms of testing. First of all, it needs to provide new Unit Tests, using the Spock framework for any novel logic introduced. In addition, integration test coverage of the new bug/feature should also be provided. Afterwards, TeamCity automatically checks the compatibility of the introduced code, preventing any unexpected failures. To give some indication of the number of tests that are run by TeamCity, for Windows with Java 1.8, 23.709 tests have been passed successfully [11].

In order to verify existing tests manually, the untested code can be introduced in the `/subproject` folder after which the `./gradlew :<subproject>:check` command can be run. During the build Gradle also executes Checkstyle and Codenarc tests that perform static code analysis. The use of PMD throughout the development process is also encouraged.

Finally, simple contributions are only merged into the project when the build succeeds, the tests are all passed and new test material is provided and reviewed to be in good order. Complex changes must face the promotion pipeline, in which they will be tested more thoroughly in different environments and levels to assure quality [23].

The Spock Framework

Gradle uses the Spock Framework as their testing framework of choice. The framework was initiated by Peter Niederwieser who joined the Gradle team in 2011 and remained an active employee until late 2014. Peter received help from Luke Daley who is still part of the core development team of Gradle Inc. Therefore, the Spock Framework is made by a (former) Gradle employee, but can be considered as an independent piece of software used by Gradle in their development process.

The open source project Spock integrates enterprise testing, mocking, stubbing and does this in a behaviour driven way [12]. It covers a diverse spectrum of test types, ranging from unit testing to integration testing and even functional testing. Figure 9 shows the different tools that can be replaced by integrating the Spock framework.

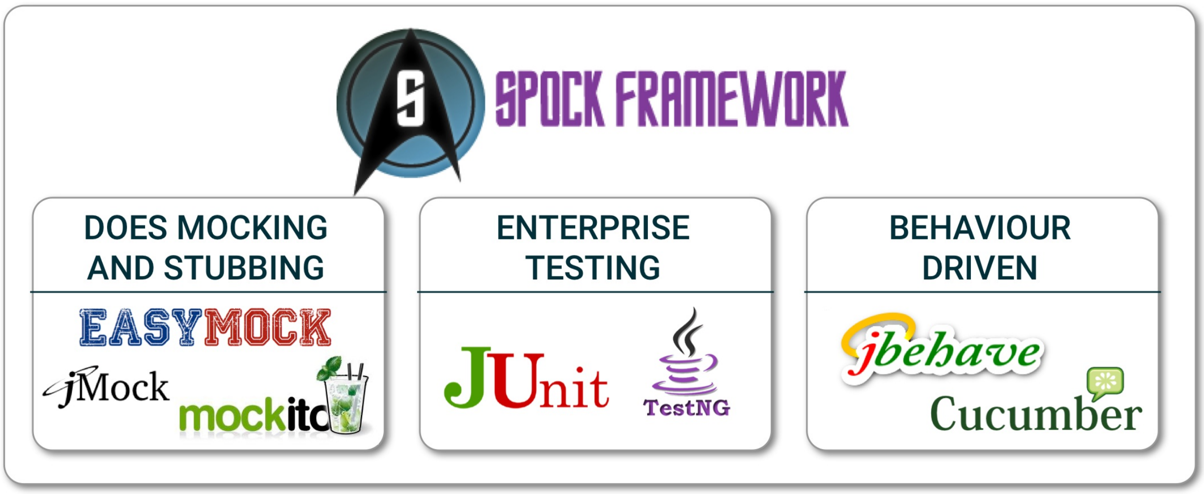
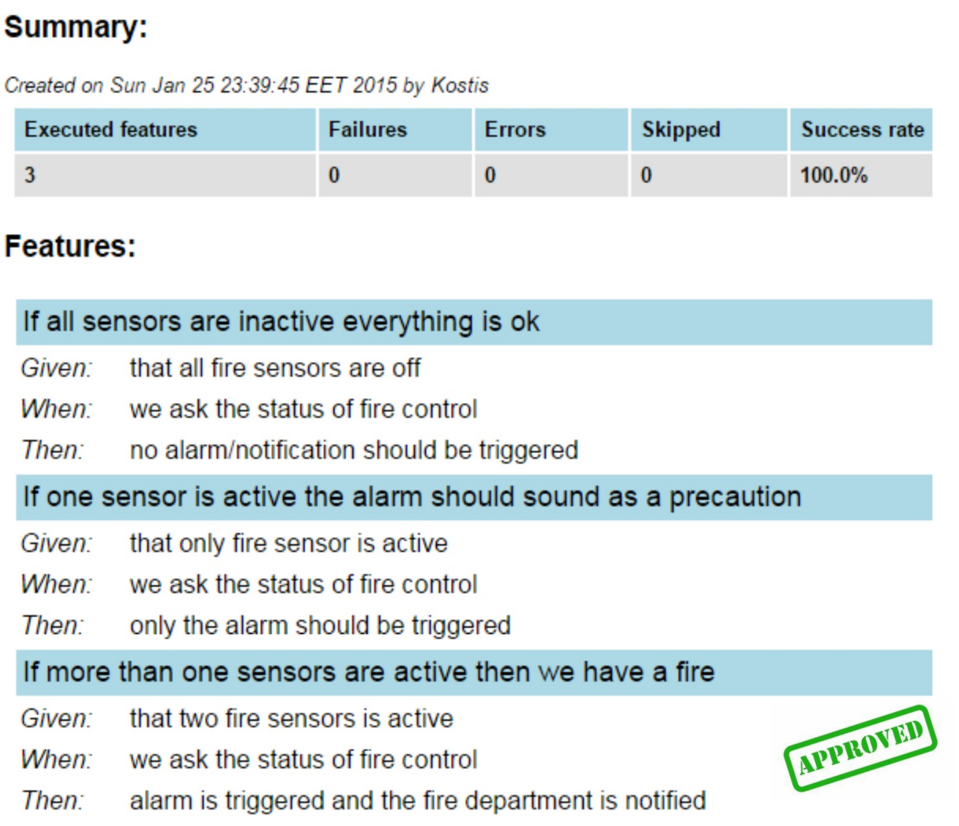


Figure 9: The Spock framework

The framework uses the JUnit runner which allows for easy integration into the system and even facilitates parallel usage of Spock and JUnit tests. Simplicity and readability are key aspects of the framework. The tests can be read like plain text English and the results of a test come in a very clear reporting format, as shown in Figure 10.



Summary:

Created on Sun Jan 25 23:39:45 EET 2015 by Kostis

Executed features	Failures	Errors	Skipped	Success rate
3	0	0	0	100.0%

Features:

If all sensors are inactive everything is ok

Given: that all fire sensors are off
When: we ask the status of fire control
Then: no alarm/notification should be triggered

If one sensor is active the alarm should sound as a precaution

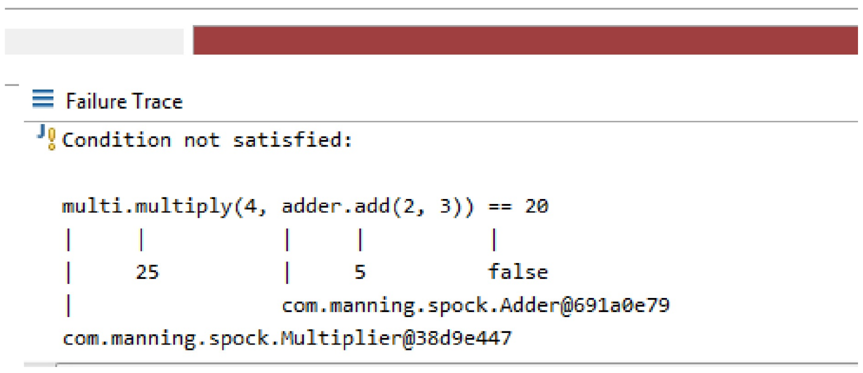
Given: that only fire sensor is active
When: we ask the status of fire control
Then: only the alarm should be triggered

If more than one sensors are active then we have a fire

Given: that two fire sensors is active
When: we ask the status of fire control
Then: alarm is triggered and the fire department is notified

Figure 10: Spock output report summary

Another very important advantage is the ability of Spock to understand the context in failed tests while using data driven testing. As can be seen in Figure 11, the values of the different variables of the failed assert are shown to give a clear overview of the reasoning why the test failed. According to the creators of Spock, only their framework knows the context of the failed test, whereas others do not [13]. The fact that Gradle uses this framework, which promotes company wide testing and understandability, enforces the idea that testing and code quality is the foundation of their development process.



```
Condition not satisfied:

multi.multiply(4, adder.add(2, 3)) == 20
|     |            |     |         |
|     25           |     5         false
|                  com.manning.spock.Adder@691a0e79
com.manning.spock.Multiplier@38d9e447
```

Figure 11: Simple test visualization by Spock

Technical debt

This section focuses on the technical and testing debt present inside the Gradle project. The definition of technical debt according to Techopedia is:

> Technical debt is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution. [14]

To identify technical debt within Gradle, a wide range of options have been used. First, (static) code analysis tools are used to assess code quality. Afterwards manual inspection took place to examine the evolution of technical debt.

Code quality tools

During the development of Gradle, many tools are used to keep the quality high. Static analysis tools are used to detect errors in the source code without running it, and Continuous Integration tools are used to prevent integration problems and allow external build tests. Checkstyle, PMD, Codenarc, Findbugs, and JaCoCo are all incorporated in the process. Gradle takes quality and testing very serious. As discussed before, contributors have to write additional tests and are subjected to code review by core developers in order to get their pull request merged. The Gradle Core developers have even developed their own Gradle Quality Plugin to standardise the output of code quality tools. This tool combines results from CheckStyle, FindBugs, PMD and CodeNarc and categorises them into one summary.

SonarQube analysis

SonarQube is a platform for continuous software quality monitoring. This platform is able to analyse a large software project like Gradle in a matter of minutes. It provides the user with crucial insights about bugs, security and technical debt within the project. Running the SonarQube analysis tool on the latest version of Gradle (v3.4.1) has given the following insights about technical debt. SonarQube reports a vast amount of bugs (1500), vulnerabilities (87) and code smells (7900), ranging from refactor recommendations to limiting the large number of lines in a class to the amount of duplicated code (which represents only 0.6% of the entire code base). Many of the indicated bugs are only minor faults, style issues or false positives.

The technical debt analysis however, provides a rather good insight into this phenomenon. SonarQube uses the SQALE methodology, which is a method to support the evaluation of source code independent of language or code analysis tools. The resulting findings include 113 days worth of technical debt. Moreover, when combining this with the age of the project (8 years as of now) and the fact that it contains over 400.000 lines of code, a maintainability ratio of 0.7% is derived. This results in the highest possible grade, namely an A. An overview visualisation, showing the time needed to fix the discovered code smells can be seen in Figure 12.

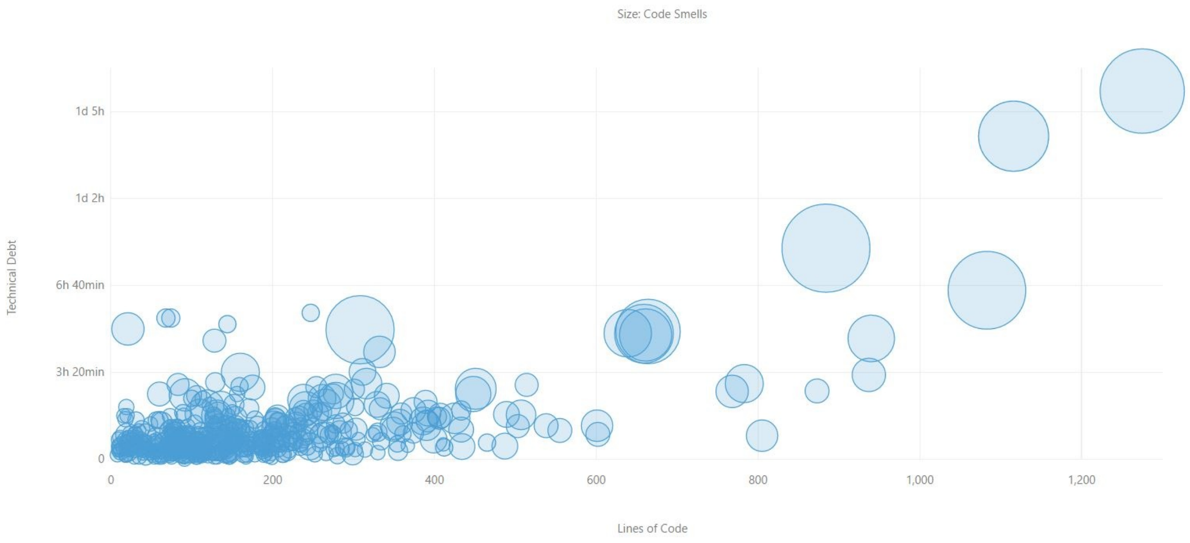


Figure 12: Technical debt in the source code, categorised by lines of code and time to fix

Furthermore, the cyclomatic complexity measure is often used as a metric to assess the complexity of a code base. This measure is calculated by accruing the different possible paths through the source code. In Figure 13 the complexity breakdown of different functions and files is given. The total complexity of the entire code base is 47.408 and the average per class is 6.9. Overall, the complexity is relatively low when comparing the values to those suggested by codecentric [15]. According to SonarQube there is not a single method that exceeds the unmaintainable threshold, with the highest having a complexity of 18.

These insights give additional indications that developers working for Gradle are highly focused on generating long-term, maintainable software.

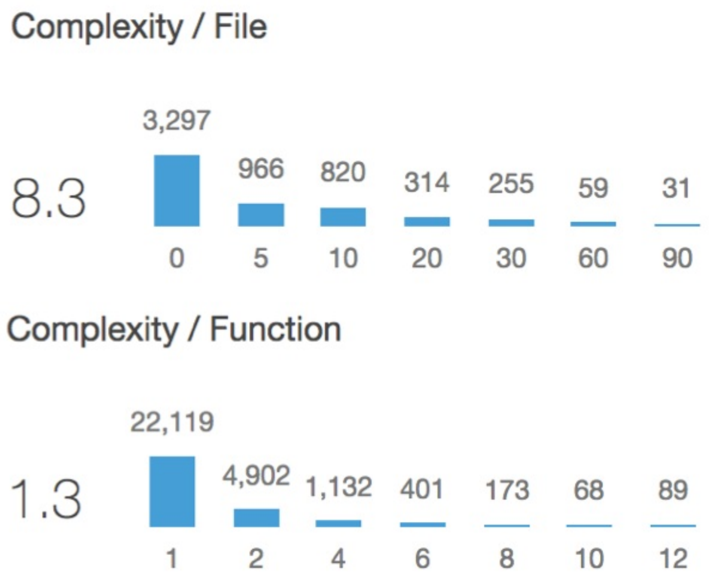


Figure 13: The Cyclomatic complexity per function and per file

Evolution of Technical Debt

Despite the extensive code review and testing at Gradle, technical debt in the project has grown over the years. This section elaborates about this debt and tries to quantify it. First, an analysis of the code base is presented, followed by the number of TODOs in code and how this has changed through time.

Code base analysis

The Gradle Build Tool started in 2009 with the release of Gradle 0.7. With only 597 files and 40.528 lines of code, the tool was not as big as it is today. The size of the Gradle source code has grown steadily to almost 400.000 lines of code in the most recent version. A visualisation of this growth can be seen in Figure 14. While the number of lines increased over the years, the percentage of comments in code has decreased slowly. From over 20% in 2009 to less than 15% in 2017, as is shown in Figure 15. This probably means that comments about possible changes needed or suggestions about new features in the code have been resolved faster than they have been introduced. The growth of Gradle can also be seen in the amount of statements per method. Method size has grown over the years from containing 2,4 statements per method to 2,8 statements per method.

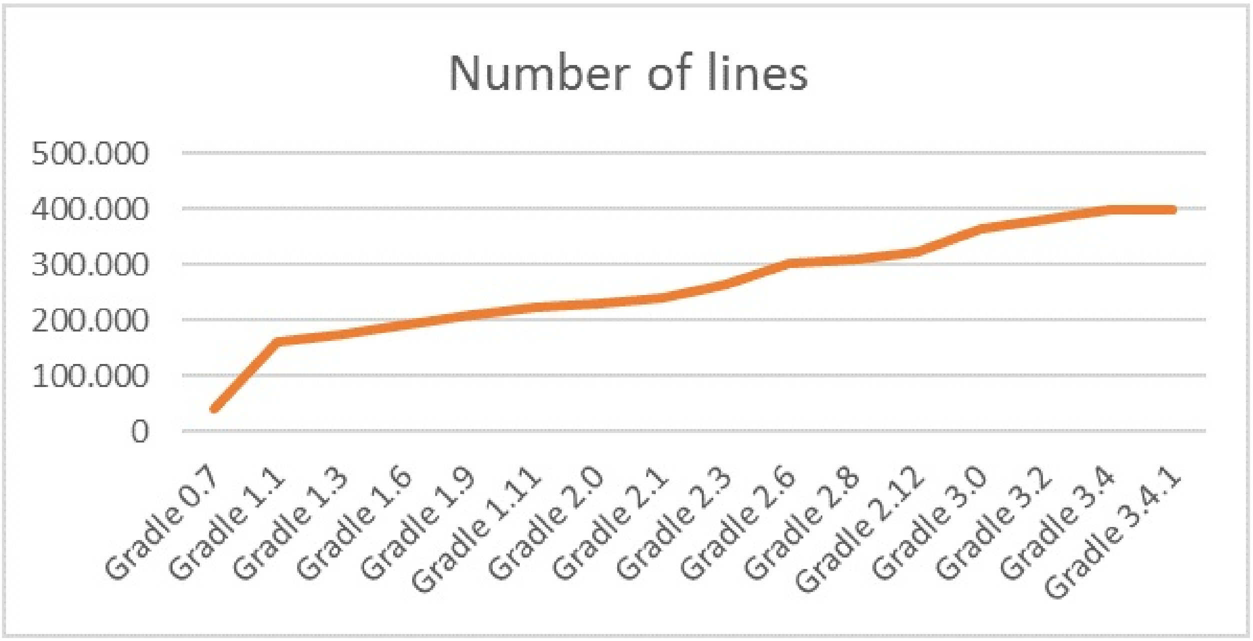


Figure 14: The number of lines for each Gradle release

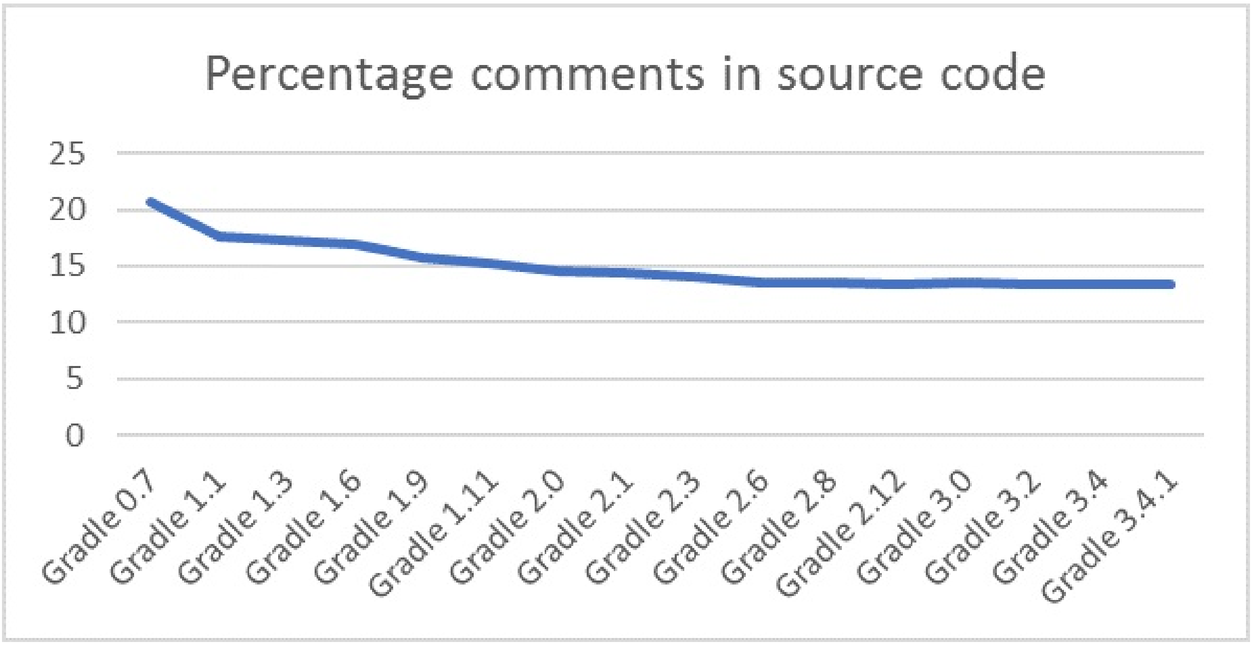


Figure 15: The percentage of comments in the source code for each Gradle release

Todos in code

The frequently used TODO annotation could hint at technical debt, indicating postponed tasks. Therefore, in order to perform further analysis on the growth of technical debt, the occurrence of the word TODO in the source code has been analysed. The results of this analysis are shown in Figure 16, showing that the development team attempts to solve as many TODO's as possible before the next major release, while on the other hand introducing additional TODOs when adding new features. Despite the major spike in TODOs due to the high occurrence rate in the user guide in the versions 2.0-2.12, the total number of TODOs is steadily rising through time at a slow pace. However, when comparing this growth rate to the increase in lines of code, this always remains a limited portion of the total code base.

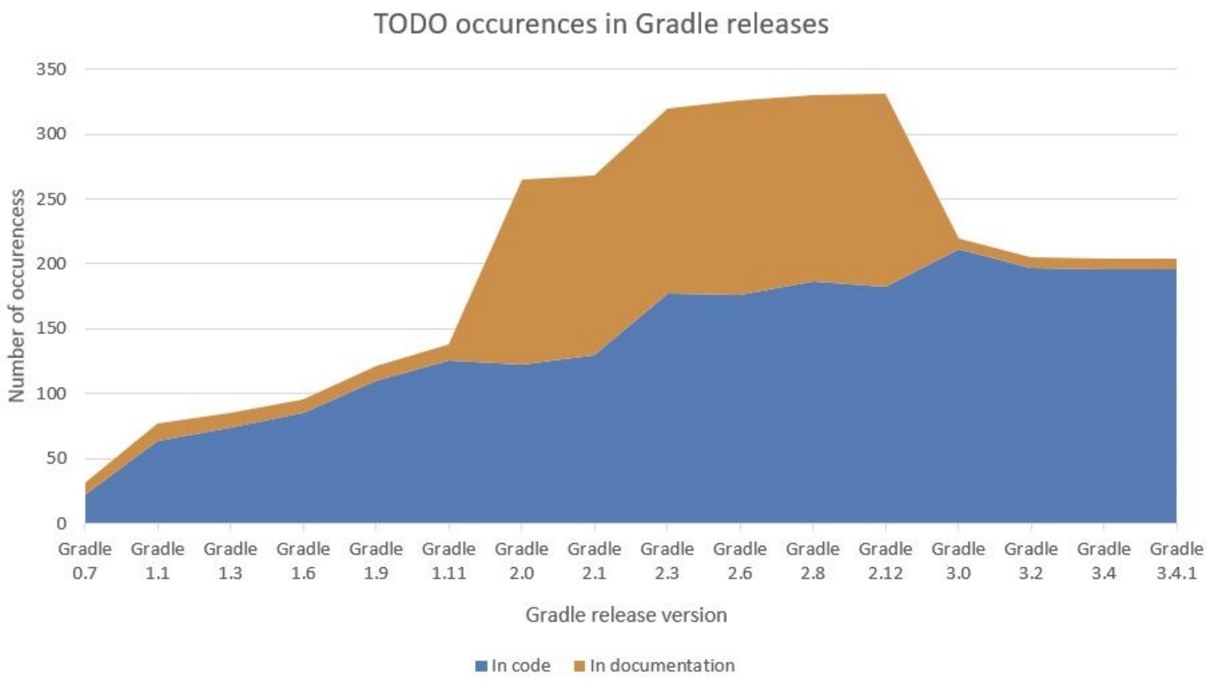


Figure 16: The amount of TODOs per Gradle release

An example of a TODO left in the source code can be found in the snippet below. The file `BuildableJavaComponent.java` is part of the Gradle API and contains some legacy code, as commented by Adam Murdoch himself two years ago. This resembles a perfect example of a postponed task that will improve the efficiency of the project when fulfilled.

```
/**
 * Meta-info about a Java component.
 *
 * TODO - this is some legacy stuff, to be merged into other component interfaces
 */
public interface BuildableJavaComponent {
    Collection<String> getRebuildTasks();

    Collection<String> getBuildTasks();

    FileCollection getRuntimeClasspath();

    Configuration getCompileDependencies();
}
```

Main findings

Overall, Gradle has proven to take code quality and technical debt very seriously and invests a lot of time and effort in building a truly long-term viable project. This is also highlighted by Hans Dockter [16] in a blogpost on the old Gradle forum regarding the release of version 1.0 stating:

> The main reason why milestone-5 took so long to be released is that we found out that a lot of stuff we have been using Ivy for needed to be fully rewritten to give it the capabilities we want. This was a huge investment and we were basically paying back a huge technical debt. We paid back most of it. This will also enable us to provide many innovations in the area of dependency management in the future.

Even though, Gradle version 3.4.1 was recently released, this philosophy is still at the core of the project. The different integrated code quality tools ensure constant code quality and reduces the number of new bugs introduced. There are some improvements to be made such as publishing their Github to Coverity, to use more generic tools that might indicate improvements based on a different vision of technical debt that these tools might maintain. Fuzz testing could also be a good addition to the test suite, allowing unexpected input values to uncover unexpected bugs.

The evolution of technical debt shows an upward trend and Gradle should not allow their growth to change their current process and leave the technical debt increase uninhibited. Gradle should maintain the awareness throughout the company in order to be successful in the long-term, technical debt should always be a priority.

Conclusion

This chapter summarised the Gradle Build Tool in many architectural views and perspectives, helping the reader to be able to understand and contribute to the project. The analysis of the Gradle project has led to the conclusion that Gradle has an interesting, yet complex architecture.

In the first section, the stakeholder analysis discussed the various classes of stakeholders involved in the project. Gradle is used by a very broad spectrum of users, but made by a relatively small group of developers at Gradle Inc., supported by contributions from the community. In general, the Gradle team aims to build anything and automate everything faster than is currently possible. This requires a highly flexible piece of software, which can be modified and expanded using a variety of plugins. The context view showed the dependencies on external libraries, tools and frameworks Gradle uses to deliver its product. After which, the development of the project is further analysed showing a very fragmented source code, providing the developers of Gradle with a flexible architecture. A relatively small core interacts with a vast number of plugins, which all have their respective test suites included. Testing is given the highest priority by the Gradle developers, as discussed in the section dedicated to the Spock framework. Literally thousands of tests are written to assess the project using TeamCity CI. In addition, every contribution needs to pass an assorted test suite named quick test, while also adding their own tests for novel code.

The Gradle Build Tool is now eight years old and the project has evolved significantly over time. Constant innovation, could lead to a growing amount of technical debt in the project. However, Gradle has proven to take code quality and technical debt very seriously by investing a large amount of time and effort in building a truly long-term viable project. The overall analysis of Gradle shows that projects like this truly benefit greatly from the open-source environment. The flexible architecture, a group of passionate developers and contributors and a focus on testing have gotten Gradle where it is today, while making it ready to evolve even further throughout the years to come.

References

1. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. Wikipedia. Gradle. https://en.wikipedia.org/wiki/Gradle, 2017.
3. Gradle.org. Gradle Build Tool. https://gradle.org/, 2017.
4. Gradle.com. About. https://gradle.com/about, 2017.
5. Business Wire. "Gradle Inc. Raises $4.2M With True Ventures And Data Collective". Businesswire.com, http://www.businesswire.com/news/home/20151210005335/en/Gradle-Raises-4.2M-True-Ventures-Data-Collective, 2017.
6. Gradle.org. Gradle User Guide. https://docs.gradle.org/3.4.1/userguide/userguide.html, 2017.
7. GitHub.com. google/osdetector-gradle-plugin. https://github.com/google/osdetector-gradle-plugin, 2017.
8. GitHub.com nebula-plugins/gradle-netflixoss-project-plugin. https://github.com/nebula-plugins/gradle-netflixoss-project-plugin, 2017.
9. GitHub.com gradle/gradle. https://github.com/gradle/gradle, 2017.
10. Gradle.org. Plugins. https://plugins.gradle.org/, 2017.
11. Gradle.org. Builds. https://builds.gradle.org/, 2017.
12. Spock. Spock Framework. http://spockframework.org/, 2017.
13. Spock. Data driven testing. http://spockframework.org/spock/docs/1.1-rc-3/data_driven_testing.html, 2017.
14. Technopedia. Technical debt. https://www.techopedia.com/definition/27913/technical-debt, 2017.
15. Codecentric. Why good metrics values do not equal good quality. https://blog.codecentric.de/en/2011/10/why-good-metrics-values-do-not-equal-good-quality/, 2011.
16. Gradle Discuss. Status of the 1.0 release. https://discuss.gradle.org/t/status-of-the-1-0-release/7735, 2011.

17. Gradle.org. Gradle User Guide: The Gradle Wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html, 2017.
18. Gradle.org. Gradle User Guide: The Gradle Daemon. https://docs.gradle.org/current/userguide/gradle_daemon.html, 2017.
19. Semver.org. Semantic Versioning 2.0.0. http://semver.org/, 2017.
20. Gradle Discuss. Gradle release distribution Link. https://discuss.gradle.org/t/gradle-release-distribution-link/21033, 2017.
21. Gradle.org. Gradle Release Notes. https://docs.gradle.org/2.7/release-notes.html, 2015.
22. Gradle.org. Gradle User Guide: Gradle Plugins. https://docs.gradle.org/current/userguide/plugins.html, 2017.
23. GitHub.com. Gradle Design Docs: QA and release automation. https://github.com/gradle/gradle/blob/master/design-docs/done/qa-and-release-automation.md, 2012.
24. Gradle.org. Gradle Release page. https://gradle.org/releases, 2017.

JabRef - A Graphical Application for Managing BibTeX Databases

Alborz Salimian Rizi, **Owen Huang**, **Rolf Starre**, and **Tim van Rossum**.

Delft University of Technology, 2017

Abstract

JabRef is an open source BibTeX reference manager, written in Java. It can be used to structure and manage large BibTeX/BibLaTeX reference databases used for writing scientific papers. The system has been in development since November 2003 and is still actively maintained. The fact that the project is under active development can be deduced from the speed at which pull requests are merged: usually within a day. As of 20-03-2017, the project has over 135.000 lines of code, 95 contributors, and 31 releases. This chapter studies JabRef by looking at its architecture, and by looking at the system through different viewpoints and perspectives.

Table of Contents

Introduction

JabRef is a graphical BibTeX reference manager, used for structuring large BibTeX databases in a orderly fashion. This chapter gives a structured overview of JabRef by providing descriptions of the various parts that compose its architecture, in order to explain the system and how it works. After introducing the system, the most prominent stakeholders and their workflow are identified. Next, the interaction of JabRef with its environment is illustrated through analysis of the context view. Furthermore, the development view is described to find code structure and dependencies. Following that, an overview of the technical debt of JabRef is given by characterizing some of its issues. After that, this chapter continues with an evolution perspective of JabRef, which shows the history of JabRef. Finally, the chapter concludes by listing the most important findings.

Stakeholders

In this section we will describe the different stakeholder classes as proposed by Rozanski and Woods [1] and relate our views of the classes in relation to the JabRef project. In addition we have also identified a stakeholder group that was not a part of the eleven types proposed by the book, the sponsors.

The **sponsors** of JabRef are Baola and Neuronade. While they support the development of JabRef, they do not play an active role in steering of the project. Currently the project relies on volunteers for the development of the project, although they are trying to attract a funded developer.

Since the core developers of JabRef determine the direction of the product we consider these the **acquirers** of the project. They share their vision and roadmap via GitHub and the JabRef website. The following persons make up the core development team of JabRef: Stefan Kolb, Oliver Kopp, Tobias Diez, Matthias Geiger, Jörg Lenhard, Simon Harrer, Oscar Gustafsson, and Christoph Schwentker.

We also consider these core developers a part of the **developers**, **maintainers**, and **production engineers** stakeholder groups. They actively contribute to the development of the project and are the ones who are responsible for reviewing and merging pull requests. This means they determine which contributions get merged and which get rejected. They are also the ones who manage and deploy the different aspects of building, testing, and running the system.

Since JabRef is an open source project and contributions from outside are actively encouraged, there are also a lot of people outside of the core development team who have made contributions to the JabRef code base. These contributors are part of the **developers** class, since they make contributions to the development of JabRef through pull requests. Contributors are also considered to be part of the **maintainers**, since they provide assistance in maintaining the system by making pull requests.

Both the core developers and contributors make up the **assessors** and **testers** groups. When someone offers to contribute to the project, they are asked to run a set of unit tests to make sure any additions do not cause problems in the system. Furthermore, the assessors and testers also request that the contributor creates additional unit tests when implementing new features. All the developers are a part of the assessors because they are responsible for conforming to certain standards and legal regulations, since the contributions are considered to be made under the MIT license.

There is a diverse group of **communicators** and **support staff**. First off we have the creators of the JabRef help website, mainly the core developers but also other contributors have worked on this. The help page contains most of the information needed for setting up and using JabRef. There is also a forum on the JabRef website where users and other people can ask questions. The communicators on this forum with the most replies are Tobias Diez, Matthias Geiger, and Christoph Schwentker, who are all part of the core developers. The user mlep also deserves a mention since he is also actively involved on the forum and is one of the moderators. Contributors and the core developers communicate through GitHub by addressing issues and discussing code. Since 2016 JabRef started organizing a yearly conference, [JabCon]http://jabcon.jabref.org/, for users of JabRef to facilitate discussion between users and developers. This year the JabCon was organized by Stefan Kolb, Oliver Kopp, and Jörg Lenhard.

Suppliers that JabRef depends on are GitHub, the different databases it requests data from (e.g. Google Scholar, Springer, ACM Digital Library), BibTeX, and BibLaTeX. Github is important because it is where the code is maintained and developed. The various databases are used to extract information from to create a bibliography, which can then be exported using BibTeX or BibLaTeX.

Since JabRef is released under the MIT license the **users** of the system are hard to pin down, since anyone could download JabRef and make use of it. Since JabRef is a system for managing references we expect the main part of the users to consist of students and universities. Since the users download the system and then run the system locally we consider the users to also be the **system adminstrators**.

Context view

In this section we define the context in which JabRef operates and the interactions of JabRef with its environment. First we give a brief overview of the system scope and responsibilities. Afterwards we briefly introduce the different entities with which JabRef interacts.

System scope

The main function of JabRef is to allow users to create, maintain, and export bibliographies. In short, the most important responsibilities and capabilities of the system are:

- Empowering the users to construct bibliographies by giving users the possibility to retrieve (information about) articles and papers from large databases.
- Enabling users to add, edit, and delete entries in their bibliographies.
- Allowing the users to export the bibliographies to BibTeX and BibLaTeX files.
- Supporting multiple languages in order to aid non-English speaking users.
- Supporting multiple platforms (e.g. Linux, Windows, and Mac).

Context view diagram

The diagram below in Figure 1 shows the context view of JabRef. The context view diagram shows the external entities, as well as the most important stakeholders. In this section we will provide some short descriptions for some of the entities that have not been mentioned before or that require some additional explanation.

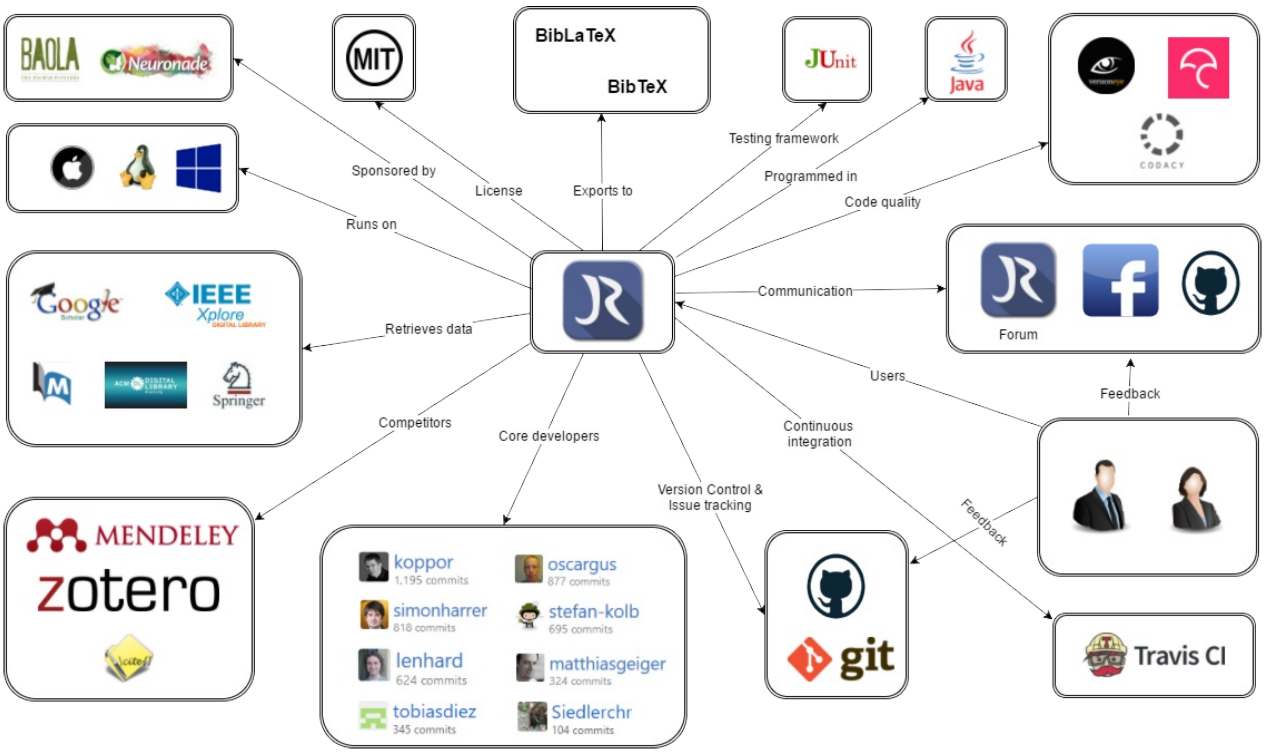


Figure 1: Context view of JabRef.

To help developing and maintaining the code JabRef uses a number of tools, these fall under the *code quality*. Codacy is used for automatic checking of code quality, by providing insight into code style, duplication, and code complexity. Codecov helps the developers to automatically check the code coverage after implementing changes and to expose bugs and vulnerabilities. VersionEye is used to prevent having outdated dependencies and license violations. For checking if new contributions do not cause any problems JabRef uses Travis CI for *continuous integration*. Furthermore, since JabRef is *programmed in* Java it uses JUnit as a *testing framework*.

To create bibliographies JabRef allows users to *retrieve data* from databases, such as Google Scholar, ACM Digital Library, etc., to create references. These bibliographies can then be *exported to* BibTeX or BibLaTeX files.

JabRef also has some *competitors* that provide similar services. Some popular alternative bibliography managers are Mendeley, Zotero, and BibDesk.

Development view

In this section, we will look into the architectural principles, patterns, and code guidelines that govern the development of JabRef. Following the definition of Rozanski and Woods [1], we show how JabRef is structured and what standards the development process upholds.

Module Organization

Prior to 6 September 2016, JabRef had reportedly a lack of clear general architecture [2]. Currently, the overall system follows a modular structure splitting the system into three major components. The architecture shows signs of a Model View Controller (MVC) inspired design, with additional smaller components hooked into an event bus. Minor components, that are not directly at the core of the application, are subscribed to this event bus. This allows them to still act upon changes of the major components without having to resort to changing the core architecture. Furthermore, these minor components can in turn influence the core components. Generally, in a MVC design the system is split into three major components: the model, view, and controller. The model contains all the data structures and is commanded by the controller. Similarly, the view is also updated by the model through notifications on changes in the model and also exerts influence on the controller [3]. However, there are several dependencies allowed between some components in JabRef indicating clear distinctions from a pure MVC design. We note this difference in the following discussion of the major components. The overall modular structure of JabRef is depicted in Figure 2, which shows the major interactions of the system and dependencies of the entire architecture.

Core components:

- *UI:* The user interface (UI) is split into two major parts: the command line interface (`CLI`) and the graphical user interface (`GUI`). The `UI` is initialized by the `Launcher` , which launches the entire application. The `GUI` can communicate directly with all the other core components, which include the `Logic` and the `Model` components. As a result, this differs from an MVC pattern, because the `GUI` is able to directly manipulate the `Model` bypassing the `Logic` component. Moreover, the `UI` component can interact with the `Preferences` containing all the application preferences of the user.
- *Logic:* The `Logic` unit is responsible for updating the model, which includes reading, writing, importing, exporting, and general manipulation of the model. The `Logic` is also responsible for logging errors and warnings that occur during run-time. Inside the `Logic` component the most important sub-components include:
    - Logging to register and report all warnings and errors.
    - Handling importing of files (such as parsing and creating the respective data structures for storing in the `Model`).
    - Fetching entries from varying identifiers (DOI, ISBN), via web search, etc.
    - Formatting of the data obtained from various sources to be displayed in the UI.
- *Model:* The `Model` contains the most important data structures of the system. Among these components, the `EventBus` is one of the most important ones, as it allows modules outside the core of the system to interact upon changes happening inside the core. Moreover, the `Model` holds the data structure that stores the bibliography databases (containing entries, categorization of entries, user comments, etc.) the user in the end creates. The `Model` component only interacts with the `Logic` component and does not depend on any other module. Upon any change occurring in the sub-modules, the `Model` (and also the `UI`) will be updated accordingly. However, the logic behind several sub-components of the `Model` (`BibDatabases` , `BibEntries` , `Events` , and related aspects) is not always 'absolute'. This means that small parts related to the logic of these sub-components are found inside the `Model` , rather than in the `Logic` component. As a result, this clearly differs from the general MVC design.

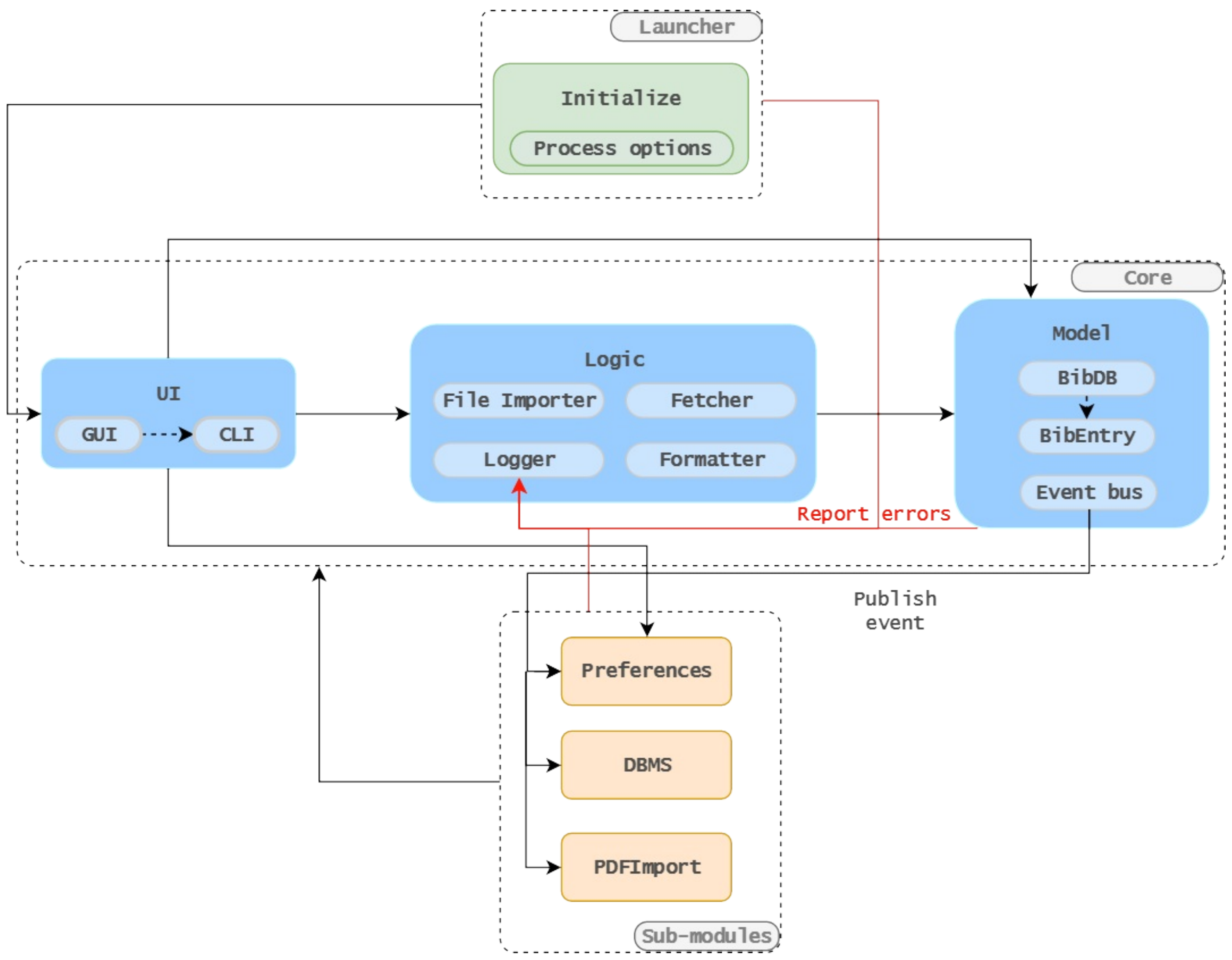


Figure 2: Modular structure of JabRef with allowed dependencies between components.

Standardization of Design

The standard of design can be split into two parts: the code style and the dependency maintenance.

The entirety of the code style is documented in the following JabRef wiki guide. Developers wishing to make contributions to JabRef should adhere to the code style described. That way a consistent style of code is guaranteed, which prevents the code from turning into spaghetti code. A short summary of the code style guidelines is as follows:

- Code should be written upholding general coding standards found in this manual.
- Exceptions should be wrapped and re-thrown with a localized message, so that users can read what the error was.
- Users should also never see the technical details, as most would not be able to make out what it is about anyway.
- Logging happens by using Apache Commons Logging.
- All labels, descriptions, and messages visible for the users should be localized.
- `GUI` confirmation dialogs should avoid asking questions, be concise, identify the item at risk, and have named buttons for the actions.

Furthermore, to improve the process of integrating changes, an automated checklist for each PR is used to verify that sufficient testing and appropriate changes have been performed. In this checklist (e.g. in PR #2614), the contributor can mark for each checklist item (e.g. including screenshots when making some 'large' UI change) whether it has been performed or leave it unchecked if it does not directly apply to the contribution.

JabRef, like most big projects, uses a lot of external libraries for its functionality. In order for developers to get these dependencies, without having to look for each and every single one of them on the Internet, JabRef uses the Gradle build tool to automate getting the dependencies from the Internet. When developers add a library, because it's needed in order to develop a feature or to fix a bug, they should preferably use a version available at jCenter and add that version to the build script, `build.gradle`.

Standardization of testing

Standardization of testing involves two components: the testing itself and the continuous integration. Testing happens exclusively through usage of JUnit 4, a unit testing framework for Java unit tests. Code coverage by the tests is measured using Codacy, a tool that does not only measure code coverage, but also gives useful code review feedback such as security concerns for certain lines of code, code style violations, best practices, etc.

Continuous integration (CI) is a software engineering practice where developers integrate their code frequently, which results in the benefit that errors are detected quickly. CI happens through the use of both Travis CI and CircleCI. Travis CI is used to execute all the tests in the system, while CircleCI creates binaries, using Gradle and Install4j, and uploads them as builds [4].

Technical debt

Technical debt is defined as "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution" [4]. If technical debt is left unaddressed, then it could cause problems in the future, possibly resulting in software refactoring being necessary.

Begin 2016, JabRef faced a few big issues [5] classified under one of these categories:

- Huge segments of the code are untested, undocumented, and have significant bugs and problems.
- In JabRef's lifespan, lots of features, UI elements, and preferences have been added, but are wired loosely in the UI. Moreover, the UI lacks consistency and structure.

Identification of technical debt

Identifying technical debt was done by applying the code analysis tool SonarQube and by manual inspection. The entire project of JabRef was analysed in the process by SonarQube.

Shockingly, it becomes apparent that JabRef has 147 days of technical debt. This amount of days is a metric that roughly indicates how much time it would cost to fix all the code that does not comply with standards that SonarQube defines [6]. These standards are roughly defined as a weighted sum of technical debt days, based on the severity level and complexity of the issue. However, it should also be noted that JabRef still has the highest possible maintainability rating (i.e. an A rating, indicating less than 5% technical debt ratio, with JabRef only having 1.7%). This technical debt ratio is the ratio between the current amount of technical debt and the amount of time it would cost to rewrite the code from scratch, which is estimated based on either the amount of lines of code or the complexity of the project.

Tools like SonarQube are almost never perfect in classifying whether or not a case of technical debt is preventable or simply because there was no better option found. As a result, manual analysis has to be performed in order to distinguish these cases a lot better.

To find the cases in which no better option was found, the rules for classifying code containing technical debt should be carefully considered. These rules are the default rules defined by SonarQube, which can be found inside the dashboard. Some of these rules that could be problematic are as follows:

Classes should have no more than five parents in the class hierarchy
Deep class hierarchies may lead to unstable code, therefore SonarQube raises alarm when the inheritance depth is more than five. However, this is problematic when one considers extending classes that are part of the Swing API. All classes that are nested deeper than five levels are extending the Swing API for added functionality and these API classes are nested deep themselves. If/when JabRef is ported to JavaFX, this technical debt could be paid simply by virtue of JavaFX possibly having its GUI-related classes less deep in the class hierarchy.

Control statements should be nested no more than three times
Because of the size of the project, there are many use cases where things can go wrong, resulting in exceptions that have to be caught causing more control statement nesting. An example is found in `org.jabref.collab.FileUpdateMonitor` . This class is a thread that needs to run forever, thus it introduces a `while(true)` control statement in the `run` -method, which continuously checks all entries using a `for` -loop, which checks if an entry has been updated using `if` , but checking this can throw an exception, which is caught using a `try` -statement. As can be seen, there are four layers of control statements, which are all essential and cannot be removed.

The cyclomatic complexity of a single method should not exceed 10
Complex code is hard to maintain and understand, thus SonarQube sets a limit of 10 on the cyclomatic complexity of any function. A shortcoming of this rule is that excessive cyclomatic complexity cannot always be avoided. For example: the `transformSpecialCharacter` method found in `org.jabref.logic.layout.format.RTFChars.java` has a cyclomatic complexity of 148, but upon closer inspection, this is necessary due to the overwhelming amount of cases possible. It is most likely possible to introduce a new method for every ten cases to eventually get code that is compliant with the "max. complexity of ten" rule, but this also leads to spaghetti-esque code, which is also bad code practice.

Testing debt

Part of the technical debt of a system is the debt that manifests itself when code is not tested after it is written. In order to find this testing debt, one can either use tools or perform manual inspection on the code.

As part of the quality control tools, JabRef uses CodeCov to generate reports about the amount of code covered by Java unit tests. The report generated by CodeCov for JabRef shows a visualization in the form of a 'sunburst', of which several are shown in Figure 3. These sunburst diagrams show all the different packages found in the project with respect to their package structure. In the center ring, the top level package is shown, and each consecutive ring around the previous ring shows an additional deeper package level. The report shows that the overall code coverage in the project is around 30%. This low percentage of code coverage is also apparent in the sunburst, by the dominating amount of red circles. However, there are also some green parts visible indicating components that are tested sufficiently with unit tests.

To understand why the code is tested so badly, the sunburst can show what the code coverage is in specific packages. Using the previous analysis from the development view on the major components of the system, we investigate the major components of the system in more detail: the `GUI` , `Logic` , and `Model` . In Figure 3, we zoom into the GUI package and see that the whole sunburst is almost red. Similarly, we zoom into the Logic and Model package as well. We observe that these classes contain a lot more green areas, indicating that a lot more unit testing is done in these other two major components.

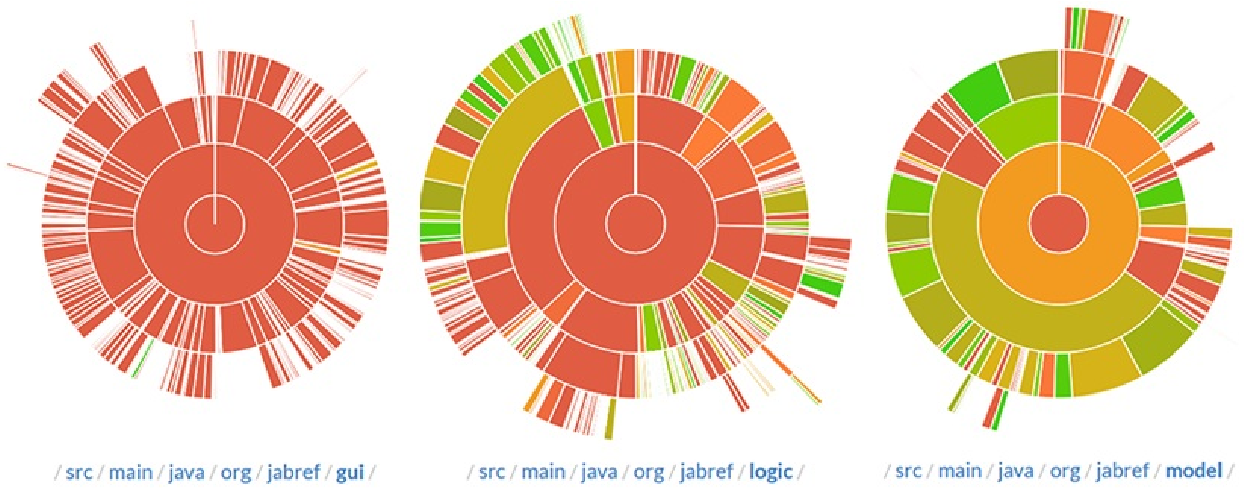
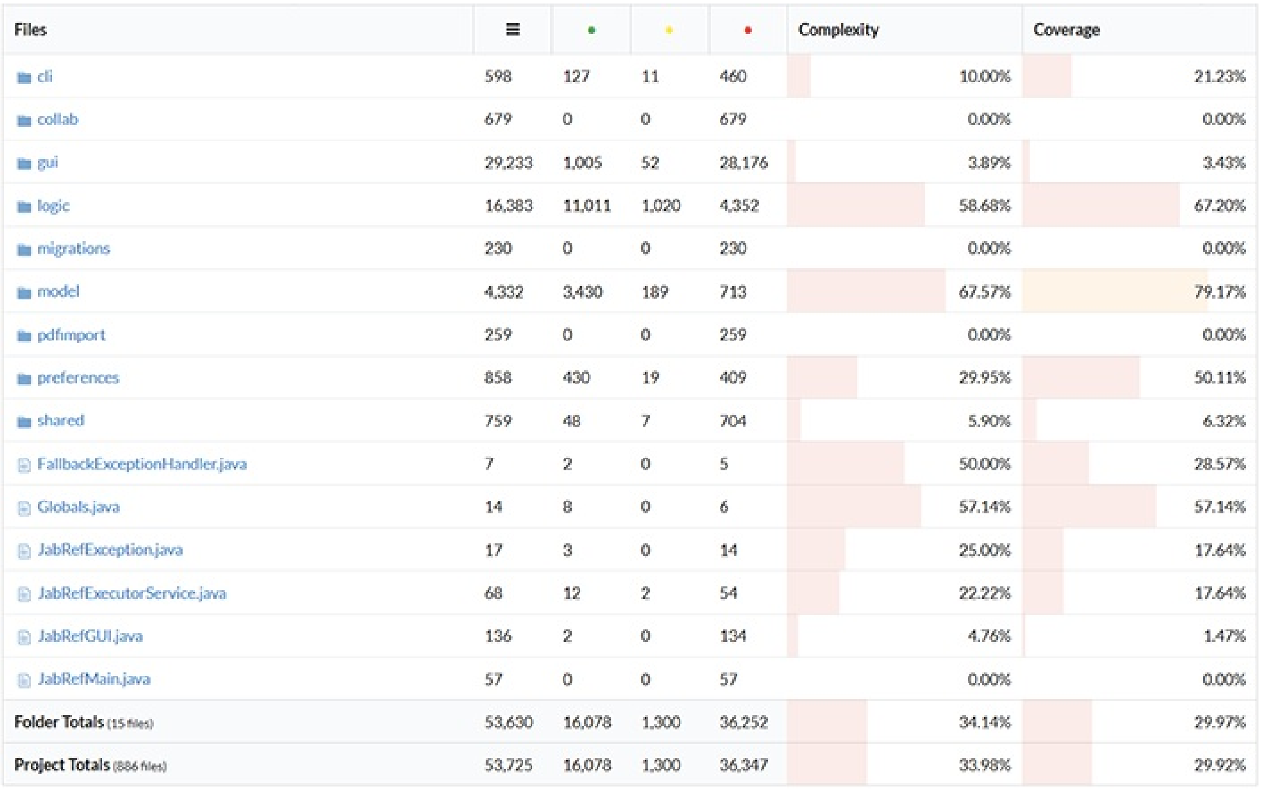


Figure 3: Sunbursts of the GUI, Logic, and Model package. Red indicates code covered for less than 70%, whereas code covered between 70% and 100% is a gradient from orange to green.

To see how much the GUI would affect the code coverage, we investigate how large the GUI component is compared to other parts of the system. In Figure 4, we observe that the GUI consists of 29,233 lines out of the total of 53,630 that are considered for the code coverage. This is already more than half of the entire codebase. Moreover, we observe that only 3.43% of the GUI was tested, explaining why the code coverage is reported only around 30%.

While the GUI is tested very badly, the other two major components have reasonable test coverage. The logic component, achieves a code coverage of 67.20%. The model component has a much higher test coverage of 79.17%.

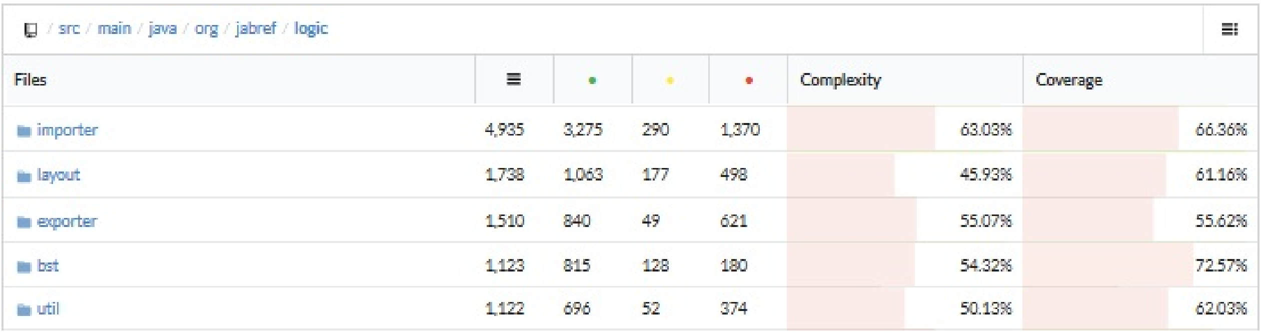


Files	≡	●	●	●	Complexity	Coverage
cli	598	127	11	460	10.00%	21.23%
collab	679	0	0	679	0.00%	0.00%
gui	29,233	1,005	52	28,176	3.89%	3.43%
logic	16,383	11,011	1,020	4,352	58.68%	67.20%
migrations	230	0	0	230	0.00%	0.00%
model	4,332	3,430	189	713	67.57%	79.17%
pdfimport	259	0	0	259	0.00%	0.00%
preferences	858	430	19	409	29.95%	50.11%
shared	759	48	7	704	5.90%	6.32%
FallbackExceptionHandler.java	7	2	0	5	50.00%	28.57%
Globals.java	14	8	0	6	57.14%	57.14%
JabRefException.java	17	3	0	14	25.00%	17.64%
JabRefExecutorService.java	68	12	2	54	22.22%	17.64%
JabRefGUI.java	136	2	0	134	4.76%	1.47%
JabRefMain.java	57	0	0	57	0.00%	0.00%
Folder Totals (15 files)	53,630	16,078	1,300	36,252	34.14%	29.97%
Project Totals (886 files)	53,725	16,078	1,300	36,347	33.98%	29.92%

Figure 4: Distribution of the size of all system's components with code coverage listed. The second column shows the total amount of code lines, the third shows the amount of lines covered, the fourth the amount of partial lines covered, and the fifth the amount of lines missed.

Naturally, Figures 3 and 4 lead to the question why it is the case that the GUI component is hardly tested. By investigating previous discussion in the issues and PRs on GitHub, @koppor stated in PR #1700 dating back to 09-08-2016 that testing the GUI is difficult and would be left alone, partly as a result of using Java Swing. He also mentions in the same comment that in order to test the GUI, JabRef would have to switch to JavaFX.

We have previously mentioned that the logic component achieved a code coverage of 67.20%. The entire logic component consists of 16,383 lines of code. While this percentage of code coverage seems acceptable, it calls for the need to investigate why this is not more. Looking into an overview of the logic component in Figure 5, we see that tests are generally covering most large packages around a similar code coverage percentage. To see why the largest package, `importer`, only achieves a 66.36% code coverage, we can continue going into the deeper layers of packages.



/ src / main / java / org / jabref / logic

Files	≡	🟢	🟡	🔴	Complexity	Coverage
importer	4,935	3,275	290	1,370	63.03%	66.36%
layout	1,738	1,063	177	498	45.93%	61.16%
exporter	1,510	840	49	621	55.07%	55.62%
bst	1,123	815	128	180	54.32%	72.57%
util	1,122	696	52	374	50.13%	62.03%

Figure 5: Top five largest packages of the code lines' distribution in the logic package.

Inside `importer` it becomes clear that the largest package, `fileformat`, is thoroughly tested. However, the second largest package, `fetcher` is almost completely untested. If we look into the amount of issues and PRs tagged with a `testing` and `fetcher` label, only two issues regarding this are found. Furthermore, no PRs are labelled with these two labels. However, through manual inspection of the source code there are tests for the fetcher to be found inside their respective package for `importer.fetcher` tests. However, running the code coverage with EclEmma (recommended by the development guide) reports that classes inside the fetcher, such as `GoogleScholar.java` has a coverage of over 70% while this CodeCov reports this as 0%. This means that CodeCov may skew the image of having a lot of untested code, even though this is not the case. We can see this very clearly by comparing the EclEmma code coverages reported on the `importer.fetcher` package by CodeCov and EclEmma in Table 1. In EclEmma the code coverage of the entire package is 86.2%, while in CodeCov only a small 5%.

Table 1: Reported line coverage of the top 5 largest files in the `fetcher` package by both CodeCov and EclEmma.

Files	CodeCov coverage	EclEmma coverage
ArXiv.java	3.64%	89.5%
MedlineFetcher.java	3.84%	85.4%
GoogleScholar.java	0.00%	83.8%
AstrophysicsDataSystem.java	12.69%	87.3%
MrDLibFetcher.java	0.00%	73.5%
Total package coverage	**5.39%**	**86.9%**

Having discovered the inconsistency in reporting of code coverage depending on tools, it calls for the need to compare the total code coverage reported by EclEmma and CodeCov together. We run EclEmma over the entire project and see that EclEmma reports a much higher code coverage than CodeCov! The reported code coverage is 42.9% on the main part of the program. This is already 12% higher than what CodeCov reported.

Deployment view

The deployment view looks at parts of the system that are relevant once it has been built [1]. It defines physical, computational, and software-based requirements for running the system. A diagram of the deployment view is given in Figure 6.

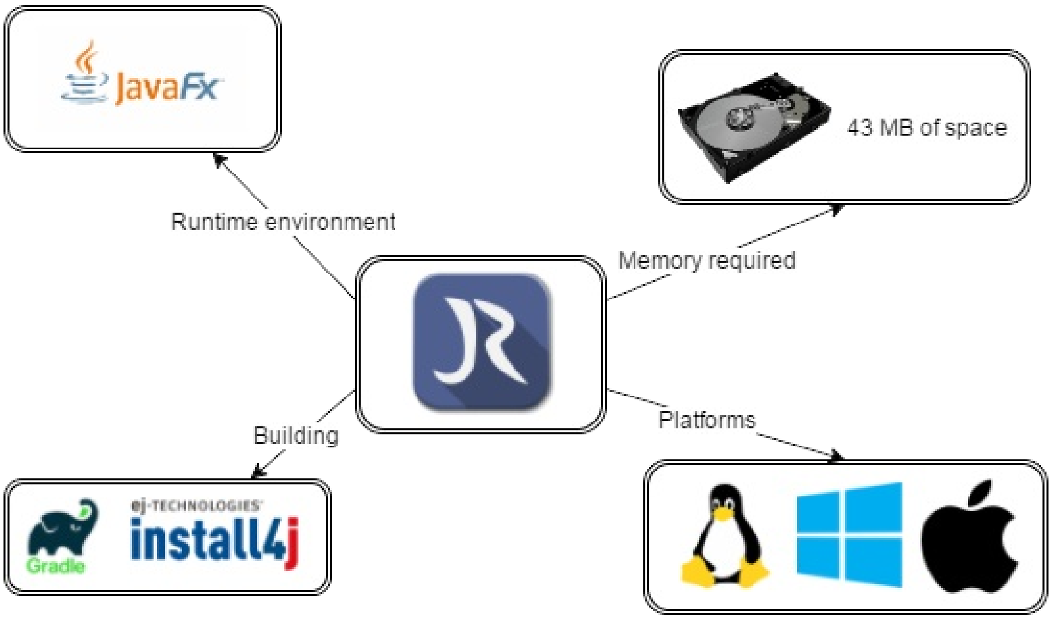


Figure 6: A graphical overview of the deployment view. All individual parts will be explained in the deployment view.

First of all, JabRef is a Java program, and thus requires Java to be installed in order to be able to run. More specifically, because of its JavaFX dependencies, running JabRef requires Java 8 (version 60 or newer). The Windows installer will actually install Java for you if no compatible Java installation is found [7]. Because JabRef uses Java 8, it can only be run on technology that can support Java 8. Moreover, Java 8 can be run on a variety of operating systems [8]. Also, in order to find papers on the Internet through JabRef, JabRef needs a working internet connection.

JabRef requires Gradle to be built and Install4j to be installed. Gradle also handles collecting all the external Java dependencies when building JabRef from source. Getting the dependencies by looking for them on Google is not only tedious, but some up-to-date versions of the dependencies cannot be found while looking on the internet. Install4j uses a executable `.jar` file and creates a platform-dependent installer for the executable.

Aside from the memory required to install JabRef in the first place, no other hardware requirements for using JabRef are mentioned anywhere. JabRef requires around 43 Mb of storage space to be installed.

Evolution of JabRef

JabRef has changed significantly since its initial release. The project started as "just" a BibTeX/BibLaTeX reader and nowadays it is able to do much more. All notable changes to the project are documented in the changelog.

JabRef 1.0 was developed to replace Bibkeeper and JBibtexManager, and to fuse their functionalities into one project. In between JabRef 1.0 and JabRef 2.0, several releases have been made, which were either to fix bugs or to add new features. One noteworthy release is the 1.3 release, made on 9-5-2004. This release contained a bug which rendered some features unavailable, so the download link for the release was removed the next day. Two days later, JabRef 1.3.1 was released which fixed the bug.

Two years and two months after the release of JabRef 1.0, JabRef 2.0 was released on 30-01-2006. This release added many features, such as new import/export filters and handling of journal name abbreviation and unabbreviation. Just like JabRef 1.3, JabRef 2.0 contained a serious bug that could corrupt data. This prompted the JabRef team to release JabRef 2.0.1 only a few days later which fixed the bug. It is rather hard to find out whether or not the added features for every new version were added per request of the community or added because the developers thought they were necessary, as the issues on GitHub only date back to January 2016, and JabRef 3.0 was released on 29-11-2015.

For almost ten years, people used versions of JabRef 2.x until JabRef 3.0 was released. JabRef 3.0 contains a lot of extra functionality, such as: support for OpenOffice 4, search from Springer, and more. The main change between JabRef 2.11.1 (the last version of JabRef 2.x to be released) and JabRef 3.0 is the usage of Java 8 features, moving the project towards making use of Java 8. Because the current changelog contains a lot of entries which are fixes for certain issues and feature requests, it is likely that most of the added features were implemented on request of the community. Another big change, beginning just before the release of JabRef 3.0, is the migration of JabRef from Sourceforge to GitHub. Issue #111 indicates that the first steps towards migrating the project were made around three months before the JabRef 3.0 release, and it was officially considered done on 16-02-2017, the date that the issue was closed.

Nowadays, the most recent stable release version of JabRef (as of 2-4-2017) is version 3.8.2, and the lead developers are making plans for releasing JabRef 4.0, which would move JabRef from Swing to JavaFX. This is necessary, considering the fact that JavaFX is intended to replace Swing as a GUI library. It would also remove a lot of testing debt, as GUI-related classes can be tested a lot easier when written using JavaFX. The evolution of JabRef is visualized in a short video.

Conclusion

JabRef is a tool that started out as a manager of BibTeX and BibLaTeX reference files, and nowadays it can do a lot more than that, e.g. looking up papers on the Internet. In this chapter, we analyzed the project from different views and perspectives. Each of these gave more insight into the architecture of JabRef.

First off, in the stakeholder analysis, we discovered that JabRef is actively maintained and developed by a small team of developers (the JabRef team) and external contributors. Aside from that, the other stakeholders were clearly defined according to the stakeholder classes in Rozanski and Woods [1].

After that, in the context view, the system scope of JabRef is given. We explained what tools JabRef uses to help maintaining and developing code, as well as give an overview of all the interactions between JabRef and its environment.

Next, in the development view, the principles that guide the development of JabRef are given. This part explained the core components of JabRef, and we discovered that the architecture of JabRef drew inspiration from an MVC design pattern. The development view also explained some of the testing and design standards.

Afterwards, in the technical debt part, we discussed the technical debt that JabRef is facing. JabRef faces a lot of technical debt, but that is mainly because it is such a big project. Also, some parts of JabRef are barely tested, but this is because these parts are GUI-related. It is very hard to test these right now as JabRef still uses the Swing API and not JavaFX. We also discovered that CodeCov gave a skewed image of the amount of tested code compared to EclEmma.

Furthermore, in the deployment view, the main components necessary for launching JabRef were given. There was not a lot to discuss here, as JabRef is a lightweight Java program without specific hardware requirements beyond 43 MB of memory available on an HDD or SSD.

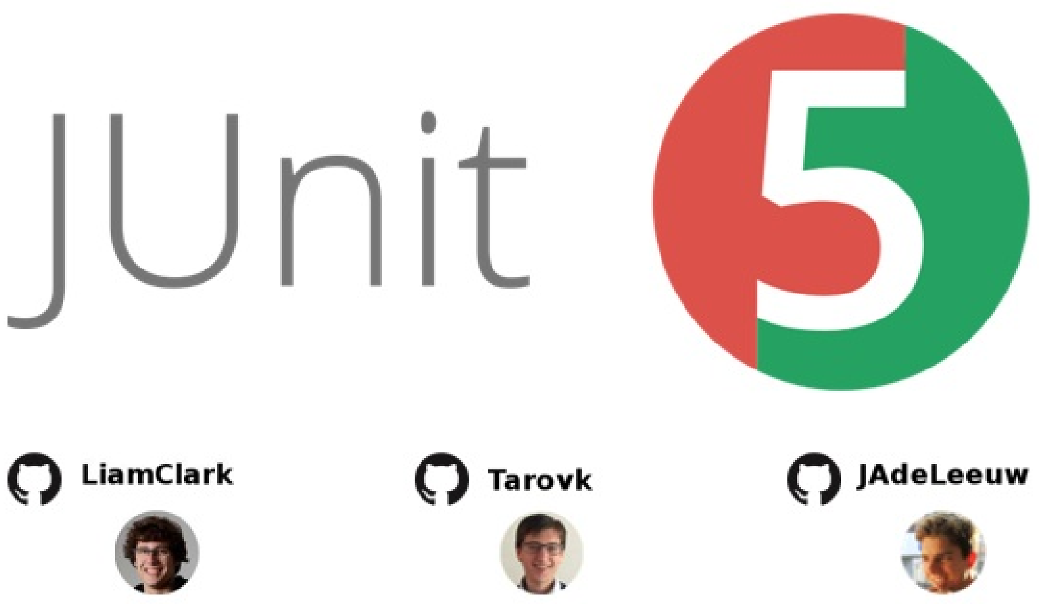
Finally, in the evolution perspective, we discussed the evolution of JabRef from its first launch in late 2003 to the version we use nowadays. We also noted the migration from hosting the project on Sourceforge to GitHub. This started right before the JabRef 3.0 release and ended mid-February this year, just before we started to analyze JabRef.

JabRef is a powerful tool nowadays, and definitely has potential to be used widely in the future.

References

1. Rozanski, N. Woods, E. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. Harrer, S. Kolb, S. Kopp, O. Lenhard, J. High Level Documentation https://github.com/JabRef/jabref/wiki/High-Level-Documentation, Wiki page, submitted 06-07-16, consulted on 06-03-17.
3. Harrer, S. Kolb, S. Kopp, O. CI https://github.com/JabRef/jabref/wiki/CI, Wiki page, submitted 14-02-17, consulted on 06-03-17.
4. Unknown author, Technical Debt, date of publication unknown, consulted on 12-3-2017, https://www.techopedia.com/definition/27913/technical-debt
5. Oliver Kopp, Development strategy, published on 22-12-2016, consulted on 12-3-2017, https://github.com/JabRef/jabref/wiki/Development-Strategy
6. Ann Campbell, Technical Debt, published 11-1-2016, consulted on 10-3-2017, https://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt
7. The JabRef team, Installation, date of publication unknown, consulted on 1-4-2017, http://help.jabref.org/en/Installation
8. Oracle, Oracle JDK 8 and JRE 8 Certified System Configurations Contents, date of publication unknown, consulted on 1-4-2017, http://www.oracle.com/technetwork/java/javase/certconfig-2095354.html

JUnit 5: the next generation of testing for the JVM



By Liam Clark, Thomas Overklift and Jean de Leeuw.

Abstract

JUnit 5 is the successor of JUnit 4, which is the largest Java testing framework and third most imported Java package currently in existence. The vision of JUnit 5 is to provide a versatile testing framework which is not tightly coupled towards several stakeholders like JUnit 4 is.

This chapter provides different views into the project, creating an overview of the project. These different perspectives range from an architectural perspective to a developers perspective and more. Furthermore, the chapter also describes the internal workings of the system and highlights the differences between JUnit 5 and its predecessor, JUnit 4. We conclude the chapter by discussing the success of the vision of JUnit 5.

Table of contents

Introduction

Unit testing Java code is traditionally done with JUnit. The fourth iteration of JUnit, JUnit 4, is at the time of writing the largest and most commonly used Java testing framework. Unfortunately, JUnit 4 suffers from architectural problems, hampering the development of JUnit 4.

The JUnit team wants to create a solid foundation for the future iterations of JUnit, extending past JUnit 5, called `junit-platform`. The foundation should solve some of the architectural problems of JUnit 4 and ease development of future JUnit versions that can be built on top of the foundation.
The first new version JUnit team wants to create is JUnit 5, a.k.a. `junit-jupiter`, containing new features while maintaining most of the core features of JUnit 4.
JUnit 4 as a whole will still be supported through `junit-vintage`, a version that is put on top of the foundation just like JUnit 5. This structure with multiple versions eases the transition from one version of JUnit to another. The foundation combined with different (new) iterations of JUnit forms the vision of the JUnit team: a framework that is versatile, able to evolve over time, and loosely coupled towards external stakeholders. This is what the architects of JUnit ultimately strive to accomplish.

At the time of writing, JUnit 5 is still in development, with the first official release planned on the 24th of August [32]. Even though it is still in development, JUnit 5 is usable and already covers a lot of use cases.

This chapter provides insight into the JUnit 5 project, giving the readers an understanding of the project as a whole. The chapter starts by covering the different stakeholders and explaining their perspectives. Then, it will put JUnit 5 into context, describing the different relationships JUnit 5 has with other projects. After that, the chapter will explain the structure of the system. At the end of the chapter the differences between JUnit 4 and JUnit 5 will be explained, and the reasons for the creation of JUnit 5. The chapter concludes with our personal opinion on the JUnit 5 project as a whole, and if, according to us, they succeeded in their mission.

Stakeholders

Using the stakeholder analysis categories in the book *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* we have identified several groups of stakeholders in the JUnit 5 system [1]. In the sections below we will identify the stakeholders by category.

Acquirers

Acquirers can be seen as business sponsors and are in of JUnit 5's case mainly external investors. The development of JUnit 5 has started with the launch of a crowd funding campaign on Indiegogo [2]. Therefore all of the companies or individuals that funded the campaign can (to a certain extent) be marked as acquirers. We would like to name American Express as the main sponsor and Pivotal as the main contributor of the Indiegogo campaign.

Assessors

Assessors are stakeholders that oversee whether legal constraints are complied with. We did identify Indiegogo as an assessor because of the crowdfunding campaign. In the campaign JUnit states a certain vision and several features they want to include in the new version of JUnit. By using Indiegogo in this manner they are required to actually implement what they promised.
Another legal constraint that one could think of are the licences that the dependencies of JUnit have. The four dependencies of JUnit (in production), Java [3], Surefire [4], OTA4J [33] and Gradle [5], luckily have licences that allow free use and distribution of software. Finally, there are the licenses [6] that JUnit itself uses. These licenses have an effect on the users of JUnit and potentially limit certain usages of the framework.

Contributors / Developers

In the case of JUnit 5 contributors are more than just developers. Contributors are also testers and maintainers. When contributing to JUnit a developer has to test his or her own contribution before it is merged into the code base. A contribution can be the implementation of a new feature, but it can also be a bug fix, an update to the documentation or a refactor. Therefore it differs per contribution what role a contributor has.
JUnit has many minor and major contributors but we'd like to identify the five people who have taken on the lion's share of the development of JUnit 5:

- @sbrannen
- @marcphillipp
- @jlink
- @mmerdes
- @bechte

Suppliers

We have chosen to divide the suppliers of JUnit 5 into two different subcategories: the first category consists of the suppliers that are used for the development of JUnit 5; the second category consists of suppliers that support the end use of JUnit 5 in some way.

- *Development suppliers*
    - Java 1.8
    - Clover
    - Gradle
    - Jenkins / Travis CI / AppVeyor
- *End use suppliers*
    - IDEs (e.g. IntelliJ)
    - Maven / Gradle

Support staff

JUnit 5 has no dedicated support staff. Whenever a user runs into problems they have several options:

1. **Consult the JUnit 5 user guide** [7]. JUnit 5 has an extensive user guide that is actively maintained by contributors and the main five developers identified above.
2. **Post a question on Stack Overflow** [8]. Stack Overflow has a very active community of developers in general and JUnit 5 contributors. Posting questions here often yields good results.
3. **Open an issue on Github** [9]. For more complicated problems or suggestions for additional functionality a user can open an issue on GitHub, requesting changes or features.

Communicators

Communicators are the experts of the system that explain the system (and its architecture) to others and additionally train the support staff of the system. Due to the lack of a dedicated support staff, the training thereof is not an applicable task in JUnit 5. Therefore the first place to go to if you want to obtain information about the system would be the user guide [7]. Additionally, if we look at the support options that are provided to end users, we can see that the five main contributors that we identified play an important role in the provision of support. They work on the support guide and comment on issues on GitHub. When looking at these five contributors, we can identify two contributors that are the most pro-active in their communication:

- @sbrannen

- @marcphillipp

Additionally we know that these two contributors spread their knowledge by giving lectures at conferences and institutions. Marc Phillip gave a guest lecture at the TU Delft, and Sam Brannen has held talks about JUnit [10] during conferences in the past.

Another type of communicator that is not necessarily directly connected to JUnit is teachers. Teachers can introduce students to the JUnit testing framework and familiarise them with testing principles. An example in this category would be Arie van Deursen.

Users

Users are the end users of JUnit 5 who use JUnit to test their software. According to statistics mined from Github [11], JUnit 4 has a lot of users, since it is the third most used package in Java applications right after Java.util and Java.io.

Ecosystem enhancers

We define *ecosystem enhancers* as stakeholders that provide additional functionality on top of the features that JUnit provides. There are quite a few of these ecosystem enhancers. Some examples are:

- Mockito
- AssertJ
- Jukito
- Cucumber
- Spring

Especially Spring is an interesting stakeholder here because one of the main developers of JUnit, @sbrannen, is also a main contributor of the Spring Framework.

Competitors

While JUnit is the largest testing framework for the JVM, it is not the only one. There are several other testing frameworks available for the JVM. Some examples are:

- Scalatest
- TestNG
- Spock

Power to interest

We have created a power vs. interest grid to show the importance of stakeholders in comparison to their interest in the development of the JUnit 5 framework. The most important stakeholders we have identified are 'the big 5'. These are the five main contributors we identified above. You can see the corresponding stakeholder grid we have created in Fig. 1.

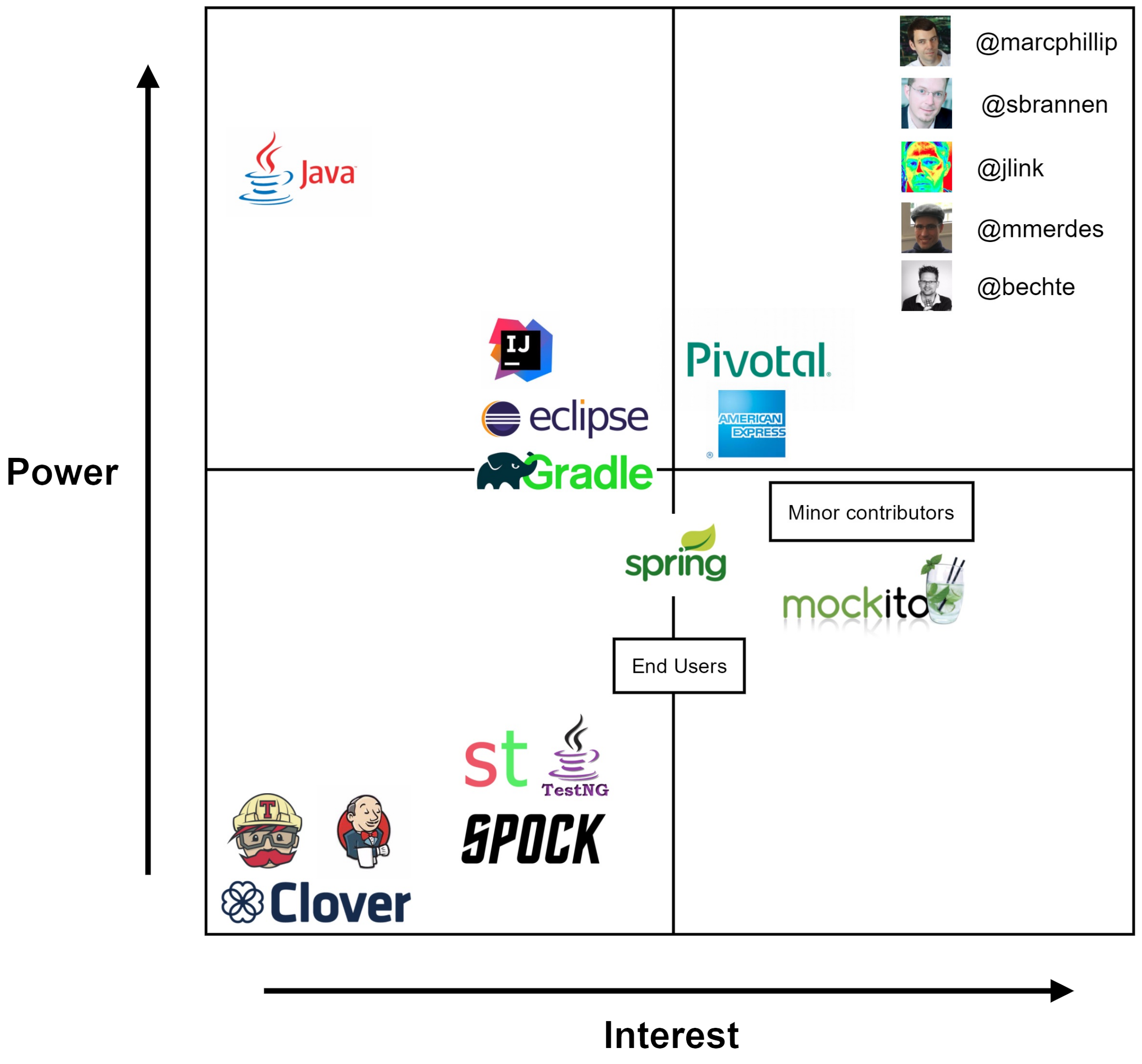


Fig. 1: The power to interest grid for JUnit 5 stakeholders

Context view

In the book *Software Systems Architecture* a context view is defined as follows: "Describes the relationships, dependencies, and interactions between the system and its environment" [1]. The environment that is mentioned is even further specified: "the people, systems, and external entities with which it interacts". To place JUnit 5 into proper context, we combine information from multiple sources: stakeholders that we have identified, together with their representation in both JUnit 4 and JUnit 5.

OTA4J

An interesting new development is the creation of the 'Open Test Alliance for the JVM'. This alliance aims to create a standardised way of test assertion failures and errors. This allows IDEs to improve the way different testing frameworks are integrated. The alliance was initiated by the architects of JUnit. Several parties, including IDEs and several (competing) testing frameworks have been contacted about the initiative. In Fig. 2 we have visualised the different parties that have been contacted. It should be noted that these parties are not initiated by the architects of JUnit. Several parties, including IDEs and several (competing) testing frameworks have been contacted about the initiative. In Fig. 2 we have visualised the different parties that have been contacted. It should be noted that these parties are not necessarily involved with the initiative (yet).

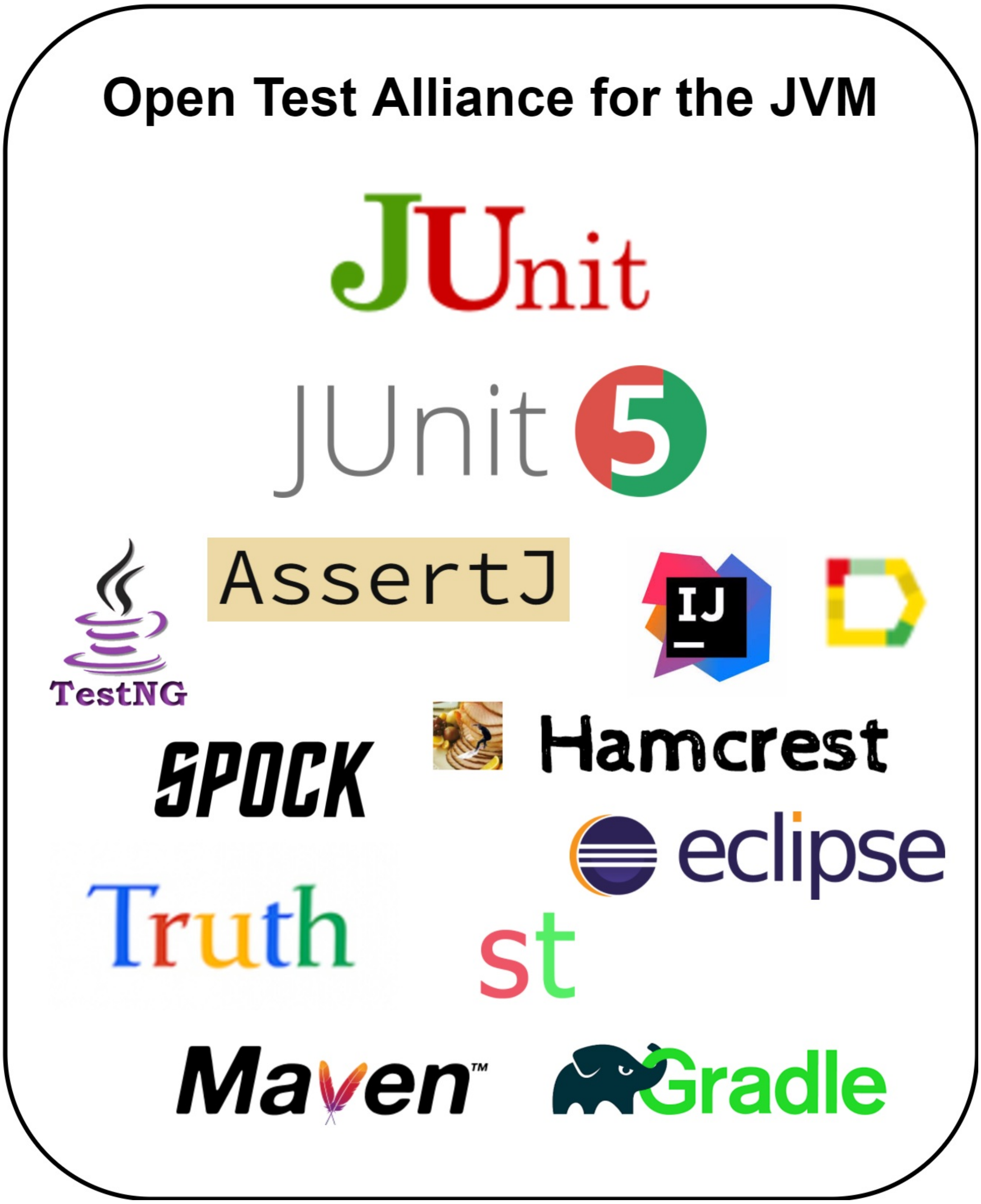


Fig. 2: All parties that have been contacted about the OTA4J.

Context View Visualisation

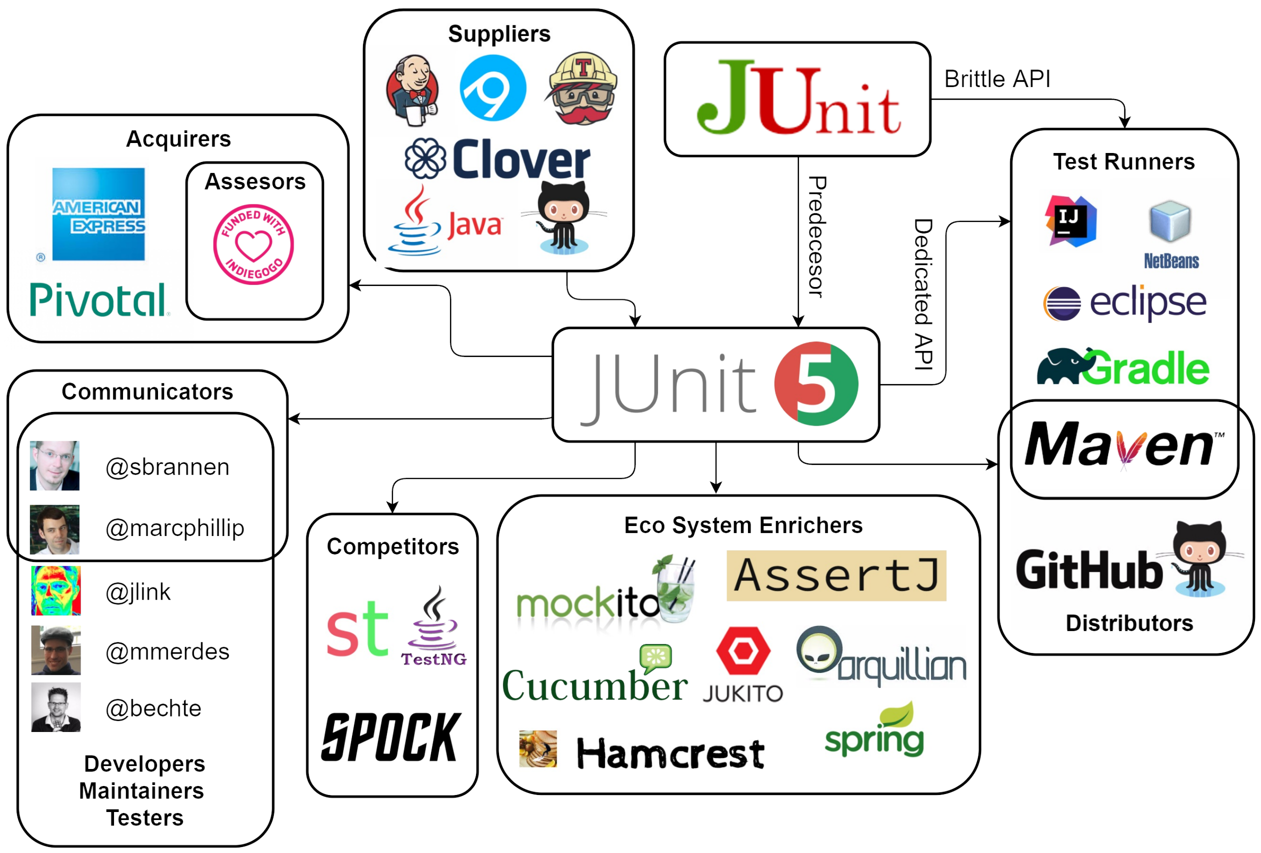


Fig. 3: The context view for JUnit 5.

You can see the main contributors of JUnit 5 (and to a certain extent JUnit 4) as the most important stakeholders, because they decide which features are included and are not. Furthermore we have decided to include American Express as an acquirer because they were one of the main sponsors of the Indiegogo campaign that initiated the development of JUnit 5 as mentioned before. We have also included Pivotal as acquirer because they were one of the main contributors to the Indiegogo campaign; they contributed cash as well as 6 weeks of developer time for the development of lambda [35].

Noteworthy is the relation between JUnit 4 and JUnit 5 as respectively predecessor and successor, we talk about this in more detail in the evolution perspective. We have tried to visualise the relation between the two versions of the framework and the IDEs in light of the API that JUnit provides for them. You can also see there are a great many 'Ecosystem enhancers'. These are frameworks or plug-ins that use JUnit and provide additional functionality, support or integration options. The relationships between ecosystem enhancers themselves shift greatly with the introduction of JUnit 5, allowing them to work together in a better way. More information on groups of stakeholders can be found in our stakeholder analysis.

Functional View

Capabilities

Functional View

Capabilities

From the JUnit 5 user guide three key capabilities can be found [16].

1. The JUnit Platform serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform.
2. JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5.
3. JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

JUnit 5 should be able to run all sorts of tests on the JVM, including those from other frameworks. It also provides its own new test engine with new features and extension possibilities. JUnit 5's approach must be able to support the older versions of JUnit as well. We will now introduce architectural principles that we think may have driven the design.

1. **Backwards compatibility** and migration are required for JUnit 5's adoption.
2. Any entity integrating with JUnit 5 should do so in a **loosely coupled** manner.
3. Key stakeholders should be provided with a **dedicated interface** for their tasks.
4. **Minimal dependencies** to further drive loose coupling.

External Interfaces

In Fig. 4 one can identify three external interfaces:

1. jupiter-api
2. junit-4
3. build-tool-plugins

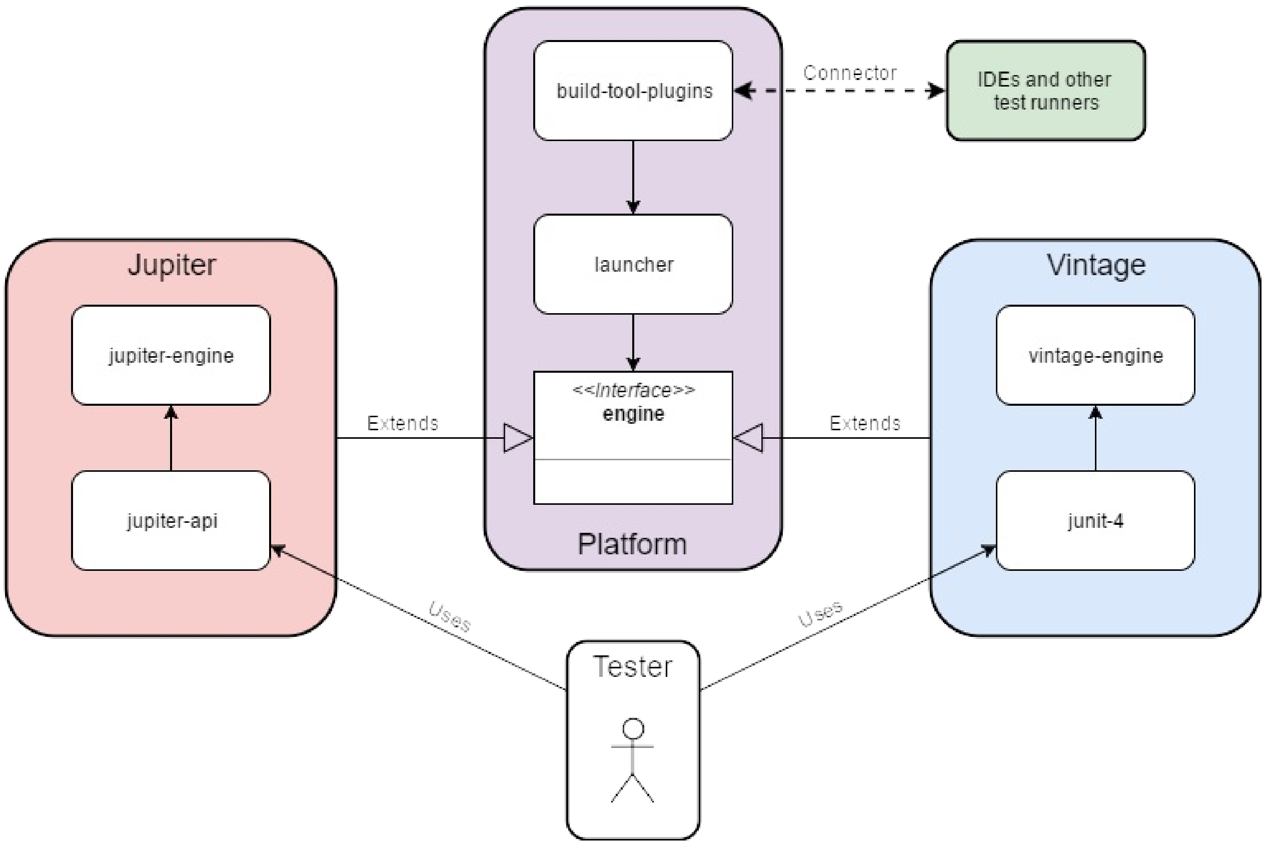


Fig. 4 External interfaces of JUnit 5.

We will now discuss the responsibilities and the philosophy underlying the design of each of these interfaces by applying the architectural principles we identified.

Firstly we consider the jupiter-api. jupiter-api strictly follows the minimal dependency principle. This interface therefore only contains a declarative API without implementation.

Secondly, the JUnit 4 interface violates the minimal dependency principle. JUnit 4 is a rather fat dependency, but it is still provided as an external interface due to the backwards compatibility principle. This indicates that the JUnit architects consider backwards compatibility to be more important than minimal dependencies.

Lastly, the dedicated interface principle applies when we consider the build-tools. In JUnit 4 build-tool integration was not taken into consideration. Because the dedicated interface principle was judiciously applied this relationship has now rigorously changed. We highlight here that it is an architectural principle of importance and discuss the effects in the evolution perspective.
The build-tools also adhere to the minimal dependency principle; they consist of a different and minimal dependency for every build-tool JUnit wishes to integrate with.

There is another external interface that is a bit harder to identify. It currently is not an external interface, but will become one in the future. When the build-tool-plugins migrate to their owners, they will depend on the launcher module. The launcher module will then

become an external interface.
In this case the loose coupling principle becomes key, this principle has been mainly instated to avoid the situation in JUnit 4 (see evolutionary perspective).

Development View

This section consists of the development view of the JUnit 5 project. We have divided this section into several subsections: module structures, development guidelines, and testing approach.

Module Structures

Due to the size of JUnit 5, the code has been split into four *sub-projects* [16] and each of these consists of multiple *modules*. Modules are parts of the code base that are related to each other and are therefore grouped together. A sub-project is a group of these modules that are related and are therefore grouped together.

These sub-projects and modules were made to create a (for humans) logical structure of the code base, granting three benefits:

1. It allows for a better understanding of the code base.
2. It gives better insight into the possible effects changes in one area of the code may have on other areas of the code.
3. It makes the code base easier to maintain.

This section consists out of two subsections, where we identify and classify the sub-projects and identify and discuss the module dependencies.

Sub-projects

Open Test for Java (OT4J)

We discuss it as a sub-project, but will not dive deeper into a module analysis.

OT4J is an initiative from the JUnit team to provide a minimal common foundation for all testing frameworks [17]. OT4J aims to provide the core abstractions of the testing domain. For this reason it is a completely separate project from JUnit, but JUnit does depend on it. The only core abstraction included so far is a common set of exceptions. Introducing this common point should reduce the workload for integrators with all testing frameworks.

If all test frameworks adopt these exceptions, it should reduce the workload for build tools that integrate with these test frameworks.

junit-platform

JUnit platform is the dedicated API for running and reporting JUnit tests. It provides a single interface for running both JUnit jupiter and JUnit vintage tests. It aims to be a more flexible and refined way for test-runners to integrate with JUnit 5 compared to JUnit 4.

junit-jupiter

JUnit jupiter is the new end-user facing functionality. This functionality consists of new features for testers and a more composition-friendly extension model for the eco-system-enhancers.

junit-vintage

JUnit vintage is a reincarnation of JUnit 4, however its tests are run through a JUnit platform compatible engine allowing it to be executed through the junit-platform.

Module dependencies

In this section we will view the dependencies between the JUnit model through a more conceptual model that we created, which allows us to focus more on the key dependencies between the modules, allowing for a more thorough understanding of the system. The JUnit 5 code base has been split into twelve different modules. Detailed information about all different modules in the JUnit 5 framework can be found in the user guide [18].

Note that there is no layering of different abstraction levels found on the module level in JUnit 5. Rather, each module contains their own abstraction levels from high to low and as such there is no real layer system in JUnit 5. This is why there will be no subsection dedicated to the layering.

In Fig. 5 you can see our conceptual module dependency model (the official dependency diagram can be found in the JUnit 5 user guide [19]).

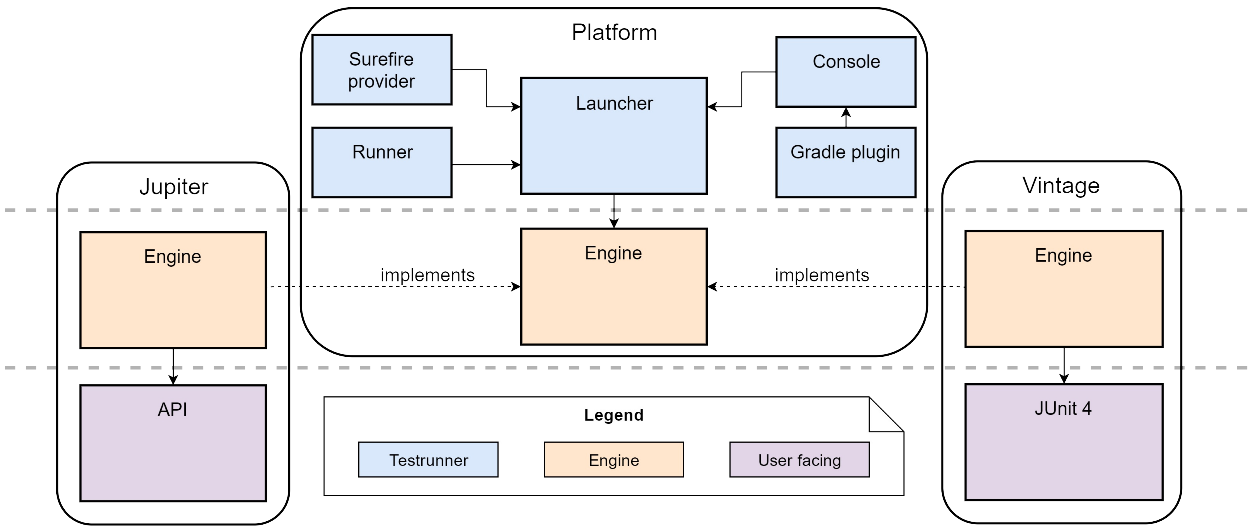


Fig. 5: The module dependency model for JUnit 5.

This model illustrates the key dependencies between the modules and organises the modules by the concerns they address.

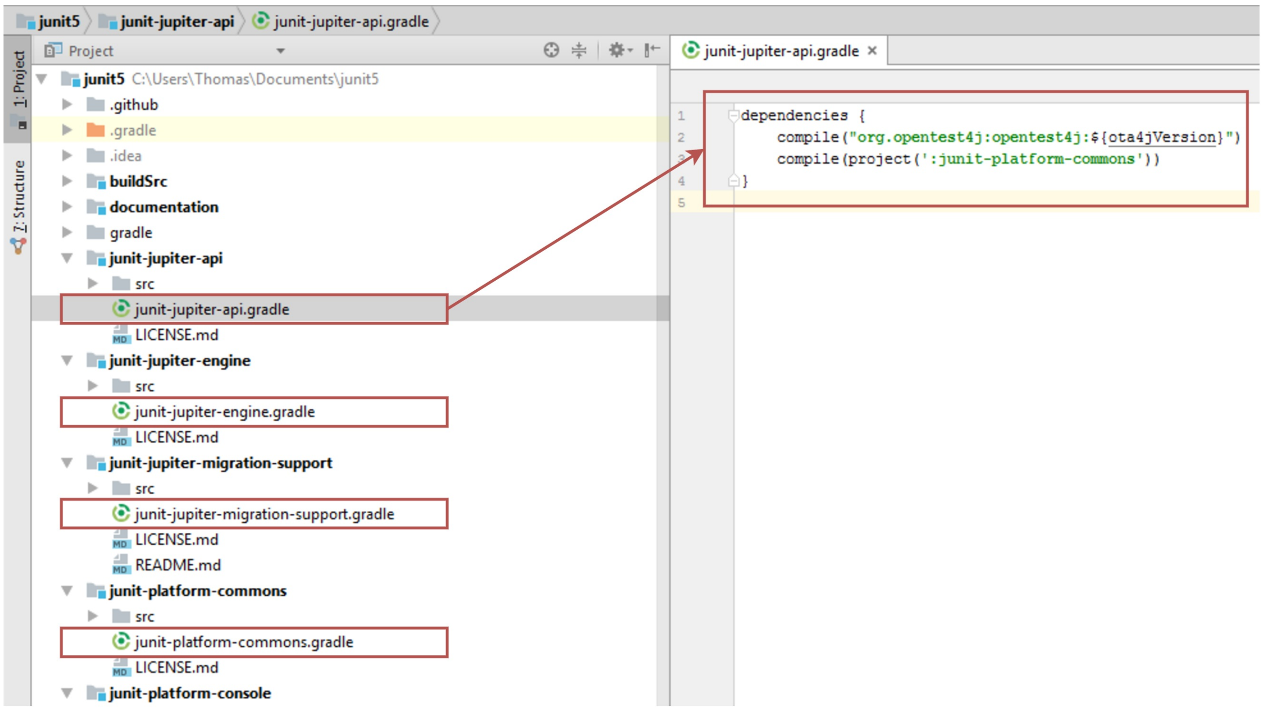
1. **Testrunner:** Integration of test runners with the JUnit platform.
2. **Engine:** Implementation of these testing features and its integration with the JUnit platform.
3. **User facing:** Provides the API for end users for testing with a particular engine.

Development Guidelines, Rules and Releases

In this section we'll zoom in on the policy revolving around the development of JUnit 5. When a contributor wants to contribute new code to the JUnit 5 project they have to adhere to certain rules.

Source code structure

When looking at the source code structure of the JUnit 5 project you can clearly recognise the same structure that has been discussed in the section on module structures. Every module has its own corresponding Gradle [20] module in the project.



```
dependencies {
    compile("org.opentest4j:opentest4j:${ota4jVersion}")
    compile(project(':junit-platform-commons'))
}
```

Fig. 6: Project view of JUnit 5 in IntelliJ IDEA.

In the project view of for instance IntelliJ (see Fig. 6) you can see the module structure of the entire project. In each module there is a separate `module-name.gradle` file that contains a list of all other modules that your module depends on. This gives contributors a quick overview of what code the module they want to modify depends on.

Coding conventions

The first, most elemental thing you should take into consideration when contributing to JUnit 5, is the way the code you write is structured. You should make sure that your code is written in the same style as existing code, otherwise you will introduce style discrepancies in the project. All the exact style rules are written down in the `contributions.md` file on the JUnit 5 repository [21]. JUnit 5 offers a file with 'formatter settings' that can be used in several IDEs. If the formatter settings are imported from the file and used for contribution they will automatically be formatted appropriately. Apart from this formatter, you can locally use `./gradlew check` as well. This check runs the so-called 'spotless' plug-in as well as checkstyle. The spotless plug-in can also automatically add the correct licences to files with the `./gradlew spotlessApply` command, this is a separate action though and does not happen during `./gradlew check`. This takes away some tedious work from the developer as licences can differ per file.

The release process

JUnit 5 has no fully automated releases but does feature an automated release process. When looking at the `gradle.build` file in the root of the JUnit 5 repository, we see that there is a section dedicated to the deployment of JUnit 5 to the Maven Central Repository [22]. The deployment sequence has to be initiated manually though.
In the roadmap document on the repository the JUnit 5 architects describe several phases in the development of JUnit 5 [23]. They intend to release a new version of JUnit 5 after each of these phases has been completed. The project is currently in *phase 5*, which means that an initial Alpha version of JUnit 5 has been released and more work is being done on additional milestones [24]. When searching on Maven Central [25] there are already three milestone releases to be found. The next milestone release (Milestone 4) is planned to be completed by March 18th 2017 [26]. After this milestone there will at least be one more milestone to be completed. After that the project will be almost ready for a production release and the architects will prepare for this by releasing one or more so-called 'release candidates' before making a 'GA' release. In Fig. 7 the development and release stages are illustrated.

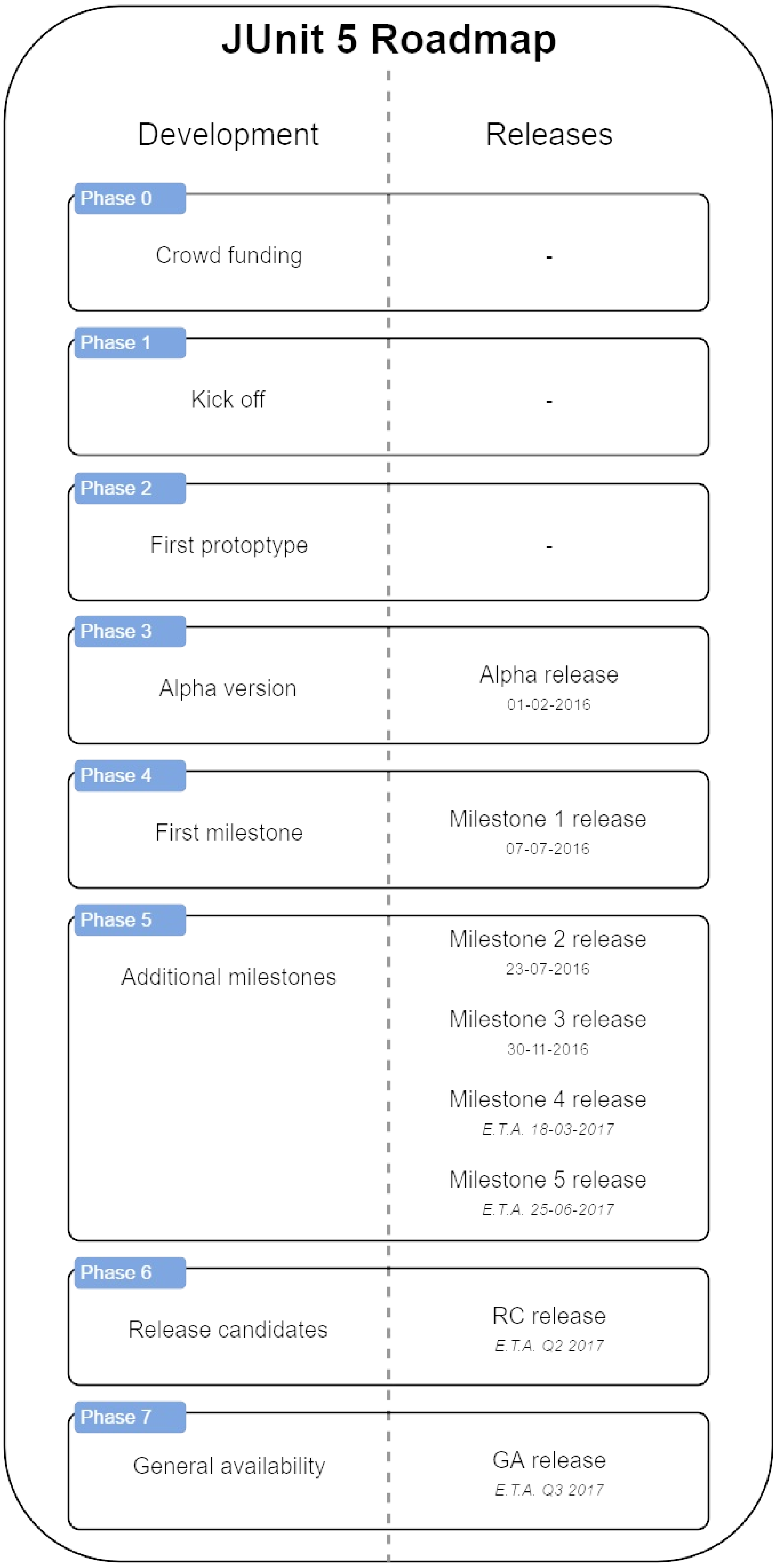

JUnit 5 Roadmap
Development
Releases
Phase 0
Crowd funding
-
Phase 1
Kick off
-
Phase 2
First protoptype
-
Phase 3
Alpha version
Alpha release
01-02-2016
Phase 4
First milestone
Milestone 1 release
07-07-2016
Phase 5
Additional milestones
Milestone 2 release
23-07-2016
Milestone 3 release
30-11-2016
Milestone 4 release
E.T.A. 18-03-2017
Milestone 5 release
E.T.A. 25-06-2017
Phase 6
Release candidates
RC release
E.T.A. Q2 2017
Phase 7
General availability
GA release
E.T.A. Q3 2017

Fig. 7: The roadmap for the development of JUnit 5.

The JUnit 5 Test Approach

The testing approach used in JUnit 5 is peculiar, and therefore has its own section. This section combines information from the development guidelines and the module structure. The development guidelines specify that every change needs to be covered by tests. However, as a developer, finding the correct place in the project to implement these tests is difficult, and requires technical information about the dependencies between the JUnit 5 modules.

In this section we will first say something about the process that the developers of JUnit 5 used to improve their testing framework without having to release it first. The next section clarifies the dependencies between several different modules and explicates why tests are not always in intuitive places in JUnit 5.

Development feedback cycle

In order to create useful features, the JUnit 5 team, like every other development team, needs end-user feedback. Testing JUnit 5 *with* JUnit 5 allowed the developers to turn themselves into end-users and receive this feedback early on. Using their own features gives the developers first-hand experience and allows them to immediately determine whether their creations are useful and easy to work with, essentially creating a feedback loop. Of course the early milestone releases also play a key role by providing extra feedback. Another benefit of this testing approach is that every test case written by the team, doubles as an end-to-end test for the system, providing extra coverage and confidence in the system.

Module dependencies

The testing approach also comes with a downside. It complicates certain dependencies between modules, resulting in the tests for certain modules ending up in odd locations.

It is the intention of the JUnit 5 team to design the JUnit platform in such a way that it will be used by many future versions of JUnit. Testing JUnit platform and JUnit jupiter *with* JUnit jupiter minimises the dependency on JUnit 4, and would facilitate dropping JUnit 4 in the future (if desired).

First we will take a look at the actual dependency diagram given by JUnit, visible in Fig. 8.

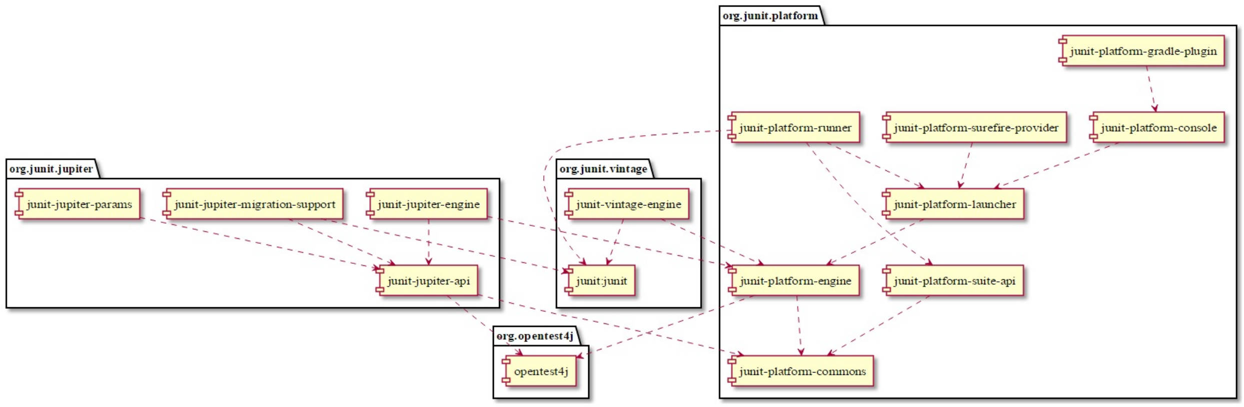


Fig. 8: The dependency graph of JUnit 5 [19]

Testing with JUnit jupiter requires three dependencies:

1. The `junit-jupiter-api` .
2. The `junit-jupiter-engine` .
3. The `junit-gradle-plugin` and its transitive dependencies.

With these dependencies in mind, we will take a look at writing tests for JUnit.

JUnit jupiter

Test for the `junit-jupiter-api` module do exist, but they are not present in the module itself. The tests for the `junit-jupiter-api` module reside in the `junit-jupiter-engine` module. The reasons why these tests are located here rather than in the `junit-jupiter-api` module is that the `junit-jupiter-engine` depends on the `junit-jupiter-api` . This means that if the tests for the `junit-jupiter-api` were placed in the `junit-jupiter-api` module itself, a dependency on the `junit-jupiter-engine` would need to be added in the `junit-jupiter-api` module to be able to run these tests. This would result in a circular dependency, which can also be seen in the dependency diagram if an arrow would be added originating from `junit-jupiter-engine` travelling to `junit-jupiter-api` .

The `junit-jupiter-engine` module already depends on the `junit-jupiter-api` module; the same goes for `junit-jupiter-params` . Tests can depend on the production code of their own module satisfying the dependency on the `junit-jupiter-engine` . The dependency graph shows that nothing in the `junit-platform` sub-project depends on `junit-jupiter sub-project` , allowing us to depend on `junit-platform` without introducing any cycles.

JUnit platform

Since JUnit is built using Gradle [20], all tests are run through the `junit-gradle-plugin` (listed as the third dependency required for writing tests). The dependency diagram shows that every module, with the exception of the `junit-platform-surefire-provider` and `junit-`

`platform-runner`, are either direct or transitive dependencies of the `junit-platform-gradle-plugin`. If any of these modules were to contain any tests it would introduce a dependency cycle.

To combat this phenomenon, JUnit has extracted one more module (not present in this diagram); `junit-platform-tests`. This module has no production code and only contains test code. It contains the tests for the following junit-platform modules: *commons*, *console*, *engine*, *launcher* and *runner*. Of the two modules that are not dependent on the `junit-gradle-plugin` (*runner* and *surefire-provider*), the `maven-surefire-provider` module is the only one that hosts its own tests. The `junit-suite-api` only contains annotations and has no tests. This leaves us with two modules in the diagram that still need to be covered:

1. `junit-vintage-engine`
2. `junit-migration-support`

The `junit-vintage-engine` and the `jupiter-migration-support` can both safely depend on the gradle plugin (just like the `junit-jupiter-engine`) and can host their own tests. Finally, several modules depend on JUnit 4, but all of JUnit 4's tests are contained in JUnit 4's own project.

Evolution Perspective

JUnit 5 is the next generation of JUnit. The goal is to create an up-to-date foundation for developer-side testing on the JVM [28]. Its development can be nicely argued for from an architectural standpoint, it resolves limitations present in JUnit 4. The architects of JUnit have identified several shortcomings of JUnit 4 over the years and have tried to address them as best as possible in the new version of JUnit.

Problems in JUnit 4

Normally in projects the size of JUnit, technical debt is a key factor inclines developers to rewrite their software product, but in the case of JUnit there is fairly little internal technical debt. The technical debt in the production code, considering the size of the project, is very limited in both JUnit 4 and JUnit 5. We were unable to discover any major issues on technical debt in the production code. Both versions of the framework are also very well tested, JUnit 4 has 89% line coverage, and JUnit 5 has an even higher coverage of 95.5% making the project extremely well tested [29].

With technical debt out of the picture the two the key issues in JUnit 4 that are addressed with the development of JUnit 5 are:

1. IDE integration

2. Extension Model

IDE integration

IDE integration is vital for a testing frameworks survival, however the way in which this integration is achieved in JUnit 4 leaves a lot to be desired. Many tools reach deeply into JUnit for their functionality, using clever hacks to get past what little boundaries are present, resulting in an ad hoc, undocumented and brittle API. Keeping this brittle API intact severely limits any kind of development on JUnit. This situation came to pass because JUnit 4 neglected IDEs as an important stakeholder for their framework and did not provide proper integration options for the IDEs. Because of the significant market share JUnit 4 has, this resulted in the IDEs getting the information they desired on their own. To emphasise this problem we can look at JUnit 4 issue 444 [31]. This issue calls for an exhaustive listener framework in JUnit that could be utilised by IDEs. This issue has been created in 2012 (while JUnit 4 has been around since 2005) and is still opened. Therefore we can conclude that this issue has not been solved for JUnit 4 as of now (and may never be).
In JUnit 5 this problem has been resolved by loose coupling and dedicated interfaces as is described in the functional view.

Extension model

The most powerful extension possibility present in JUnit 4 is a TestRunner. It gives a lot of control how tests are run and has been a key to the success for integrating other tools. As can be seen in our stakeholder analysis there is a vast amount of libraries and, as we call them, 'ecosystem enhancers', providing their functionality by extending JUnit. These tools are successful in enriching JUnit by providing additional functionality, indicating their individual needs as stakeholders have been met. Many of these tools use the test runner to achieve their goals. However JUnit 4 comes with a limitation: each test suite can only utilise a single test runner. This makes different tools that solve different problems compete for the runner for no reason. This results in tools with completely different goals being unable to function in combination with each other. JUnit 5 introduces a new extension model that allows tools to work together and even provide new functionality. Details on how the extension model of JUnit 5 solves the problem can be found in the user guide [34].

The road from JUnit 4 to JUnit 5

To address the architectural issues in JUnit 4, the JUnit team readjusted their stakeholder priorities. This can be seen in the JUnit Lambda kickoff [12] which had IDE and build tool owners present to discuss the first start of the work on JUnit 5. Further evidence can be found in the communication between the JUnit team and Mockito [13], where the JUnit team discusses the concerns of Mockito in their new extension models and tries to incorporate Mockito's needs. While this issue is currently a hot topic, there are issues that date back further where extension points in JUnit 4 and 5 are discussed [14] [15].

To further elaborate on the relationship between JUnit 4 and JUnit 5 and the effect this migration will have on the ecosystem we want to highlight a few important decisions made by the architects. To limit the impact of the transition from JUnit 4 and JUnit 5 for end user and ecosystem enhancers, the architects of JUnit have created a dedicated migration support platform. Users are able to use a combination of JUnit 4 and JUnit 5 tests during a transition period using this system. This way they won't have to adjust all of their tests overnight. JUnit 4 lives on in the JUnit platform as JUnit Vintage.

You can see a flow chart that visualises the transition from JUnit 4 to JUnit 5 in Fig. 9.

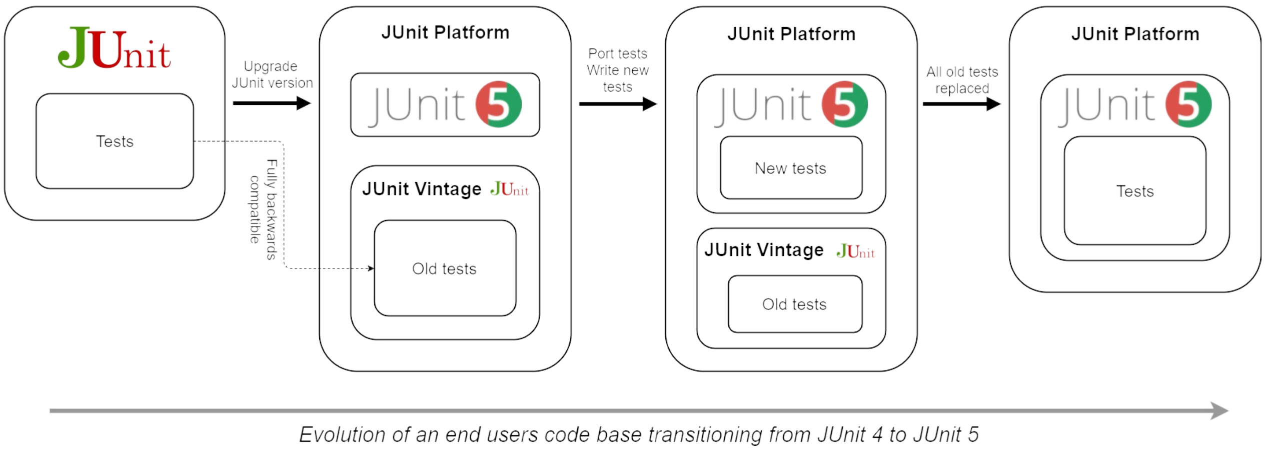


Fig. 9: A flowchart of the transition from JUnit 4 to JUnit 5 for a typical system.

JUnit secures its future by upgrading their build dependency from Java 1.5 to Java 1.8 [30]. While it is perfectly possible to use JUnit 4 in combination with Java 1.8, this upgrade allows the JUnit team to use Java 1.8 while developing JUnit 5, thus allowing them the use of lambdas, optionals and new types in the standard library among other things.
These new features and solutions ensure that JUnit 5 will become just as versatile and loosely coupled as envisioned by the JUnit architects.

Conclusion

Even though JUnit 5 is still in development at the time of writing and has yet to see the first official release, it has already accomplished a lot. In according with their vision, the JUnit team was able to create a solid foundation which solves the architectural problems JUnit 4 was facing. JUnit 5 already supports most of the features of JUnit 4 and has improved and added features on top of that. The support for JUnit 4 eases the transition from JUnit 4 to JUnit 5 and as a result, we can expect more and more projects to do so.

The JUnit 5 team also managed to keep the code base of the project extremely clean, accumulating negligible amounts of technical debt and maintaining an extremely high testing coverage (average of 95.5%). These high standards reflect their desire to create a lasting foundation, and shows their perseverance in accomplishing it.

In our opinion, the JUnit 5 team is on the right track towards realising their vision. We have no doubts that the JUnit 5 team will be successful due to how far they have managed to come and the quality of work they have been able to maintain during this period.

We look forward towards the official release and public adoption of the JUnit 5 project.

References

1. Nick Rozanski and Eoin Woods. 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.
2. JUnit lambda campaign, http://junit.org/junit4/junit-lambda-campaign.html, access date: 30-03-2017
3. The Java programming language licence, http://www.oracle.com/technetwork/java/javase/terms/license/index.html, access date: 30-03-2017
4. The Apache surefire licence, https://maven.apache.org/surefire/, access date: 30-03-2017
5. The Gradle licence, https://github.com/gradle/gradle/blob/master/LICENSE, access date: 30-03-2017
6. The JUnit 5 licences on Github, https://github.com/junit-team/junit5/blob/master/LICENSE.md, access date: 30-03-2017
7. The JUnit 5 user guide, http://junit.org/junit5/docs/current/user-guide/, access date: 30-03-2017
8. Stack Overflow, http://stackoverflow.com/, access date: 30-03-2017
9. Issues for JUnit 5 on Github, https://github.com/junit-team/junit5/issues, access date: 30-03-2017
10. Sam Brannen pitch about JUnit, https://www.youtube.com/watch?v=UHN_HcjZa7o, access date: 30-03-2017
11. Statistics about package use in Java projects on Github, *Google*, https://cloud.google.com/bigquery/public-data/github, access date: 30-03-2017
12. JUnit lambda campaign on Indiegogo, https://www.indiegogo.com/projects/junit-lambda#/, access date: 30-03-2017
13. Issue on Mockitos Github with communication with JUnit 5 stakeholders, https://github.com/mockito/mockito/issues/445, access date: 30-03-2017
14. Pull Request 1158 on the JUnit 4 Github, https://github.com/junit-team/junit4/pull/1158,

access date: 30-03-2017
15. Issues 1161 on the JUnit 4 Github repository, https://github.com/junit-team/junit4/issues/1161, access date: 30-03-2017
16. JUnit 5 user guide, JUnit 5 overview, http://junit.org/junit5/docs/current/user-guide/#overview-what-is-junit-5, access date: 02-04-2017
17. Open Testing Alliance for the JVM, https://github.com/ota4j-team/opentest4j, access date: 02-04-2017
18. JUnit 5 user guide, dependency metadata, http://junit.org/junit5/docs/current/user-guide/#dependency-metadata, access date: 02-04-2017
19. JUnit 5 user guide, dependency diagram, http://junit.org/junit5/docs/current/user-guide/#dependency-diagram, access date: 02-04-2017
20. Gradle, https://gradle.org/, access date: 02-04-2017
21. JUnit 5 repository on Github, contributions.md, https://github.com/junit-team/junit5/blob/master/CONTRIBUTING.md, access date: 02-04-2017
22. Maven Central Repository, http://central.sonatype.org/, access date: 02-04-2017
23. JUnit 5 road map, https://github.com/junit-team/junit5/wiki/Roadmap, access date: 02-04-2017
24. JUnit 5 milestones, https://github.com/junit-team/junit5/milestones, access date: 02-04-2017
25. JUnit 5 milestones released on Maven, https://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.junit.jupiter%22%20AND%20a%3A%22junit-jupiter-engine%22, access date: 02-04-2017
26. JUnit 5 Milestone 4 on Github, https://github.com/junit-team/junit5/milestone/7, access date: 16-03-2017
27. Git, https://git-scm.com/, access date: 02-04-2017
28. JUnit 5 homepage, http://junit.org/junit5/, access date: 02-04-2017
29. JUnit 5 test coverage by Clover, https://junit.ci.cloudbees.com/job/JUnit5/clover/dashboard.html, access date: 02-04-2017
30. JUnit 4 dependencies, http://junit.org/junit4/dependencies.html, access date: 02-04-2017
31. JUnit 4 issue 444 on Github, https://github.com/junit-team/junit4/issues/444, access date: 02-04-2017
32. JUnit 5 release date, https://github.com/junit-team/junit5/milestone/10, access date: 02-04-2017
33. OTA4J usage licence, https://github.com/ota4j-team/opentest4j/blob/master/LICENSE, access date: 03-04-2017
34. The JUnit 5 user guide, extensions, http://junit.org/junit5/docs/current/user-guide/#extensions, access date: 03-04-2017
35. Tweet on Indiegogo sponsoring by Sam Brannen,

https://twitter.com/sam_brannen/status/861202719293530113, access date: 07-05-2017

Jupyter Notebook

By Lorenzo Gasparini, Ajay Adhikari, Giannis Papadopoulos and Anelia Dimitrova

Delft University of Technology

Abstract

Jupyter notebook, the next generation of IPython, is an open-source web-application that allows the user to create and share documents containing live code, comments, formulas, images and more, all in one place. The current chapter analyses this project in depth. Our team was able to make six pull requests regarding technical debt and bug fixes, which were accepted. A good overview of the whole project is given by analysing stakeholders and the context view. Next, various viewpoints and perspectives on the architecture of the project are investigated. This chapter aims to help people who would like to join the Jupyter community and make contributions, by providing extensive analysis of the project.

Table of Contents

Introduction

Jupyter Notebook is a cross-platform open source web application which allows running code, visualising its output (such as plots, images, tables etc.) and writing explanations in natural language, accompanied by equations and all this in one so-called Notebook file. It provides the opportunity to process large amount of data and supports **Ju**lia, **Pyt**hon, **R** (origin of the name **Jupyter**) and many other programming languages. It is intuitive, easy to install and use, which makes it suitable for scientists, programmers and learners.

It is adopted by teachers in universities such as UC Berkeley where they use JupyterHub (server, hosting Notebooks where users log in and immediately start coding) in which students do their assignments. Furthermore, Github supports the rendering of Notebook `.ipynb` files. Entire books have been written with Notebook and published on Github. Multiple online courses on Machine Learning, Computer Science, Python programming, Data Analysis and others can be found in this Github repository.

This chapter aims to give users a quick understanding of how Jupyter Notebook is organised, developed and maintained. First, the Stakeholders and Context view are analysed to provide insights into the organisation of the project. Furthermore, detailed analysis of the architecture is presented in order to understand its structure. In addition, technical debt in terms of code, testing and documentation was also found throughout the analysis of the system and is presented in the chapter. This is followed by the evolution of the project explaining how Notebook emerged from IPython and discussing the JupyterLab project which is considered as the future of Notebook. Finally, the chapter concludes with our findings of the project.

Organization

This section discusses the parties that are involved in the development, maintenance, testing and building of the system. These stakeholders are then visualised using a context-view diagram.

Stakeholders

A number of different types of stakeholders exist as defined by Rozanski & Woods [1] and this section goes through them w.r.t. Jupyter Notebook.

Assessors

As Jupyter Notebook's team consists of fifteen core developers who oversee the conformance of the (programming & other) standards. They are responsible for handling pull requests and communicate them between each other to decide whether or not to merge. Furthermore, they do the planning of future releases and/or sub-systems of the Notebook.

Communicators

Communicators are people who explain the system and the architecture to other stakeholders via documentation or training materials. A number of presentations are available online, prepared and presented by Fernando Perez and Matthias Bussonier. Jupyter Notebook communicators are also all the developers who have written the documentation. They are a small team and all of them update the documentation when necessary. The documentation is useful as well for developers and contributors to the system.

Developers

Two major types of Developer stakeholders were identified in Jupyter Notebook: core and contributors.

- **Core**: Core developers work actively to write, fix, test and improve the system. Their main job is to also go through the open pull requests on Github and ensure that the contribution proposals are well tested and do not interfere with the current system in any way other than to improve it. Core developers are @carreau, @takluyver, @jasongrout, @rgbkrk, @minrk and @blink1073.
- **Contributors** are important for open source projects as they often solve issues the core team has no time for or implement new features making the product even better. They contribute also by improving documentation, bug fixes and even by writing their own kernel supporting a new programming language. Examples of these contributors are @vidartf and @yuvipanda.

Maintainers

The **Steering Council** consists of Project Contributors writing substantial code of high quality and quantity for over one year. Working with the BDFL (Benevolent Dictator for Life) Fernando Perez the council ensures the long-term progress and existence of the project. The NumFOCUS Foundation serves as the projects' fiscal sponsor and acts as a parent legal entity. The NumFOCUS Subcommittee consists of four Council and one external member.

Support Staff

As per the book [1], support staff are the people who provide support to the users. Therefore Jupyter Notebook members taking care of issues in their Github repository could be considered support staff. These users are usually @minrk and @takluyver. In case they are not sure about something, they know who to get involved.

Testers

Testers are people who test the system to ensure it is suitable to use. Since the Notebook team consists of only fifteen core developers, each of them has to write tests whenever they implement a new feature or improve the current code. All of them can thus be considered testers as well. However, while going through the project a few members were identified who wrote new or updated the old tests frequently: @jdfreder (no longer active), @minrk (core dev), @ssanderson (contributor) and others.

Users

The two biggest categories of users are the following:

- **Scientists:** Jupyter is mainly used by data scientists working with R and Python on daily basis, however, it is not only aimed at them but is generally used by other types of users as well.
- **Learners** are stakeholders who, for example, use Notebook for study & teaching purposes.

Sponsors

Sponsors are an additional stakeholder class to the main ones, related to the project. It consists of companies and universities that support the project financially. Some of them are Google, Microsoft, Rackspace, Leona M. and Harry B Helmsley Charitable Trust, Gordon and Betty Moore Foundation etc. They do not take decisions about the projects' progress, however, they may influence these decisions.

Context View

In this section, the context viewpoint of Jupyter Notebook is described. The context view describes and visualises the relationships and interactions between Jupyter Notebook with the environment.

System Scope

According to the Jupyter foundation website, the Jupyter Notebook is

> an open-source web application that allows you to create and share documents that contain live code, equations, visualisations and explanatory text.

The Notebooks are typically used for data cleaning and transformation, numerical simulation and machine learning tasks; the target audience is the broad tech audience, spanning from high school students to the most advanced researchers of the world.

Jupyter Notebook is the main application of the broader Jupyter project, which focuses on interactive and exploratory computing that is reproducible and multi-language.

Context Model

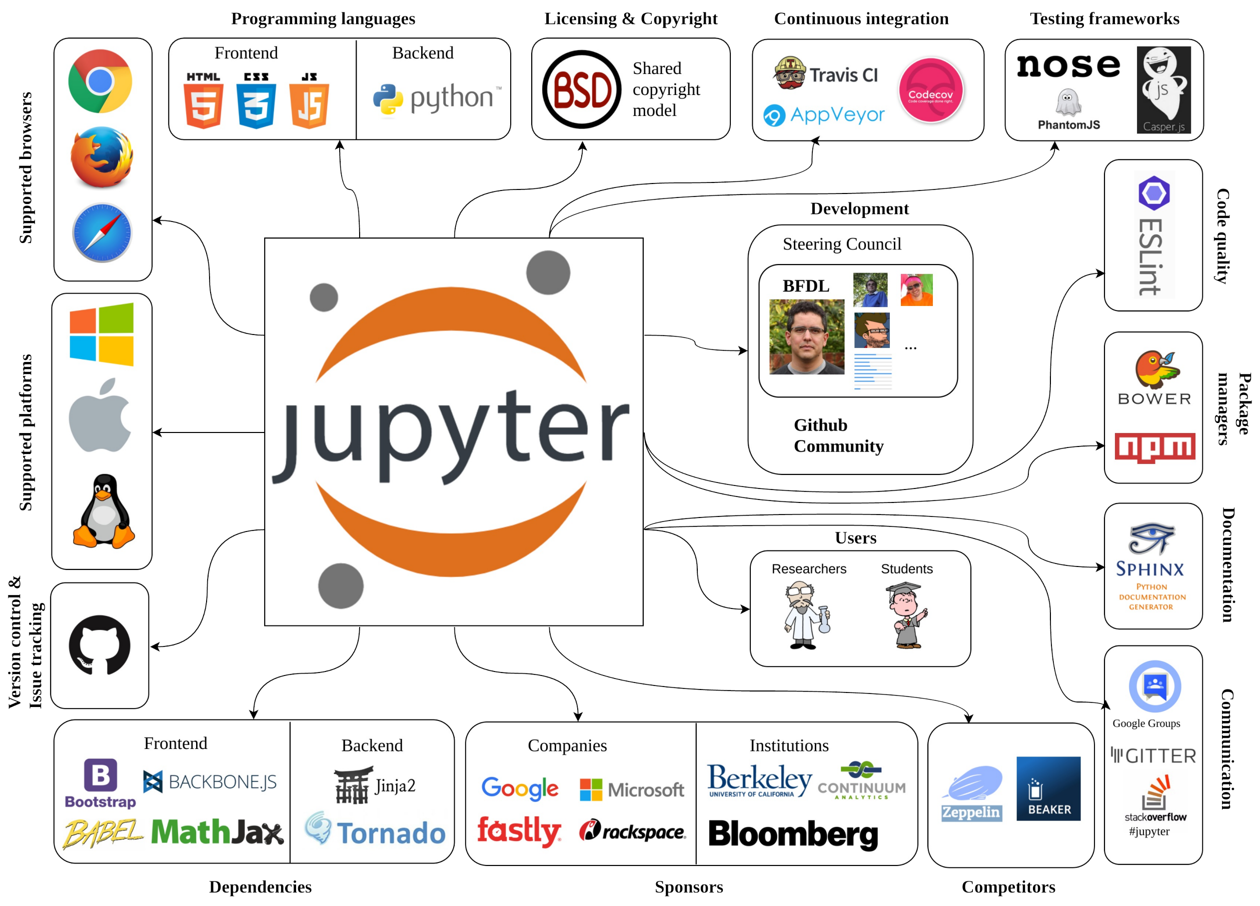


Figure 1 - *Context Model of Jupyter Notebook.*

In Figure 1, the context model of Jupyter Notebook is displayed. A short description of the most important entities in the diagram follows.

The project, which is a complex web application mainly **used by** *researchers* and *students*, can be conceptually separated in *front-end* and *back-end*. For what concerns the **programming languages**, *Python* is used for the back-end while the *LESS* stylesheet language is used in combination with *HTML* and *Javascript* for the front-end.

Thanks to the interoperability of Python, the project can run on most **platforms**, including *Windows*, *Mac OS X* and other *Unix-like* systems. There are no clear indications of officially supported **browsers**, but the documentation recommends the usage of *Chrome*, *Firefox* or *Safari*.

In terms of **testing frameworks**, *Nose* is used for the Python back-end while *CasperJS* is used for the front-end. Tests are executed automatically at each pull request, thanks to **continuous integration**. To this regard, *Travis CI* is used for testing on Linux and *AppVeyor* for Windows. Code coverage is analysed by *Codecov*. *ESLint* is used to enforce the **code quality** on the Javascript sources.

The system has multiple external **dependencies**, both in the front-end and in the back-end. In the former case, the key ones are *Bootstrap*, *Backbone.js*, *CodeMirror*, *Marked* and *MathJax*. In the latter, they are *Jinja2* and *Tornado*. The dependencies are managed using **package managers**, in particular, *Bower* is used for the front-end and *NPM* is used for the building tools.

Sponsoring occurs in two ways: directly from companies like *Google*, *Microsoft* and *Rackspace* or by employing members of the Steering Council. The latter is the case for companies like *Bloomberg*, *Netflix* and for the *Berkeley University of California*. The **development** is carried on by the members of the Steering Council, including the BFDL Fernando Pérez, and by the Github community.

As far as the **documentation** goes, it is maintained using the *Sphinx* documentation generator and published on the *Read The Docs* hosting platform.

The main **competitors** are the *Beaker Notebook* and *Apache Zeppelin*, both of which have shorter history and have not yet reached the maturity level of Jupyter.

Jupyter Notebook is **licensed** using the *3-clause BSD license*, while the **copyright** of the code belongs to the respective authors

Architecture

Development Viewpoint

The development view describes the architecture of a project from the viewpoint of the developers. According to Rozanski and Woods [1], the development view is responsible for addressing different aspects of the system development process such as code structure and dependencies, build and configuration management of deliverables, system-wide design constraints, and system-wide standards to ensure technical integrity. In the following sections, the development view of Jupyter Notebook is presented based on the three models that Rozanski and Woods [1] define in their book: Module Structure Model, Common Design Model and Codeline Model. In addition to this, a high-level view of Jupyter Notebook is included in order to ensure a thorough understanding of the project in different granularity.

High-level view

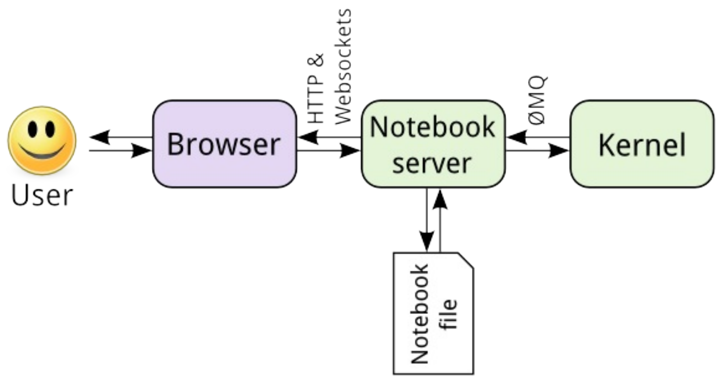


Figure 2 - *High-level view of Jupyter Notebook.* [2]

The high-level view is visualized in Figure 2. First, the user interacts with the browser, consequently, a request is sent to the Notebook server. This can be either HTTP or WebSocket request. If user code has to be executed, the notebook server sends it to the kernel in ZeroMQ messages. The kernel returns the results of the execution. At last, the notebook server returns an HTML page to the user. When the user saves the document, it is sent from the browser to the notebook server. The server saves it on disk as a JSON file with a `.ipynb` extension. This notebook file contains the code, output and markdown notes. [2]

Module Structure Model

The module structure model defines the organisation of the system's code clustering related source code files into modules and determining the dependencies between them [1]. In this section first the modules of the project are briefly described and then the dependencies between them are visualised in a diagram. It should be also noted that this section focuses only on the internal modules of the project and not the external dependencies.

Here follows a short description of the modules that make up the project:

- **notebookapp**: it is the entry point of the web application, including the Tornado-based server; it is responsible for the configuration of the server, including the mapping of requests to the handlers classes.
- **auth**: it manages the authentication to the notebook and other security-related features.
- **base**: it contains the base Tornado handlers that are extended by the other ones throughout the structure of the directory hierarchy; moreover, the handlers for the ZeroMQ sockets, which are responsible for the communication with the kernel, are included.
- **bundler**: it is used in order to bundle the notebook documents, packaging them in tarballs or zip archives.
- **edit**, **kernelspecs**, **nbconvert**, **view**, **files**, **terminal**, **tree**, **notebook**: they contain the handlers for rendering the text editor interface, handling the views that are displayed to the user, converting notebook files to other formats, serving files via the content manager, rendering the terminal interface, displaying the tree view and controlling the live notebook view respectively.
- **frontend**: this module contains the code related to the user interface; in particular, `static` contains the Javascript & CSS sources while `templates` consists of the Jinja2 HTML templates rendered by Tornado.
- **services**: it contains the handlers for the backend services, including kernels, kernel specifications and contents web services; all the handlers registered here start with the `/api/` prefix and communicate with the frontend using JSON whereas the other handlers render the HTML views.

The module structure and dependencies are shown in Figure 3.

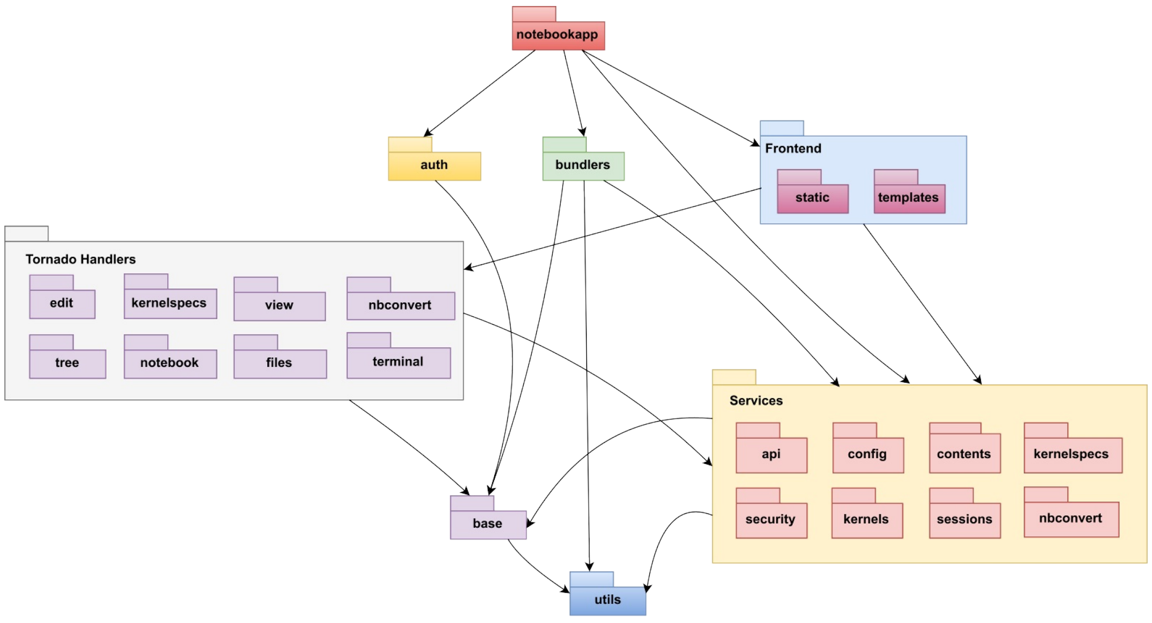


Figure 3 - *Module dependencies diagram.*

Furthermore, it should be noted that the figure above shows high-level dependencies between the most significant modules of the system. In this graph modules that share a similar role in the system are grouped together for the convenience of the reader. The same purpose serve the colours of each of the modules. The diagram starts from the top with the `notebookapp` module which constitutes the entry point of the web application. It is worth mentioning that there are no circular dependencies which makes the system easier to understand and maintain.

Source code structure

The overall structure of the directory hierarchy of the Jupyter Notebook is organised as follows. The root folder consists of several files that are used to configure the system. More specifically, the root folder contains YAML files that are responsible for configuring the continuous integration of the project such as `.travis.yml` and `appveyor.yml` . Other important configuration files that are included in the root folder are `bower.json` , `setup.py` and `setupbase.py` . The `bower.json` manifest file keeps track of the right versions of the front-end packages that are needed for the system. The `setup.py` and `setupbase.py` files are essential for configuring the installation of packages and their dependencies.

Furthermore, the root folder is divided into five separate folders: the `docs/` folder where all the documentation files that are built using Sphinx are located, the `git-hooks/` which consists of custom git hooks that are available for Jupyter Notebook. For example, there is a git hook that can be enabled to automatically compile the LESS stylesheets and the Javascript code after a git checkout. The `scripts/` which contains scripts that are used as an interface to the components of the notebook for the command line, the `tools/` folder containing tools used in the build process and the `notebook/` which contains all of the source code.

The `notebook/` folder is divided into sub-folders by grouping together source files with similar purpose and functionality. First off, it is divided into sub-folders that contain back-end functionality which are structured in a similar way. The related initializers can be found in `__init__.py` for the given sub-folder and next to that every sub-folder contains a module called `handlers.py` in which the Tornado handlers are included and registered. In addition, a `tests/` folder is included which consists of files that test the Tornado handlers functionality.

The front-end code is located in separate folders. In particular, the Javascript and CSS files are located in the `static` folder using as the name for the sub-folders the one of the back-end components they interact with. The Jinja2 HTML templates are included in the `templates/` folder. Concerning the tests that are responsible for testing the front-end functionality, they are contained in the `notebook/tests/` folder. Finally, the `notebook/` folder contains the entry point of the web application which is the `notebookapp.py` module and some common utilities that are used throughout the whole source code. A graphical overview can be found in Figure 4.

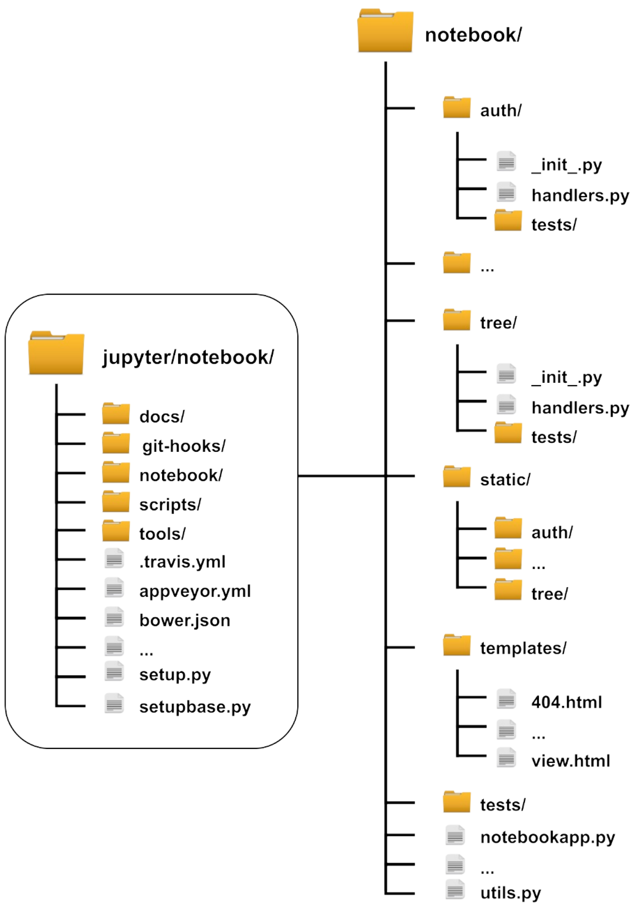


Figure 4 - *Source code structure of Jupyter Notebook.*

Continuous Integration and Testing Approach

Continuous Integration (CI) refers to the development practice in which developers *integrate* the code to a shared repository frequently. Each integration is verified by an automated build and test system.

Testing Frameworks

The Jupyter Notebook project has a Python backend and an HTML + Javascript frontend. There are two separate testing suites: one for the backend and one for the frontend.

The Python **backend** is tested using the `nose` testing framework, a package built on top of `unittest` to simplify test-driven development. This testing framework has been in maintenance mode for several years now and its maintainers are suggesting to move to the newer framework `nose2`, but the Jupyter Notebook maintainers don't seem to have any plan to move forward.

For the **frontend** code testing, CasperJS is used. It is a Javascript testing framework built with JS itself and leveraging PhantomJS. In the recent issue #2243, the developers discuss the problems related to the Javascript tests. Apparently, race conditions and the asynchronous nature of Javascript make the tests fail at times. Developers are investigating a fix for this.

Continuous Integration Services

The project uses the integration offered by Github to automate building and testing of the pull requests as they are opened or updated. In particular, three CI services are used: *Travis CI*, *AppVeyor* and *Codecov*.

Travis CI is used by the project to automate build and testing on Linux. The configuration, specified using the `.travis.yml` file, is such that:

- Testing is done using both Python 2.7 and 3.5.1
- *Both* the frontend and the backend are tested
- When testing is done, the Python coverage reports generated by Nosetests are uploaded to Codecov.

Since Travis CI does not yet support the Windows OS, **AppVeyor** is used for the testing on this platform. The configuration is such that only the backend is tested, using both Python 2.7 and 3.5, and code coverage is not checked.

Codecov is a hosted CI service that computes a coverage score of the unit tests. The Jupyter Notebook project uses Codecov to analyse the coverage of the Python backend tests at each pull request. The coverage is not checked for the frontend because the testing framework does not support the generation of coverage reports. The service is configured by means of the `codecov.yml` file [6] so that the automatic checking of the pull requests fails if a decrease in the coverage score greater than 10% is observed.

Functional Viewpoint

Rozanski and Woods [1] explain that the functional view "defines the architectural elements that deliver the function of the system being described" (p. 294). This view documents the key runtime functional elements, their responsibilities, interfaces and primary interactions. In the notebook project, different key functional elements are present that interact with internal and external elements during run-time. Interfaces like in `Java` are not present in `Python`. But these functional elements do behave like interfaces, as they have well-defined functions that can be called by other elements, to perform different actions. The functional view of our project is visualised on figure 5. The key elements and the interaction between them is described next.

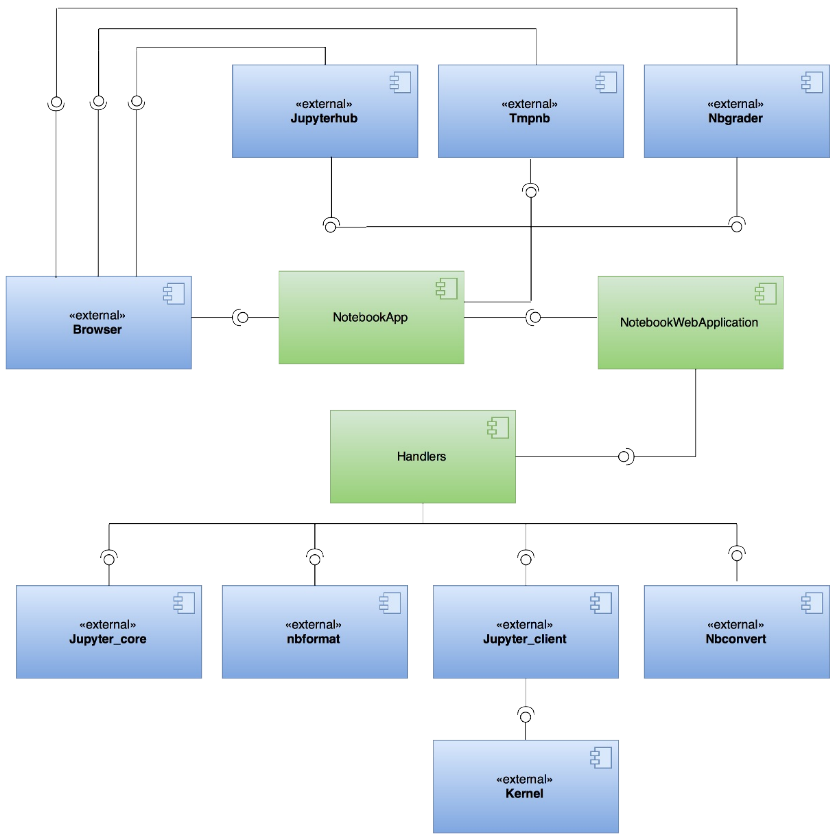


Figure 5: *UML diagram of the Functional View.*

The main entry point of the web application is an instance of the class `NotebookApp`. It sets up a Tornado based Notobook server that serves an HTML/Javascript client. It also launches an instance of the `NotebookWebApplication` class. The server delegates all the incoming requests to the `NotebookWebApplication` object. This class is a sub-class of `Tornado.Web.Application`, which is used to map the different types of incoming requests to the correct handlers.

The handlers use external elements to leverage existing functionalities from the other Jupyter projects.
`Jupyter_core` is used for basic functionalities like handling configuration files and filesystem locations. `Jupyter_client` provides APIs for working with kernels. Kernels are used to execute user code. Furthermore, `nbformat` provides APIs for working with notebook files, such as loading, reading and saving. Notebook uses `nbconvert` to export notebook files into various static formats such as PDF, HTML and LaTeX.

The `NotebookApp` can only be used by a single user with no login. On top of the notebook, there are other external elements which provide additional functionalities. First, `tmpnb` can launch a temporary Jupyter notebook server for multi-user use, without login. Second, `jupyterhub` can create a multi-user hub with login which manages multiple instances of the single-user `NotebookApp`. At last, `nbgrader` provides a toolbar extension which allows a teacher to make and grade assignments on the notebook.

Technical Debt

The technical debt concept is related to the extra development work that is introduced when solutions that are easy to implement in the short run are used instead of applying the optimal ones that will keep the project maintainable in the long run; it is not only related to the code, it can also be associated with the (lack of) documentation, or with the low testing coverage.

This section focuses on the following three types of technical debt: *code debt*, *testing debt* and *documentation debt*.

Code Debt

Coping with code debt as soon as possible is important because the size of the code, as it grows, will make the maintenance difficulty ever increasing. The identification of code debt has been done using both automated analysis tools and via manual inspection.

In relation to the code readability, the Jupyter project documentation states in the coding style section that the code should follow the PEP8 guidelines. To investigate the adherence of the code to the guidelines the `flake8` tool was used to lint the codebase, discovering numerous inconsistencies, including mis-indentation, usage of tabs, unused imports, multiple imports on the same line and lambda expressions assigned to variables, for a total of 2075 errors/warnings.

Deprecation warnings are another symptom of code debt, as they mean that the developers are not keeping the code up to date with the evolution of the ecosystem. An example of this kind of debt was discovered while running the automated backend test suite. In particular, the `decodestring() is a deprecated alias, use decodebytes()` deprecation warning was shown. As the Python documentation states, there is a new API to work with base 64 encoding (base64.b64decode) which is supposed to be used. A pull request (#2280) was submitted to the project repository to fix this issue.

Another way of identifying code debt is by manually searching for "TODO" and "FIXME" comments in the codebase. As of *13/03/2017* Jupyter Notebook had **21 "TODO"** and **10 "FIXME"** comments in the code. A TODO comment was found in the page.js file, referring to the removal of hard-coded selectors from the code. Hard-coded strings are generally considered a bad coding practice that seriously hinders the flexibility of the code. A pull request (#2279) that tackles this code debt refactoring the parameters to be dynamic was submitted. A FIXME comment was also found in the file handlers.py, mentioning that a certain functionality should have been implemented in the frontend rather than in the backend. An examination of the frontend code revealed that the functionality was already present there and thus pull request #2281 was submitted to remove the redundancy.

Testing Debt

For what concerns the **backend** testing, the coverage is reported by `nose` and analysed by CodeCov. The actual coverage on the master branch is 77.24%. Test coverage usefulness as a metric has been debated, and the general consensus is that it is mainly a tool to discover untested parts of the codebase, rather than being an assurance that the codebase is well tested. The source files with the lowest reported coverage were analysed, and in the process, it was discovered that the tests for a certain file were not run because of a missing requirement in Travis CI; pull request #2283 was opened to fix the issue resulting in a 1.09% increase of the overall coverage.

On the **frontend** side, test coverage is not analysed because the testing framework chosen cannot output coverage reports. It seems anyway that there is significant technical debt. For instance, a number of pull requests were lately accepted without requiring the unit tests, since the Javascript testing suite seems to have significant problems; issue #2243 was opened by the maintainers to track the status of the testing suite and possibly fix the problems. It also seems like not all the developers are up to date with the frontend testing. In addition to this, Travis CI was configured to use the `travis_retry` for the frontend tests, in order to repeat them 3 times in the case that they fail. This function is supposed to be used to deal with network timeouts during the requirements installation and is therefore misused in this context.

Documentation Debt

Developers need to write documentation of their source code, which may be of help for other developers to understand their code. More importantly, when they would not work anymore for a project, people coming after and new contributors should also be able to understand the system and its setup. In addition, documentation for the end user of the product is needed in order to understand how to download, install and use the product.

Jupyter Notebook documentation is lacking in many parts of the project. Specific examples are listed below:

- An example of bad documentation is observed in issue #603 where a new user tries to use the shortcuts within Notebook but runs into trouble because the functionality is not clearly explained. This is supposed to be fixed in release 5.0 as @pkgw and @ellisonbg advised.
- Another documentation debt is identified in the issue #1754 where user @dsblank wanted to contribute to the project and tried to install the development environment on Ubuntu but got a lot of errors. Eventually, he managed to fix the issue by looking at the setup of the `.travis.yml` file. The user notes that this is not explained in the documentation.

Finally, the biggest part of documentation debt was found from the first sight, by just looking at the title, which says *(source: old IPython wiki)* (IPython Development Guide) and the whole documentation is outdated. There is, in fact, a note that states:

> This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

This is a clear indication of debt that needs to be paid and may cause confusion about where the correct documentation is if such exists and where is it located. Improvements of documentation are always welcome by the core developers and they try to encourage contributors to help them with this.

Evolution Perspective

In this section, the evolution of the project is presented, along with the most important facts and changes part of its history.

The Past: IPython

The predecessor of Jupyter Notebook is IPython. It was born in 2001 as an afternoon hack by the hands of Fernando Perez [3], in the attempt to improve the Python REPL. It was a 250 lines Python script trying to mimic the Wolfram Mathematica prompt system to enable scientific computing with Python.

Over the years the need for a notebook-like front-end grew, and after multiple unsuccessful attempts, in 2011 the IPython Notebook was born. The key in building this fully working web-based notebook system was the coupling of the Tornado webserver for asynchronous WebSocket-based communication and ZeroMQ for the communication with the kernels.

Many features were added in the following years following the users' needs, and the codebase developed organically incorporating different components that were increasingly becoming distinct projects. Moreover, at this point many different languages were supported by the project, so the name *Interactive Python* was starting to feel odd.

The Present: Jupyter

For these reasons, in 2014 at the SciPy conference, the Jupyter project was announced, incorporating all the language-agnostic features of IPython, such as the notebook.

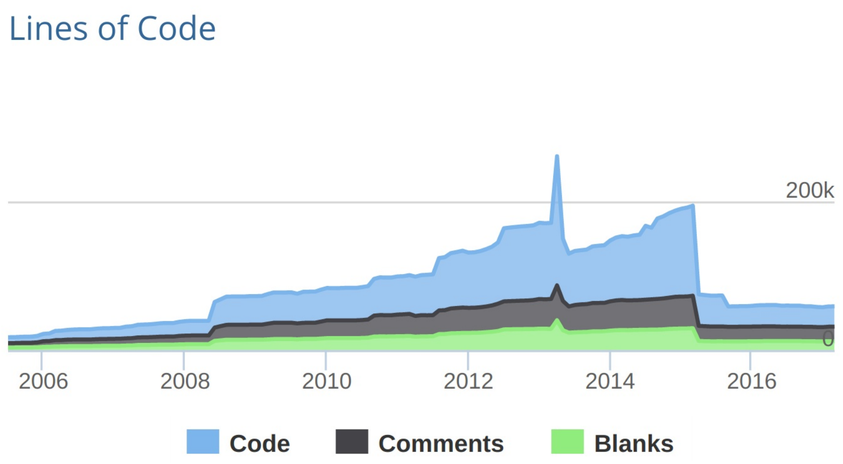


Figure 6 - *Evolution of LOC of the IPython project.* [4]

In 2015 the first phase of The Big Split™ was completed, splitting all the components into subprojects. As can be seen from Figure 6, a vast amount of code was removed from the IPython project and moved to separate repositories corresponding to the different subprojects, such as *traitlets*, *nbformat* and the *notebook* itself. IPython continues to live to this day as a kernel for Jupyter and as the interactive shell environment.

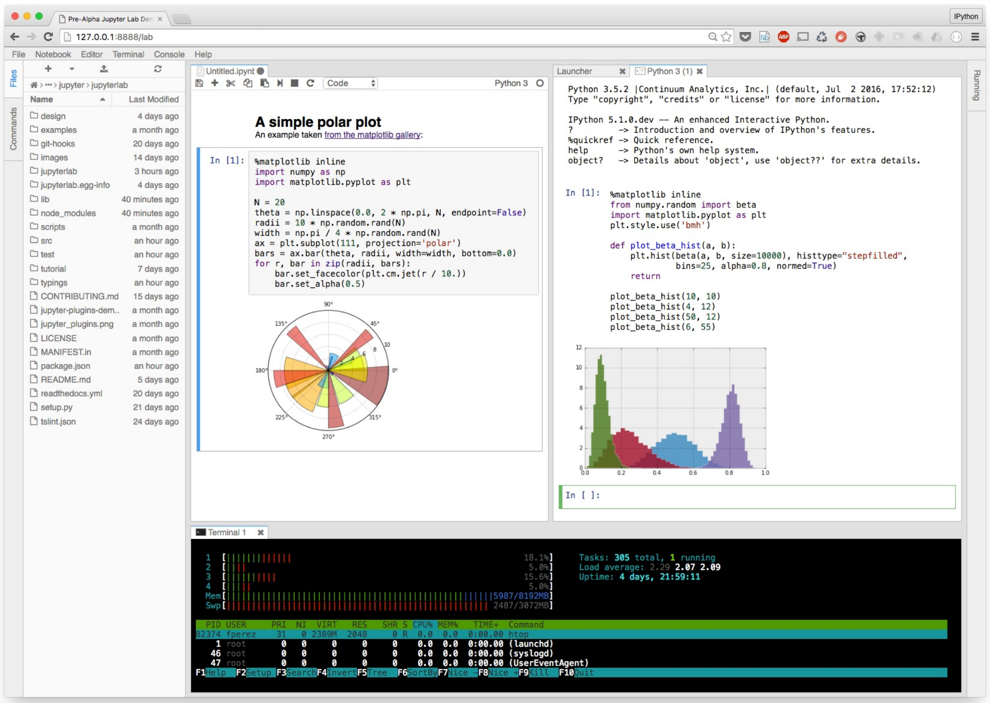
The Jupyter project gained more popularity, not only among data scientist but also among software engineers. At this point, Jupyter Notebook was not only about the notebook experience because it also shipped with a web-based terminal, text editor, and file browser. All these components were not blended together and the user community started expressing the need for a more integrated experience.

The Future: JupyterLab

At the SciPy 2016 conference, the JupyterLab project was announced. It was described as the natural evolution of the Jupyter Notebook interface.

The codebase of the Notebook was showing its age and it was becoming more and more difficult to extend. The cost of maintaining the old codebase and implementing new features on top of it was ever increasing.

The developers incorporated in this new project all the lessons that they learned from the usage patterns of the Notebook, to build a robust and clean foundation for a flexible interactive computing experience and an improved user interface.



Name	Last Modified
design	4 days ago
examples	a month ago
git-hooks	20 days ago
images	14 days ago
jupyterlab	3 hours ago
jupyterlab.egg-info	4 days ago
lib	40 minutes ago
node_modules	40 minutes ago
scripts	a month ago
src	an hour ago
test	an hour ago
tutorial	7 days ago
typings	an hour ago
CONTRIBUTING.md	15 days ago
jupyter-plugins-dem..	a month ago
jupyter_plugins.png	a month ago
LICENSE	a month ago
MANIFEST.in	a month ago
package.json	an hour ago
README.md	5 days ago
readthedocs.yml	20 days ago
setup.py	21 days ago
tslint.json	24 days ago

A simple polar plot
An example taken from the matplotlib gallery:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        N = 20
        theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
        radii = 10 * np.random.rand(N)
        width = np.pi / 4 * np.random.rand(N)
        ax = plt.subplot(111, projection='polar')
        bars = ax.bar(theta, radii, width=width, bottom=0.0)
        for r, bar in zip(radii, bars):
            bar.set_facecolor(plt.cm.jet(r / 10.))
            bar.set_alpha(0.5)
```

```
Python 3.5.2 |Continuum Analytics, Inc.| (default, Jul  2 2016, 17:52:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0.dev -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.


In [1]: %matplotlib inline
        from numpy.random import beta
        import matplotlib.pyplot as plt
        plt.style.use('bmh')

        def plot_beta_hist(a, b):
            plt.hist(beta(a, b, size=10000), histtype="stepfilled",
                     bins=25, alpha=0.8, normed=True)
            return

        plot_beta_hist(10, 10)
        plot_beta_hist(4, 12)
        plot_beta_hist(50, 12)
        plot_beta_hist(6, 55)

In [ ]:
```

```
  1  [|||||||||||||                                             18.1%]   Tasks: 305 total, 1 running
  2  [||||                                                       5.0%]   Load average: 2.29 2.07 2.09
  3  [||||||||||                                                15.6%]   Uptime: 4 days, 21:59:11
  4  [||||                                                       5.0%]
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||5987/8192MB]
  Swp[|||||||||||||||||||||||||||||||||||||||||||||||  2487/3072MB]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
82374 fperez     31   0 2389M  2048     0 R  0.0  0.0  0:00.00 htop
    1 root        0   0     0     0     0    0.0  0.0  0:00.00 (launchd)
   46 root        0   0     0     0     0    0.0  0.0  0:00.00 (syslogd)
   47 root        0   0     0     0     0    0.0  0.0  0:00.00 (UserEventAgent)
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 7 - *JupyterLab User Interface.*

In figure 7 the user interface of the JupyterLab project, based on the PhosporJS framework, is shown. The division in tabs and panes allows using different components (file browser, terminal, notebook), that can be used only separately in the classic Notebook, at the same time.

As of April 2017, the project is still in the early preview phase, and it is not suitable for general usage yet. A series of phases are planned to enable a smooth transition from the classic Notebook to the new environment, as eventually JupyterLab will become the default UI and the classic notebook will only be available as a separate download.

Conclusion

Jupyter notebook is currently the most popular and widely used tool for students and data scientists to create and share documents containing live code, explanations and visualisations. Its popularity can be validated by the fact that Github natively supports the rendering of notebook files.

While going through the code, our team found five issues that could give problems in the long-run i.e. non-compliance to code style, hard-coded variables, duplicate code, deprecated methods and low test coverage. Pull requests were made to pay off these technical-debt-related issues and four of them got accepted. Two additional pull requests, related to bug fixes, were made and both were merged to master. The developers provided extensive feedback and suggestions to our pull-requests which was very helpful.

Even though the documentation of the project was lacking for the most part and the analysis of the project's architecture proved to be more challenging than we thought, our team managed to understand the inner workings of the system. We learned that real-world software does not always correspond to the ideal architectures that we discuss in theory, but it can nonetheless be successful. We also learned that *The code is the truth, but it is not the whole truth* (Grady Booch), and that the Jupyter ecosystem, including its community, plays a major part in the success of the project. We are excited to see what the future has in store for the Jupyter project.

If you are considering joining the Jupyter notebook community and make contributions, do not hesitate. The Jupyter community will welcome you with open arms. This chapter together with the contribution guidelines and documentation will guide you towards your contributions.

References

1. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. Jupyter documentation (http://jupyter.readthedocs.io/en/latest/architecture/how_jupyter_ipython_work.html)
3. The IPython notebook: a historical retrospective (http://blog.fperez.org/2012/01/ipython-notebook-historical.html)
4. IPython LOC Analysis (https://www.openhub.net/p/ipython/analyses/latest/languages_summary)

Apache Kafka

Marc Juchli, Daan Rennings, Ron Wierzchowski and Mateusz Garbacz

Delft University of Technology

Abstract

Apache Kafka is an open source streaming platform, which allows to publish, store and process streams of information. The software is widely used by companies such as LinkedIn and Spotify and has a large and active development community. This chapter aims at analyzing the software and its development from three viewpoints, i.e. the context view, the development view and the information view as defined by Rozanski and Woods [10]. We thereby try to create a knowledge base, for the current state of the project as well as its evolution.

We found that the Apache Kafka project has a rich environment of stakeholders with regard to its use and development, which even includes researchers. Furthermore we elaborate upon the module structure of Kafka and discuss the strong standardization of development, testing and release management practices of Kafka, that overall has a reasonable and consciously maintained amount of technical debt. Finally, we have concluded that Kafka is overall an example of a neat software project of large scale. However, a few improvements could still be implemented, especially with regard to architectural documentation.

Contents

1. Introduction

Streaming has become an essential way of sharing the data over the internet, which allows to view and process the content before it has been fully transmitted [8]. The application of streaming in most of the companies is done separately for each two platforms or modules, which causes a high number of dependencies and processes to maintain and supervise. Moreover, most of the programs focus on a stream delivery, which does not suffice the needs of large companies. A solution for these problems that includes additional functionality can be found in Apache Kafka.

Apache Kafka is an open source streaming platform, which allows to publish, store and process streams of information across multiple entities [8].
Moreover, these actions are executed in a distributed manner, which allows it to adhere to parallelism, fault tolerance and high scalability of the processes. Finally, Kafka is designed to bring together multiple sources of data as well as many destinations, making it simple to adapt as a broker for the data streaming.

This chapter aims at analyzing Kafka as a software project based on viewpoints and perspectives defined by Rozanski and Woods [10]. To this end we discuss the stakeholders surrounding Kafka and their power and interest in section 2. A more elaborate view of it's environment in terms of a context view is discussed in section 3. Afterwards, in section 4, we have a closer look at the architecture of, development practices adopted in and technical debt accumulated by Kafka. We elaborate upon the information viewpoint in section 5, discussing the structure, flow, ownership and guarantees with regard to information in Kafka. We furthermore discuss various views in a timely perspective in section 6. Finally, we conclude our findings in section 7.

2. Stakeholders

In this section we discuss the stakeholders involved in Apache Kafka, the official structure as an Apache software project and identify its two official stakeholders.

2.1 Kafka, an Apache Software Project

Since Kafka is a project of the Apache Software Foundation (ASF), the foundation holds the copyright on all Apache code including the code in the Kafka codebase [9]. In their own bylaws [1] and in accordance with 'the Apache Way' [4], Kafka distinguishes two project roles: the Committers and the Project Management Committee (PMC). The Committers are responsible for the project's technical management and have access to and responsibility for all of Kafka's source code repository. The PMC is responsible for the technical direction and oversight of the project and reports quarterly to the Board of Directors of ASF. All of the PMC members are also Committers.

2.2 Classifying Stakeholders

Rozanski and Woods have classified stakeholders into a framework of stakeholder classes [10]. Based on this, we present an overview of how the various stakeholders in Apache Kafka could be classified into these stakeholder groups below.

Class of stakeholders	Role(s) and concern(s)	Kafka stakeholder
Acquirers	Oversee the procurement of the system or product	PMC
Assessors	Oversee the system's conformance to standards and legal regulation	PMC
Communicators	Explain the system to other stakeholders via its documentation and training materials	PMC and Committers, but also developers and system administrators that use Kafka that communicate through the Apache Kafka website and documentation

Developers	Construct and deploy the system from specifications (or lead the teams that do this)	PMC (in the form of development managers), Committers and other GitHub contributors as core developers
Maintainers	Manage the evolution of the system once it is operational	PMC and Committers
Production Engineers	Design, deploy, and manage the hardware and software environments	Third party software organizations such as JIRA, Jenkins, Gradle, CheckStyle, ect.
Suppliers	Build and/or supply the hardware, software, or infrastructure on which the system will run	JVM, Scala, Apache Zookeeper, Clients
Support Staff	Provide support to users for the product or system when it is running	PMC and Committers, but also individual developers active in the Kafka's users mailing lists
System Administrators	Run the system once it has been deployed	System administrators at companies that use Kafka
Testers	Test the system to ensure that it is suitable for use	PMC and Committers vote for approval, Users also contribute to (context specific) testing and the creation of issues, as well as Developers in general
Users	Define the system's functionality and ultimately make use of it	Confluent and LinkedIn seem to play a major role in defining functionality, a multitude of (large) companies are regular users
Researchers	Work with or on the system from a research perspective, Apache Kafka is thereby a separate object of study or used in a larger system under study	An active community of researchers (Kafka is stated in over 800 papers that may also be PMC members, Committers or other developers
Competitors	Offer (a subset of) similar functionalities as Kafka	Traditional messaging platforms such as ZeroMQ and RabbitMQ, related modern messaging platforms such as Amazon Kinesis and Apache Flink

Table 1: Classes of stakeholders in Apache Kafka

2.3 Power versus Interest

We extend upon the previous classification of stakeholders to include the power and interest of the various stakeholders [10]. We do so by means of a power interest grid, of which the result is shown below. For an explanation of images that were not introduced yet but are depicted in this grid, we direct the reader to section 3.

The third party software used by Kafka can be divided in three groups, all in the lower left quadrant of the power interest grid. The group having the least power and interest consists of GitHub, JIRA, CheckStyle, FindBugs and Jenkins. Less easy to replace are Gradle, Ducktape and Zookeeper, as they have less alternatives that could be used with regard to Kafka's usage. The third group is formed by the programming languages, again this group has no interest in Kafka, but does have more power as the whole codebase consists of them, i.e. they are hardly replaceable.

More interest - but hardly any power - resides with companies such as Netflix that use Kafka, but do not officially contribute. Developers on products such as HortonWorks and Cloudera - which include Kafka in their core - and researchers that study Apache Kafka itself or use it in their research generally contribute more and therefore have a larger amount of power. We define competitors of Kafka to have a similar amount of interest, but somewhat more power as they can steer the development process of Kafka (e.g. by adopting new features).

In the quadrant containing the most interest and power, we can furthermore see a growth of both from simple developers to Committers to PMC members to Kafka's Vice President Jun Rao. As early explained, the ASF holds a significant amount of power, although it's interest may be less compared to e.g. LinkedIn and Confluent from which Kafka originated and still gains much steering and development contributions.

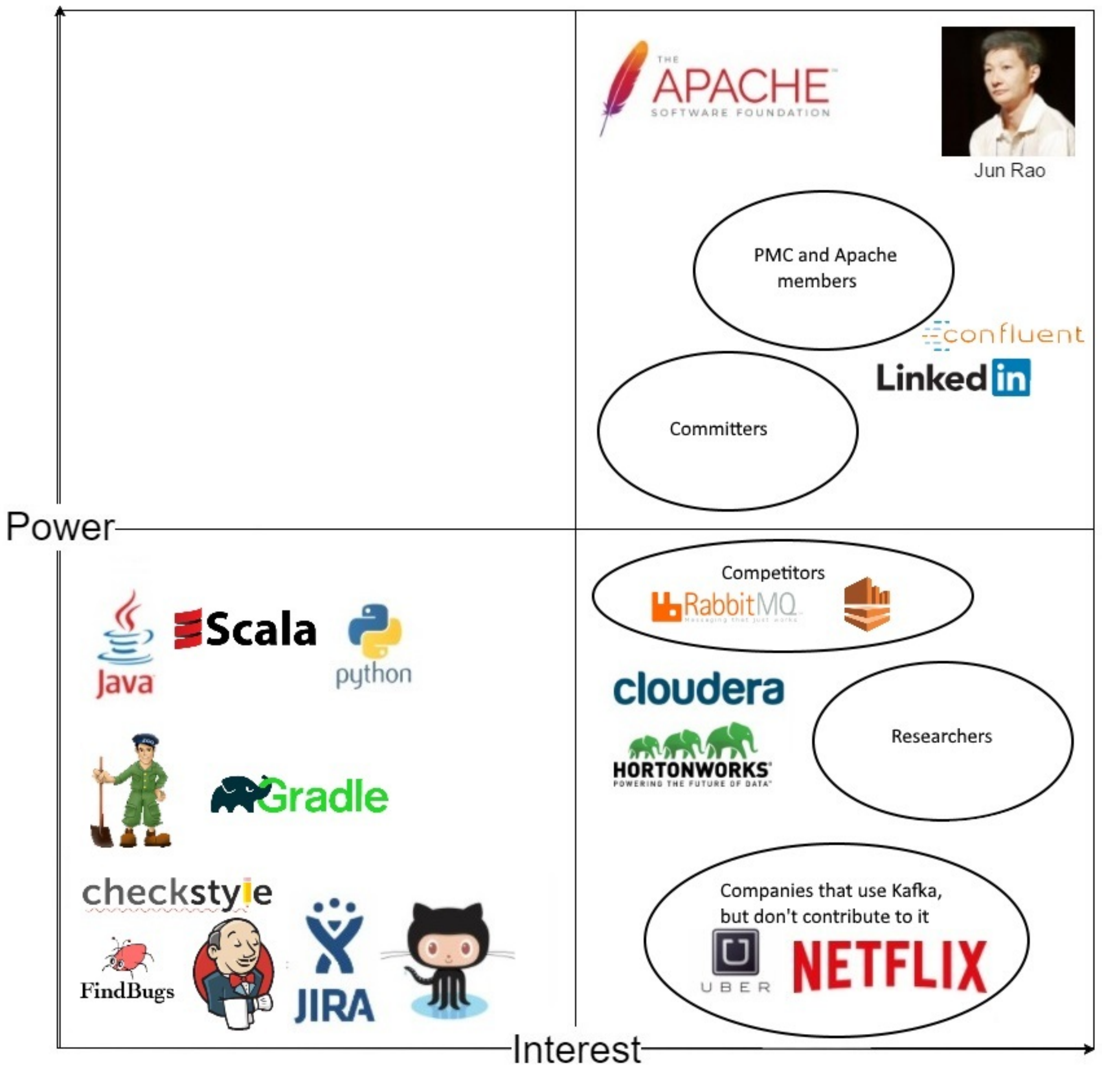


Figure 1: A Power Interest Grid of stakeholders in Apache Kafka

3. Context View

This section describes the system's scope and responsibilities as well as relations with its environment consisting of users and external entities.

3.1 System Scope and Responsibilities

The system scope and responsibilities define what the system should do in order to fulfil its objective [10]. These include the following [20]:

- Enabling users to publish and subscribe to streams of records. In this respect it is similar to a message queue or enterprise messaging system.
- Enabling users to store streams of records in a fault-tolerant way.
- Enabling users to process streams of records as they occur.
- Enabling users to build real-time streaming data pipelines that reliably get data between

systems or applications
- Enabling users to build real-time streaming applications that transform or react to the streams of data

3.2 External Entities

There are several external entities surrounding Kafka's environment. Here, we first elaborate upon them and display them in an overview afterwards.

- The organisation that has started the project: LinkedIn. [21]
- Owner of the project: Apache Software Foundation.
- The license of the project: Apache Licence 2.0.
- Programming languages used in the project: Java (63.7%), Scala (31.3%), Python (4.2%), Other (0.8%)
- The services used for development and issue tracking: GitHub and JIRA.
- The test entities: EasyMock and JUnit software used to develop test cases. The other test entities through which automated testing is applied are Ducktape and Jenkins.
- The code quality assurance tools: Checkstyle and FindBugs for codestyle and bugs respectively.
- The integration software: ASFBot, which integrates GitHub with JIRA.
- Users: companies that use the Kafka platform in their own processes e.g. Spotify, Uber and Netflix, which may also actively contribute to the project, e.g. LinkedIn and Cloudera.
- The development community: consisting of PMC members, Committers, Users, Researchers and other GitHub contributors
- The entities upon which the software is dependent: e.g. Zookeeper and Log4J. The complete list of dependencies can be found at the build.gradle file.

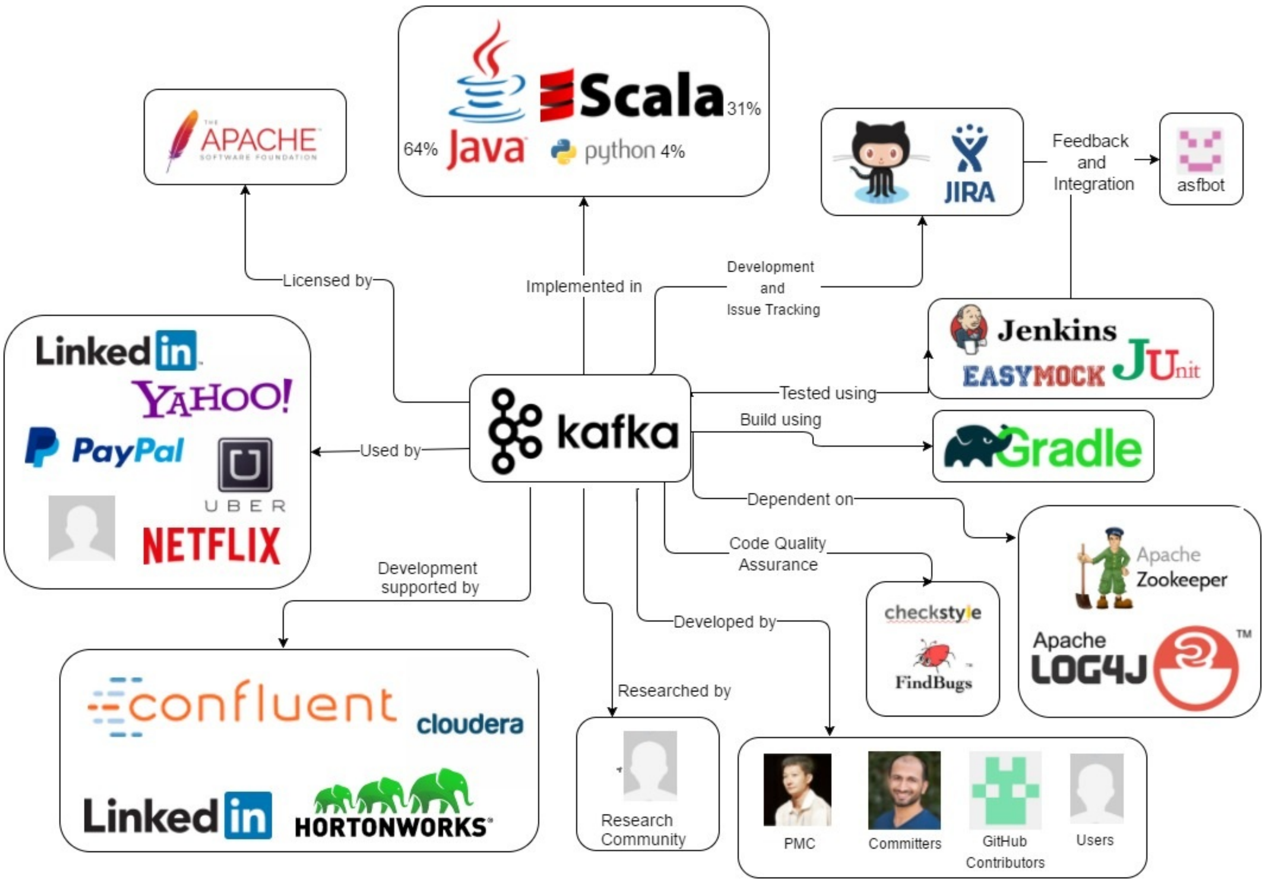


Figure 2: Context view of Apache Kafka

4. Development View

"The development viewpoint describes the architecture that supports the software development process" [10]. It can be applied to all systems that have significant software development involved in their creation. We will present an overview of the modules that are part of Kafka, elaborate upon the adopted standardization of design through the as-designed and as-implemented development views and discuss the key architectural styles we identified. We will furthermore discuss the standardization of testing and the release management and how the various actors are involved in these processes.

4.1 Module Overview

Over the past years Apache Kafka has seen a rapid growth in terms of the components and functionalities it offers. As further described in the Evolution section the codebase of the project has grown equally big. The following section therefore aims to display how the source code is organized currently (v0.10.2.0) from a high level perspective.

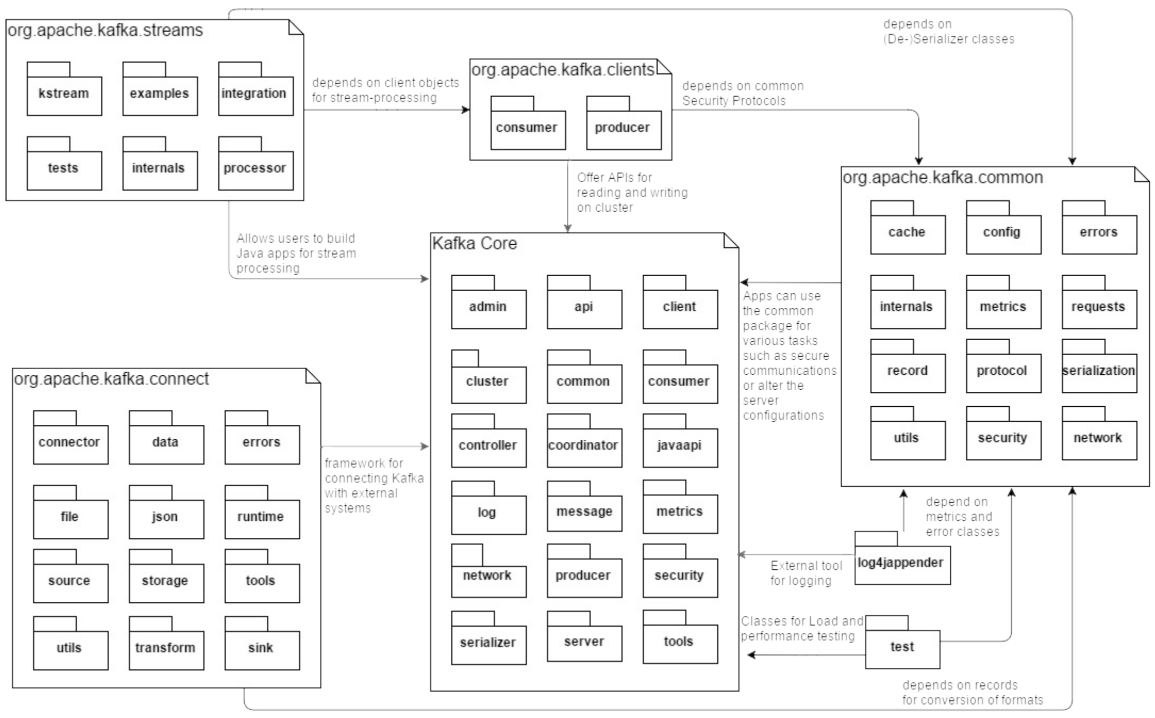

org.apache.kafka.streams
kstream
examples
integration
tests
internals
processor
depends on client objects for stream-processing
org.apache.kafka.clients
consumer
producer
depends on common Security Protocols
depends on (De-)Serializer classes
Offer APIs for reading and writing on cluster
Allows users to build Java apps for stream processing
Kafka Core
admin
api
client
cluster
common
consumer
controller
coordinator
javaapi
log
message
metrics
network
producer
security
serializer
server
tools
org.apache.kafka.common
cache
config
errors
internals
metrics
requests
record
protocol
serialization
utils
security
network
Apps can use the common package for various tasks such as secure communications or alter the server configurations
org.apache.kafka.connect
connector
data
errors
file
json
runtime
source
storage
tools
utils
transform
sink
framework for connecting Kafka with external systems
depend on metrics and error classes
log4jappender
External tool for logging
Classes for Load and performance testing
test
depends on records for conversion of formats

Figure 3: Code Module overview. The arrows towards the Core module show which functionalities each module provides. Furthermore dependencies between these modules are displayed.

As mentioned before, the project is implemented in Java as well as Scala. However, most functionalities of the message broker core is written in Scala, whereas packages built on top of that, such as clients or streaming, are written in Java. Through this separation the **Core** component can be trusted with the characteristics that the Kafka project values most, i.e. handling data feeds with high-throughput and low-latency while ensuring persistency and fault tolerance across clusters. The Scala code is grouped under kafka/core and displayed in the center of Figure 3. As shown in that figure, all other major components extend the Kafka core with further functionalities and interfaces.

- `org.apache.kafka.streams` : is a library for building scalable stream processing applications on top of Apache Kafka.
- `org.apache.kafka.connect` : is a framework for reliably streaming data between Apache Kafka and other data systems and allows to use already existing connector implementations for common data sources. Through these interfaces it possible to deliver data from Kafka topics into secondary indexes like Elasticsearch or into batch systems such as Hadoop for offline analysis.
- `org.apache.kafka.clients` This module provides the two main interfaces through which applications access data in the cluster, the Producer API and the [Consumer API].
- `org.apache.kafka.common` The Common module provides essential tools that are used by the other modules as well as user apps, e.g. for De-Serialization, performance

metrics, error handling, communication protocols and server configuration.

4.2 Standards in Kafka

In such a complex system it is important to standardize the design of key aspects in order to ensure the maintainability and reliability of the resulting product.

4.2.1 Standardization of Design

In this section we analyze how Kafka has standardized the way design decisions are made.

Kafka has standardized the design process by requiring a Kafka Improvement Proposal (KIP) for changing any public API or major feature, as defined in their Wiki [17]. A KIP is created as a Wiki entry by the person proposing the changes and then discussed in the mailing list. A KIP should motivate the proposed changes and discuss alternative solutions. The goal is that the integrators as well as the community can ensure that changes have the correct impact on the project. General coding guidelines are also given on their webpage [19]. Lastly, it should be mentioned that many software projects define the use of common design patterns in development. Also in Kafka these patterns are employed though their use not fully standardized, rather this is guided by the use of KIPs and coding guidelines.

4.2.2 Standardization of Testing

Testing in Kafka is highly standardized and automated. There are two main tools used by the developers. The first is Ducktape tool, a distributed testing framework, which provides the runner, result reporter and additional utilities. Furthermore, the Jenkins system has been incorporated to enable automated tests (see example) in the cloud for different builds of the system. The main difference in use of these two systems is that Jenkins is mainly used for the integration of pull requests, while Ducktape tests may be triggered by a developer at any time, locally. However, it requires bringing up a cluster of virtual machines using 10G RAM.

Considering test writing, with every feature introduced to the system, there are test cases added as well, to make sure that only parts of code that well covered by tests are merged into the system, see example pull request. The official guidelines for making contributions that match the test standards can be found in the documentation. Moreover, the newly introduced test cases are merged to the test cases with most pull requests merged to the system.

The most significant test cases are run over night on a regular basis to ensure that the vital parts of the systems are not corrupted. Next to that, all the test cases are run with every pull request on GitHub. This is executed automatically using Jenkins.

Finally, performance tests are executed as well, to analyze common statistics, logs and server side metrics, as described on the website. These tests are automated and incorporated into frequently updated metrics that are monitored to make sure that the system is stable and efficient.

4.2.3 Release Management

Management of Apache Kafka releases is standardized even more than it's design and testing processes, as will be discussed in this section. On the Release Process page for Committers, which follows the ASF Release Creation Process, it is specified that a new release requires careful preparation regarding implementation and documentation, but also approval to become reality. The person that actually manages the release process is called the Release Manager (RM). The release process consists of five major steps in which the RM, but also the Kafka development community and of course the PMC play a part. Below you can find an overview of these steps that are discussed more elaborately on Kafka's release process webpage, as well as an overview of past and future release plans.

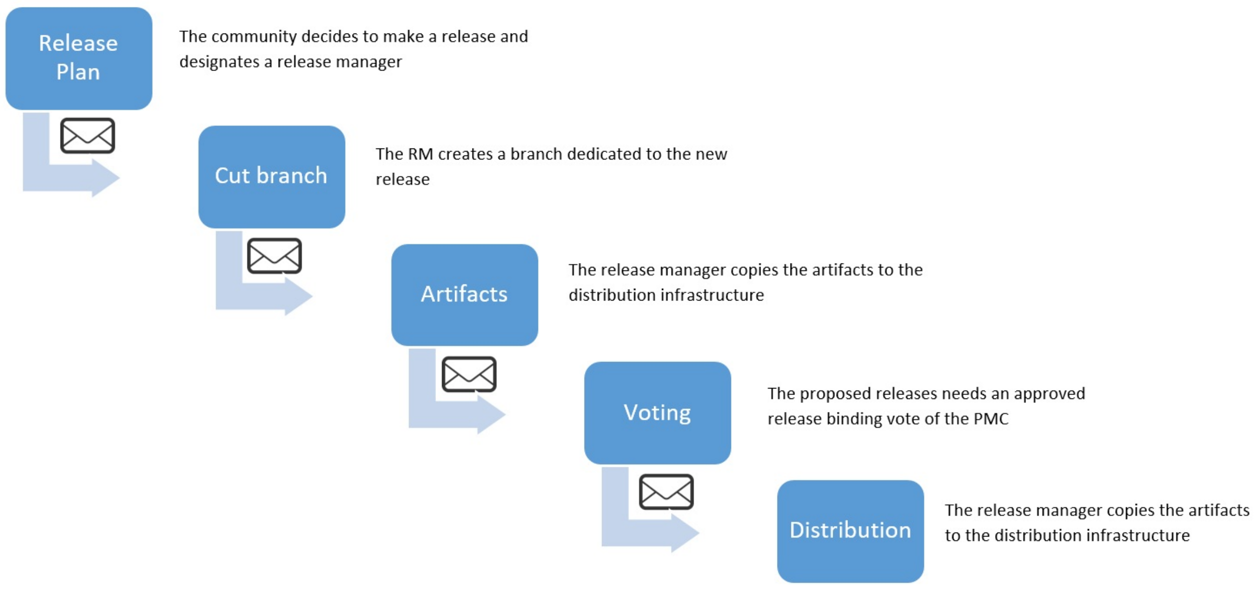

Release Plan
The community decides to make a release and designates a release manager
Cut branch
The RM creates a branch dedicated to the new release
Artifacts
The release manager copies the artifacts to the distribution infrastructure
Voting
The proposed releases needs an approved release binding vote of the PMC
Distribution
The release manager copies the artifacts to the distribution infrastructure

Figure 4: The Release Process in Apache Kafka

As an extension of the above, it was decided to move to a time-based release plan, as described on the dedicated Wiki page starting with Apache Kafka 0.10.1.0 (October 2016). The motivation to do so was to create a quicker and more predictable development cycle, with higher transparency. However, time pressure and specified time holes between releases form the downside of this approach. Nonetheless, it was decided that the benefits outweigh the cons of this process.

4.3 Technical Debt

In this section, we discuss the technical debt (a term first identified by Cunningham [6]) with regard to Apache Kafka. The subsections respectively describe the debt identified in the system with regard to defects, code-style, testing and documentation.

4.3.1 Defect Debt

In the Apache Kafka project, FindBugs is used to automatically detect defects in the form of bugs. However, for a truly critic technical debt collector, it may be good to know that Kafka does not let FindBugs check for all types of bugs, as can be seen in the exclude XML file. Though, we found these omissions not to have a significant impact on the judgement of FindBugs, although the reason for exclusion is not stated explicitly. In our FindBugs runs, we have found a considerable small relative amount of bugs in the FindBugs reports (less than 100 High Priority and less than 3000 Medium Priority Warnings in over 100000 analyzed lines of source code). The parts of code with highest defect debt are related to the core functionality of Kafka, while, minority of them to the external entities. However, since these warnings are well-visible and well-maintained by the Kafka development community (mainly through fixes labelled as "MINOR"), we conclude that the technical debt with regard to defects is pretty low.

4.3.2 Codestyle Debt

Similar to the adoption of FindBugs for defects, Kafka has adopted CheckStyle to control the style of code throughout its codebase. Although we found some severe violations around (e.g. a cyclomatic complexity of 35), we have concluded that Apache Kafka does not have much debt with regard to the code-style. However, we have to state that Kafka both has relaxed some of the default values of CheckStyles' parameters (as specified in the checkstyle XML file) and instruments CheckStyle to suppress some files (see the suppressions XML file). On the other hand, the relaxed parameters are within an appropriate range with regard to the default values. The amount of debt is again mainly accumulated for the core of the system, while for the external packages it is much less visible.

4.3.3 Testing Debt

Testing debt of Apache Kafka was mainly investigated in terms of test coverage based on the JaCoCo Java code coverage library. Firstly, we have analyzed the test coverage for different modules and concluded that some of them accumulated testing debt. One of the main issues when it comes to the test coverage is connected to the streams module, which is a major part of the system. This was mainly caused by its continuous development in the previous releases 0.10.1.0 and 0.10.2.0. This debt is being paid by extending the tests for the related classes. Another module with a low test coverage, yet, much lower importance for the project is log4j-appender. Therefore, we conclude that the developers put much higher focus in having low testing debt in terms of the Kafka core functionalities, while the external libraries are much less tested and have accumulated quite a significant amount of debt.

4.3.4 Documentation Debt

Even though there are various, spreaded knowledge sources of the project, there is no clear document describing the architectural design of the system, which constantly changes. Despite the fact that it would be extremely hard to maintain such documentation when keeping in mind how often new features are introduced, the project currently highly discourages an understanding of the design, which is in our view accumulated documentation debt.

5. Information View

The Information view can be used to answer, at an architectural level, questions about how the system will store, manipulate, manage, and distribute information. This section therefore aims to provide an overview over the key architectural concepts that govern the information structure, flow and ownership of Kafka.

Information Structure and Flow

As explained in Rozanski and Woods [10], an architect's challenge is to focus on the right aspects of information structure and to leave the details to the data modelers and data designers. For this purpose, as described in their documentation [11], Kafka has introduced a core abstraction for a collection of records — the topic.

Topics

Kafka categorizes data feeds by topics. It is the conceptual unit, which all reads and writes go through. A log is a simple, append-only data structure which contains a sequence of ordered records, whereas each entry is assigned to a unique number called offset. Thanks to the strict ordering inside a log, the record offset can be used as a timestamp where a log gets decoupled from any time system. Since records are immutable, Kafka is able to serve multiple users to read at any offset simultaneously. Similarly, it allows multiple producers to write to a topic at the same time, by simply appending the records to the logs. [11]

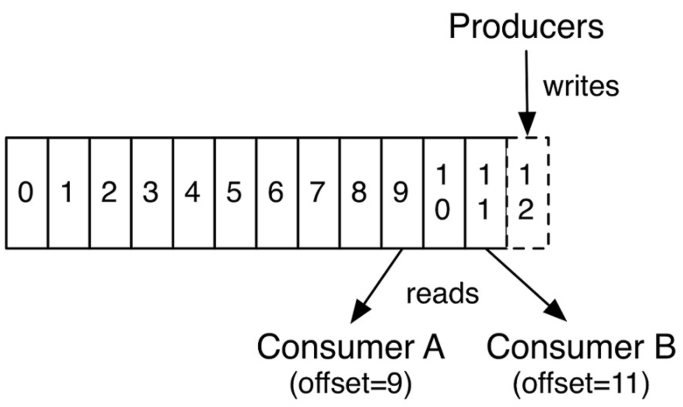


Figure 5: The log structure, showing simultaneous reads. [11]

Partitioning

In order to improve scalability and fault tolerance, Kafka divides topics into multiple partitions. This not only allows to move replicas across machines to guarantee fault tolerance, but it is also a way to parallelize the consumption of messages. Each client – be it on consumer or producer side – will split the burden of processing the topic between themselves, such that one member will only be concerned with messages in the partition it is assigned to. Thus, the throughput of the system will scale linearly with the Kafka cluster size. [11]

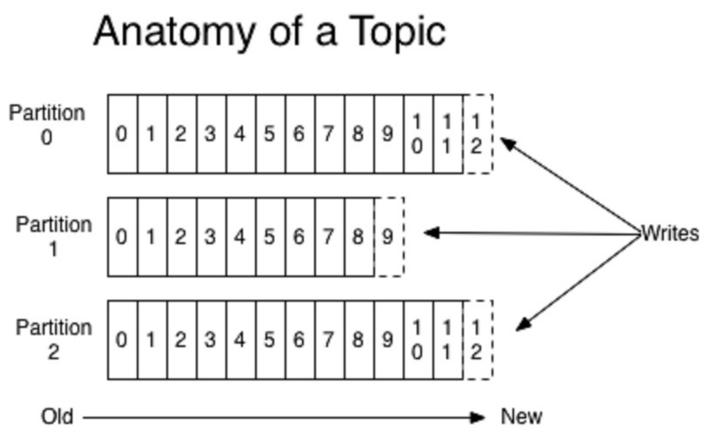


Figure 6: Partitioned log structure [11]

Information Ownership

As argued by Rozanski and Woods [10], it especially important to be able to reason about the synchronicity and recency of records when data is physically distributed across multiple data stores and accessed in different ways. We therefore analyze the theoretical guarantees given by Apache Kafka.

Distribution

The partitions of the log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance.

Guarantees

As a resume from the descriptions above we list guarantees of Kafka regarding message delivery, fault tolerance and ordering.

Delivery Model: Kafka guarantees *at-least-once* delivery by default and allows the user to implement *at-most-once* delivery by disabling retries on the producer and committing its offset prior to processing a batch of messages.

Exactly-once delivery requires co-operation with the destination storage system, but Kafka provides the offset which makes implementing this straight-forward [2].

Fault Tolerance: It is guaranteed that any successfully published message will not be lost and can be consumed. Furthermore, a topic with replication factor N will tolerate N-1 server failures without losing any messages.

However there is no guarantee for the producer that a message is fully published in case of a network error [2].

Message Ordering: The use of a log with sequential numbers for each entry (offset) guarantees that messages from a producer to a topic partition will be appended in the order they are sent. Subsequently, consumers see messages in the order they are stored in the log [2].

Replication

To satisfy the guarantees (see above), Apache Kafka supports replication on log partitions level. Replication of topics is executed across a configurable number of other Kafka servers (a.k.a. nodes), whereas each topic has an associated leader node. Producers send data directly to the leader node. A leader node can have zero or more follower nodes which are responsible for replicating the entries of the active log. An incoming message needs to be replicated by every in-sync follower before any other consumer can consume this message. A fully replicated message is therefore considered as *committed* [11][3].

6. Evolution and Outlook

Apache Kafka was initially developed at LinkedIn and subsequently released as an open source project under the umbrella of the Apache Software Foundation. The following sections will put Kafka in an evolutional perspective [10] more elaborately.

6.1 History of Kafka

At some point in the history of LinkedIn, the system landscape was overwhelmed by point-to-point pipelines, each delivering data to a single destination where data was required. Messages were written to aggregated files and then copied to ETL servers and further loaded into data warehousing and batch processing clusters. As a result, the company faced inevitable delays in adding new types of activity data. But integrating new features into the existing system landscape was not the only hassle. The detection time of operational problems noticeably increased over time as a result of ever-increasing pressure on the latency of the data warehouse processes. At this point, stream processing seemed to be the answer to those problems. However, providing continuous data with the current system architecture was not feasible at that point anymore – due to complexity – and thus, the need for a central platform that can provide continuous- and batch data was born. [11]

First attempts towards a piece of infrastructure that can serve stream and batch processing systems were made by consulting classical message broker applications such as ActiveMQ. During tests under full production load they ran into several significant problems as the queue backed up beyond what could be kept in memory. Further difficulties were faced with ActiveMQ's built in persistence mechanism that lead to very long restart times. According to LinkedIn, it would have been possible to provide enough buffer to keep the ActiveMQ brokers above water but would have required hundreds of servers to process a subset of activity data. Eventually, the engineers decided to a custom tailored messaging infrastructure, targeting high-volume, scale-out deployment that can serve batch- and stream processing systems. [13]

In early 2011 LinkedIn open sourced Apache Kafka, and soon after, on 23 October 2012, the graduation from the Apache Incubator followed. The released state of the application allowed LinkedIn to migrate their entire set of point-to-point queues, reaching above the entire space of their system landscape, into a well structured central message hub which would allow to scale easily while attaching new systems that provide or demand data. [14]

In November 2014, the engineers who worked on Kafka at LinkedIn, Jun Rao, Jay Kreps and Neha Narkhede, co-founded a new company, Confluent Inc., with a focus on Kafka. The company focusses on building a streaming platform around Apache Kafka. Besides, Confluent is actively contributing to the open source project. [15] [16]

6.2 Evolution of Development Practices

When Kafka was open sourced, the project consisted of 163.181 lines of code. Prior the Apache graduation, a refactoring was applied which lead to a total of 140.864 lines of code. Since then, with the help of the open source community, the project increased by more than a factor of two, up to 298.375 lines of code, as of today (03.04.2017).

The project was initially written entirely in Scala. Over the course of the past few year, however, a great part of the components were re-implemented in Java 8. Likewise, newly introduced components such as "streaming" were implement in Java from the very beginning. Furthermore, Apache Kafka tries to avoid as much third party dependencies as possible. For example, the extensive use of Zookeeper early in the project, acting not only as a intermediary for the broker core but also for the clients, was abolished to a great extent.

6.3 Evolution of Testing

At the beginning of the Apache Kafka development, multiple separate stand-alone programs were created for testing of features. The more new features were introduced, the harder testing appeared to be, because some of these programs became obsolete over time: see this issue.

In 2015, there has been a turning point of the project. At that time a proposal for the Test practice improvement has been issued, which lead to unification and automation of testing. The Confluent’s Ducktape system has been selected as a system that best matches the needs of the project. Further, continuous integration with Jenkins system has been established for an effective testing, for instance for each Pull Request on GitHub.

7. Conclusions

In its early days, Kafka came as a solution for struggles with moving data at LinkedIn. Later on, it became an Apache project and currently it is widely adopted by a multitude of big companies. From a contextual point of view, it can be concluded that Kafka has a rich environment of stakeholders that use and/or contribute to Kafka. In this environment, we also identified the entities through which Kafka is developed, ranging from programming languages and testing tools, to automated feedback and integration bots. With regard to development, we have identified that documentation debt surrounding the architecture of Kafka from a medium level perspective has been accumulated. Apart from this technical debt, Kafka's codebase was found to have hardly any debt and a good quality. But Kafka is not just a good software project, it is actually one of the big players in the domain of message brokers.

References

1. Apache Kafka. Bylaws. https://cwiki.apache.org/confluence/display/KAFKA/Bylaws, 2015.
2. Apache Kafka Website: https://kafka.apache.org/documentation/#intro_guarantees, 2016.
3. Apache Kafka Wiki - Replication. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication, 2016.
4. Apache Software Foundation. Apache Corporate Governance Overview. http://www.apache.org/foundation/governance/, 2016.
5. Arie van Deursen and Rogier Slag (eds). Delft Students on Software Architecture. http://delftswa.github.io, 2015.
6. Cunningham, Ward. The WyCash portfolio management system, Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), 1992.
7. Graham, Dorothy; VAN VEENENDAAL, Erik; EVANS, Isabel. Foundations of software testing: ISTQB certification. Cengage Learning EMEA, 2008.
8. Jay Kreps and Neha Narkhede and Jun Rao. Kafka: A Distributed Messaging System for Log Processing, at NetDB workshop, 2011.
9. Maximilian Michels. An Introduction to Apache Software. https://maximilianmichels.com/2017/an-introduction-to-apache-software/, 2017.
10. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
11. Apache Kafka Documentation: https://kafka.apache.org/documentation/#intro_topics, 2016.
12. Agile Mike: https://agilemichaeldougherty.wordpress.com/2015/07/24/types-of-technical-debt/, 2015.
13. Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building linkedin's real-time activity data pipeline. IEEE Data Eng. Bull., 35(2):33–45, 2012.
14. The Log: What every software engineer should know about real-time data's unifying abstraction, LinkedIn Engineering Blog. https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying, 2013.
15. LinkedIn engineers spin out to launch 'Kafka' startup Confluent. http://fortune.com/2014/11/06/linkedin-kafka-confluent/, 2014.
16. Confluent Inc. - About. https://www.confluent.io/about/, 2016.
17. Apache Kafka Wiki - Kafka Improvement Proposal. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals, 2016.
18. Apache Kafka Wiki - Contributing Code Changes. https://cwiki.apache.org/confluence/display/KAFKA/Contributing+Code+Changes, 2016.

19. Apache Kafka Website - Code Guidelines: https://kafka.apache.org/coding-guide, 2016.
20. Kafka Documentation - Design (Motivation): https://kafka.apache.org/documentation/#majordesignelements, 2016.
21. How Kafka got started at LinkedIn: http://insidebigdata.com/2016/04/28/a-brief-history-of-kafka-linkedins-messaging-platform/, 2016.

Kibana

Abstract

Kibana is your window into the Elastic Stack. It is a dashboard for real-time visualization and analysis of your Elasticsearch data. You can also use it to configure and manage clusters, and other products that are part of the Elastic Stack. In this chapter, we give an overview of Kibana in terms of its software architecture. We provide different views on the system, including a stakeholder analysis, context view, development view, and deployment view. Furthermore, we have a look at the internationalization status, as well as at issues and integration. Hereafter, we discuss the current status of technical debt and its evolution over time. We conclude with an overview of our findings.

Introduction

With the amount of data on the web growing exponentially every year, organizations are pushing the boundaries of using this data to improve their products. In 2012, the people behind Elasticsearch and Apache Lucene founded a company called Elastic, setting forth a vision that search can solve a plethora of data problems.

Today, Elastic offers a wide range of products, better known by the community as the Elastic/ELK Stack. The heart of this stack has always been Elasticsearch, a distributed, RESTful search and analytics engine, which centrally stores data to be accessed by the rest of the Stack. Apart from Elasticsearch, the largest component of the Stack is Elastic's Kibana. Kibana (Figure 1 displays a snapshot) is considered the window into the Elastic

Stack. It lets you visualize your Elasticsearch data, as well as configure and manage clusters and other products that are part of the Stack. All this funtionality is offered to the user through an intuitive, easy to use, yet powerful web interface.

We, four TU Delft students from the Delft Students on Software Architecture group, have made an in-depth analysis of the Kibana system. By providing insight into different views, we hope to make meaningful contributions to the system and make the system more accessible to future contributors. We do so by first providing insight into Kibana's stakeholders, context view, development view, and deployment view. After that, we will elaborate on issues and integration handling, and on current technical debt in the Kibana system. Finally, we will conclude with an overview of our findings.

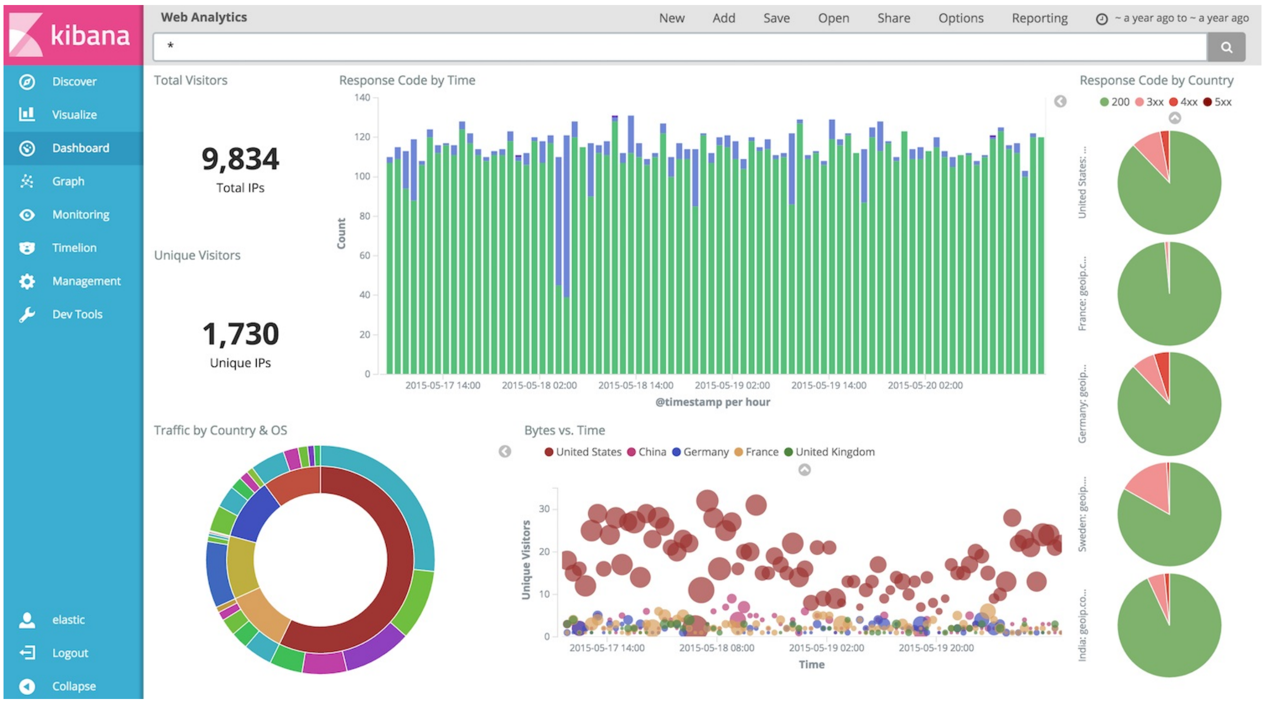


Figure 1 - A snapshot of a Kibana dashboard showing metrics.

Stakeholders

Stakeholders are the people that have an interest or concern in the product and organization. Stakeholders can have their effect on the product's objectives and policies. Rozanski and Woods [1] discuss different types of stakeholders in their book. In this section, we will first give an overview of the different stakeholders identified for Kibana. Hereafter, we will discuss how issues and integrations are handled. We will conclude this section with an overview of how power and interest in the system are divided amongst different stakeholders.

Overview

Acquirers: Elastic's founding and management team, as well as the investors and board, which partially overlap. Their interest lies in managing and growing the company and its assets.

Assessors: Baird Garrett, SVP of Legal and Robin Sharpe, VP of Operations. They make sure the company conforms to standards and legal regulation.

Communicators: Main responsibility for Jeff Yoshimura, VP of Worldwide Marketing. Elastic also has their own Education and Consulting Services department. Furthermore, elastic has a large community. Their responsibility lies in letting the world know Kibana exists.

Developers: Main responsibility of Kevin Kluge, VP of Engineering. Kibana is supported by an open source community and by Elastic's engineers. The top three recent Git contributors are:

1. @Spalger
2. @W33ble
3. @rashidkpc

Developers deploy the Kibana software and try to make it work as smooth as possible.

Maintainers Gaurav Gupta, VP of Products, and developers from both the open source community, as well as those who work at elastic. These propose new features and make sure that the system evolves for the better.

Suppliers: In order to make use of Kibana, one must have an *Elasticsearch* instance running. Elasticsearch and Kibana coexist, so it is in their best interest to work together as closely as possible.

Support Staff: Support is mainly provided by the community. Elastic Cloud users are provided with a service-level agreement based support, for which a team exists within the Elastic company. They try to solve all ad-hoc user problems.

System Administrators: Users either administrate their own systems when they host the Elastic stack themselves, or make use of the Elastic Cloud Service which has a dedicated system administration team. The administrators try to guarantee optimal uptime.

Testers: Developers write tests when implementing new functionality. After creating a Pull Request, Jenkins will execute the test suite and return the results which are then used by the integrators. Testers try to make sure software is bug-free before implementation

Users: Any user who runs an instance of the open source version. This group includes both the average user who is trying out new things, as well as large companies listed here. Users want to get the most out of Kibana's features.

Power/Interest Grid

The power/interest grid (see Figure 2) contains the main stakeholder categories. *Investors* and *Founders* evidently are the most important to manage, *maintainers* come second as they have a hand in the evolution of the product. *Suppliers* are relatively powerful because the system is dependent on their choices, but their interest is relatively low. We have to keep *Developers*, *Communicators*, and *Testers* very well informed to be able to perform their tasks. *Users* also have to be kept informed but have very limited power. Finally, *Support staff* and *System Administrators* can be monitored but do not require a lot of attention, as they have both low interest and power [2].

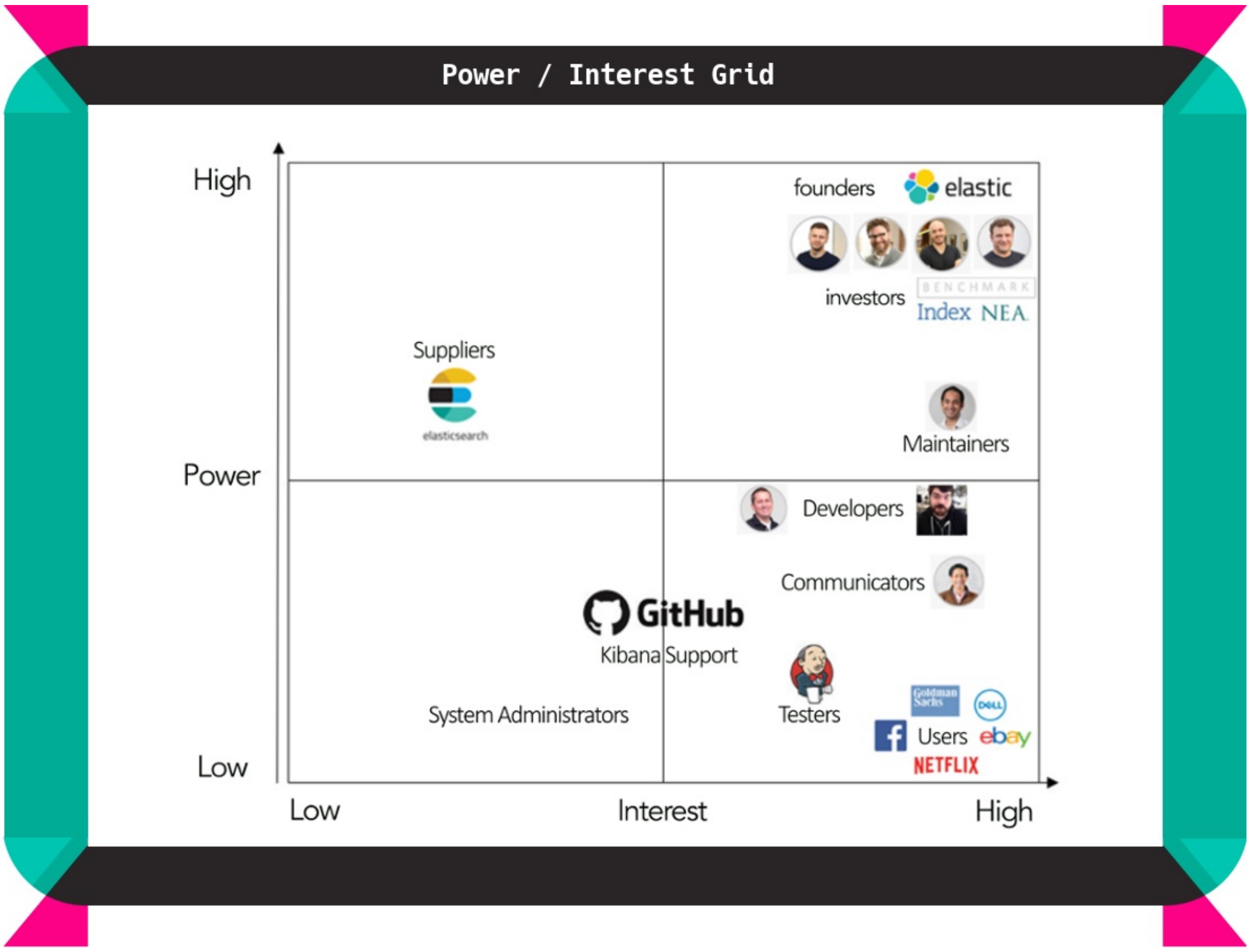


Figure 2 - Power/Interest Grid containing Stakeholder Prioritization

Context view

The context view [1] of a system defines the relationship and shows the interactions between the system and its environment. It is thus used to show the system's responsibilities and external entities. An overview of Kibana's context view is given in Figure 3.

Kibana's communities primarily interact on Github, which is used both as version control and issue tracker. Jenkins can be called from Pull Requests for continuous integration. The community also communicates via a dedicated IRC channel on Freenode.

Kibana itself is developed primarily in HTML, CSS, and JavaScript, and is tested using Karma and Mocha. Its dependencies are managed by package manager NPM and bundled using Webpack.

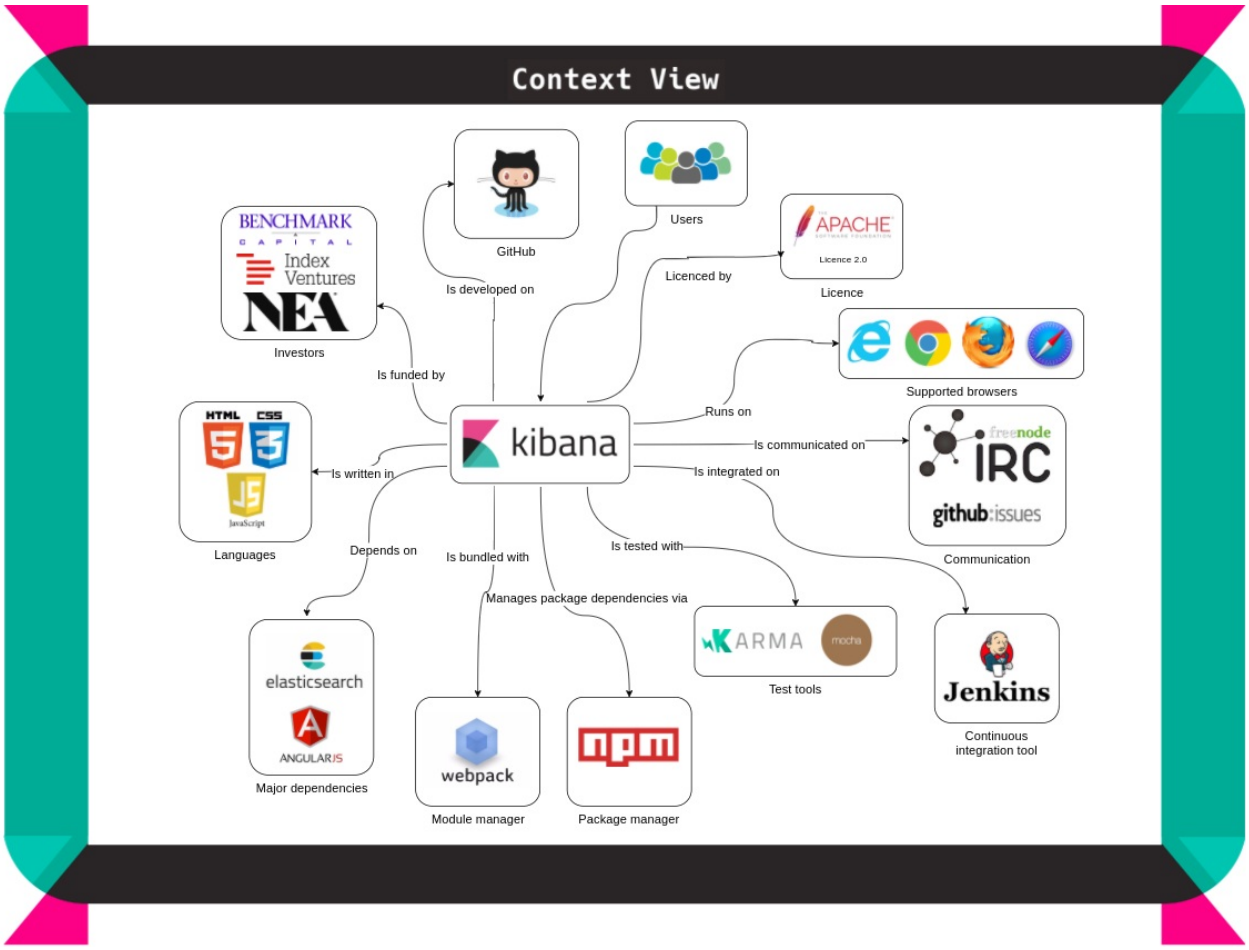


Figure 3 - Context view depicting interactions between Kibana and its environment

Development view

The development view of a system describes the architecture that supports the project's software development process. This section addresses the concerns of a developer such as module organization, standardization of design, common processes, and codeline organization. As Kibana has almost no technical documentation, it proved to be difficult to give a clear overview of the difference between as-designed and as implemented. Kibana does have a set of style guides for different types of individual components, and these will form the foundation for the analysis of the as-designed development view.

Furthermore, having no technical documentation can be an entry barrier for newcomers to join open source projects [2]. That is why issue #10710 proposes to address this. The issue description included the findings that are also discussed in this section.

Module organization

Kibana can be divided into three core modules. First, the UI module that defines the graphical user interface that users usually interact with. Second, the CLI module that contains the command line interface. Third, the Server module that serves data from the Elasticsearch clusters to the first two modules through an internal API.

Functionality in Kibana is implemented through plugins, these plugins contain the business logic and communicate with the CLI and UI module. This modular approach creates a more loosely coupled codebase. It also makes it easier for third-party plugin developers to add functionality to Kibana. A simplified high-level overview of the architecture is depicted in Figure 4.

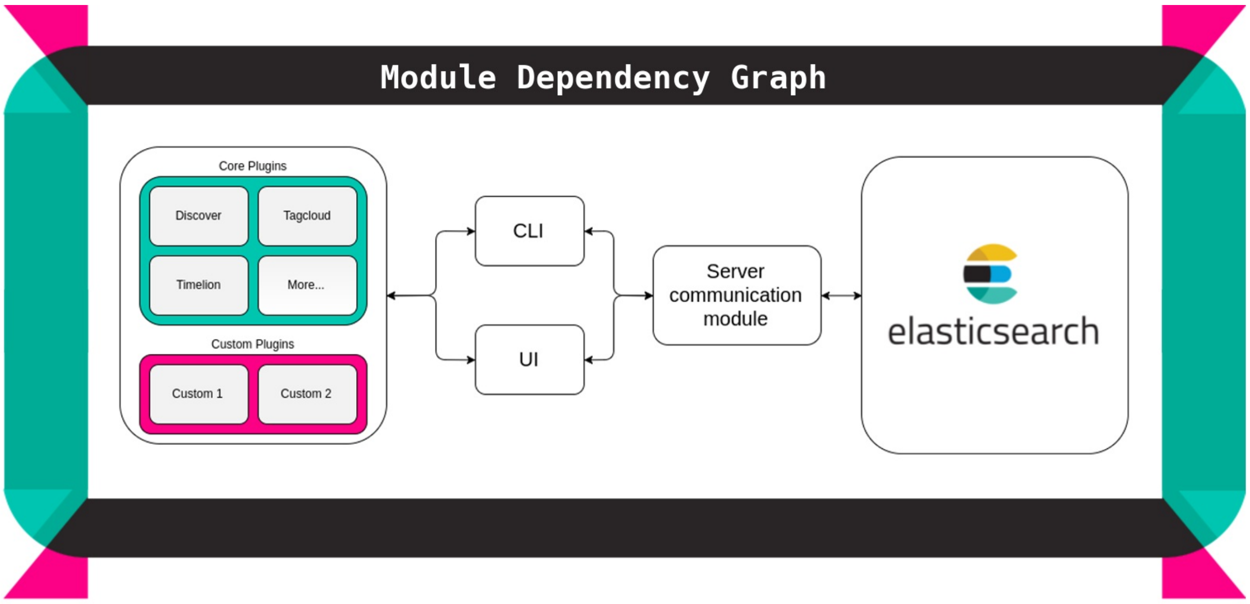


Figure 4 - High-level module dependency diagram

The Server module is connected with Elasticsearch and provides an internal API to be consumed by the UI and CLI module so that they can perform their task.

When the user accesses Kibana via the graphical user interface, the UI module loads all core plugins, which comprise the core functionalities of Kibana, and therefore should always be included. It also loads the Utils module, this is a collection of helper functions and objects that are grouped to prevent code duplication.

All Kibana modules depend on one or more external dependencies, which together form the final module in the diagram. These third-party packages are installed, updated, and deleted by package manager NPM.

Standardization of design

Open source projects such as Kibana are developed by a community. To keep Kibana's codebase consistent, the core developers (primarily employees of Elastic) composed a contribution guidelines document `CONTRIBUTIONS.md`. Main topics of the document are:

- How issues should be reported.
- How the community uses Git and GitHub.
- What to consider when contributing code.
- Information on the contributor license agreement (CLA).
- How to submit Pull Requests.
- Information about code reviewing.

They also created a set of style guides for different types of system components. These include both small styling conventions such as whether to use spaces or tabs, as well as larger conventions such as what the directory structure of a third-party plugin should look like. Essentially, the guides explain the standard design approaches for designing Kibana's core system elements. These guides are found under the `style-guides` directory.

Common processing

According to Rozanski and Woods [1], any large system would benefit from identifying common processes. Apart from the standardization described in the style guides mentioned in the previous section, other common design elements can be identified. This section will elaborate on how different common design elements are implemented in Kibana.

Termination and restart of operation This refers to conventions to be followed in case of program termination. Kibana itself does not store data but communicates with Elasticsearch through a RESTful API. Therefore, there is no need for Kibana to have complex termination procedures as termination of data related operations are handled by Elasticsearch.

Message logging and instrumentation: Logging is handled both in the command line interface, as well as in the browser. Many different approaches have been implemented for almost all of the different components in the `src` directory.

Use of third party libraries: Third party libraries are installed through NPM and contained in the `node_modules` directory. NPM keeps track of the different versions used for each of the libraries, and installs them accordingly.

Processing configuration parameters: The processing of configuration parameters is done by injecting the different configurations, managed in `kibana.yml`, into the different core plugins and UI components.

Security and cluster interaction: Kibana supports SSL encryption for both client requests and for requests the Kibana server sends to Elasticsearch. Furthermore, one can use X-Pack Security, a plugin used to control what Elasticsearch data users can access through Kibana. As long as developers use the internal APIs to connect UI components to the server, and make use of the earlier mentioned techniques, Kibana users can safely communicate with their Elasticsearch cluster in a production environment.

Testing: For main components of Kibana that are not part of the core plugins, tests are included in the `test` directory. This directory includes tests on fixtures, scripts, internal APIs, and more. These tests must pass in order for Kibana to run any of its base features. At the level of individual plugins, each plugin has its own `__test__` directory, containing plugin-specific tests.

Internal and external interfacing: Both the UI and CLI communicate directly and exclusively with the server to obtain their data through internal interfacing. The server is the only part of the system that interacts with the Elasticsearch cluster through external interfacing.

Codeline organization

According to Rozanski and Woods [1], the codeline organization of a system specifies how source code is stored in a directory structure, how the configuration is managed, how it is built and tested regularly, and how it is released as tested binaries for further testing and use.

Organization of the source code

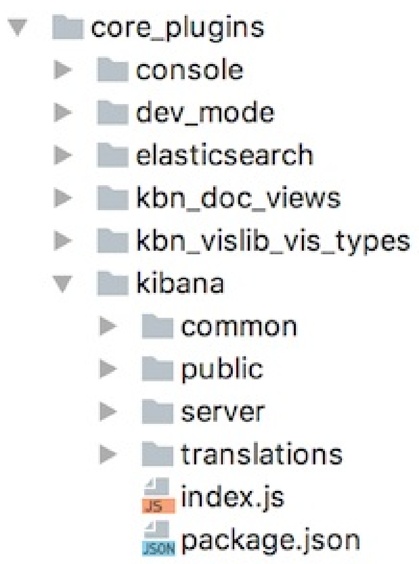
Under the root directory, both files and folders can be found. The files are either configuration files of third-party services and tools, or text files.

File	Purpose
`CONTRIBUTING.md` , `FAQ.md` , `README.md` , `STYLEGUIDE.md` , `LICENSE.md`	Several text files describing the repository, organizational conventions and the license
`.editorconfig`	Configuration file for editors and IDE's
`.eslintignore` , `eslintrc`	Configuration files for ESLint
`.node-version` , `.npmrc` , `package.json`	Configuration files for Node.js and NPM
`.travis.yml`	Configuration file for Jenkins (can also be used for Travis)
`Gruntfile.js`	Configuration file for Grunt
`.gitignore`	Configuration file for git describing which files should not be sent to the remote repository

Among the directories, there are some that are empty and will be filled during usage. Others contain auxiliary items such as styling guides, documentation, and Grunt task descriptions.

Directory	Purpose
`src`	The actual product
`test`	The tests
`utilities`	Code for visual regression
`config`	Kibana's configuration
`ui_framework`	UI components to build user interfaces
`docs`	The documentation of Kibana
`style_guides`	Style guides for the different languages and frameworks
`tasks`	The Grunt tasks
`data` , `optimize` , `plugins`	Files generated during use. These directories are initially empty.

The `src` directory contains a more in-depth analysis. As mentioned above, it contains the actual product. This directory is further divided into several subdirectories. Among these is the `core_plugins` directory that contains Kibana's core modules, each of which encapsulates a core functionality of Kibana. These components are designed and structured in the same way as third-party plugins to ensure that they are loosely coupled. They also contain their own `package.json` file.

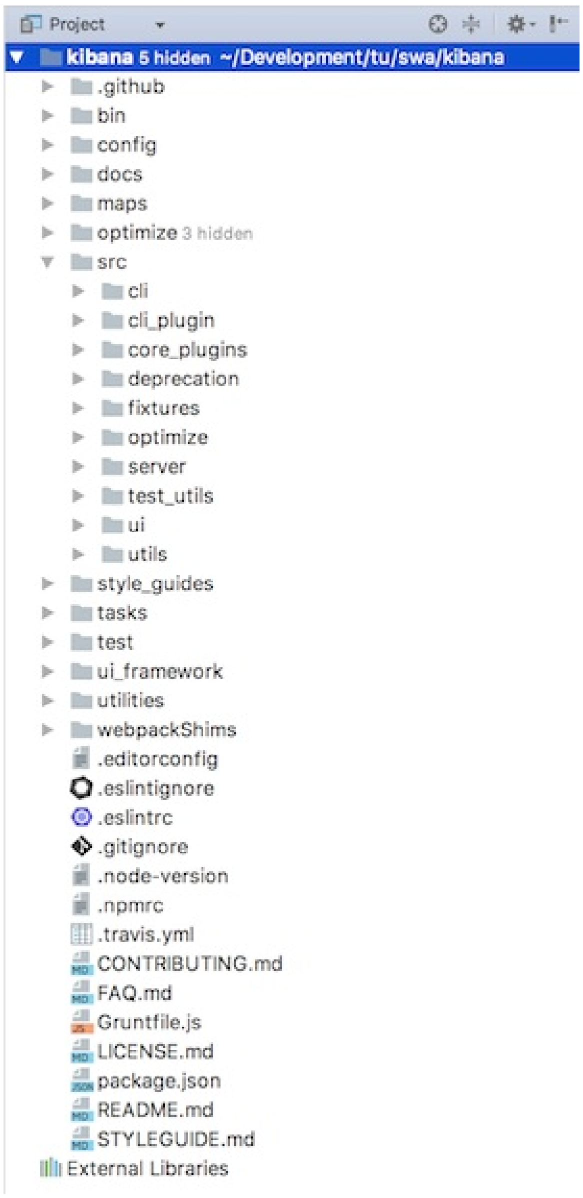


```
▼ core_plugins
    ▶ console
    ▶ dev_mode
    ▶ elasticsearch
    ▶ kbn_doc_views
    ▶ kbn_vislib_vis_types
    ▼ kibana
        ▶ common
        ▶ public
        ▶ server
        ▶ translations
          index.js
          package.json
```

Figure 5 - Plugin directory structure

Figure 5 shows the directory of a plugin inside Kibana. The roles of these elements are discussed briefly below.

File/directory	Purpose
`common/`	This folder is where code that is useful on both the client and the server belongs.
`public/`	This folder is where client-side code for your application is stored.
`server/`	This folder is where server code belongs, think of custom routes, data models or other code that should be executed on the server.
`translations/`	This is a plugin specific directory and is not included in the average plugin.
`index.js`	The entry point each plugin. This file is always loaded when a plugin is being accessed. This is where you define things like dependencies on other plugins and applications and configuration.
`package.json`	Contains information of the plugin, it's name and version.



```
Project
▼ kibana 5 hidden ~/Development/tu/swa/kibana
    ▶ .github
    ▶ bin
    ▶ config
    ▶ docs
    ▶ maps
    ▶ optimize 3 hidden
    ▼ src
        ▶ cli
        ▶ cli_plugin
        ▶ core_plugins
        ▶ deprecation
        ▶ fixtures
        ▶ optimize
        ▶ server
        ▶ test_utils
        ▶ ui
        ▶ utils
    ▶ style_guides
    ▶ tasks
    ▶ test
    ▶ ui_framework
    ▶ utilities
    ▶ webpackShims
      .editorconfig
      .eslintignore
      .eslintrc
      .gitignore
      .node-version
      .npmrc
      .travis.yml
      CONTRIBUTING.md
      FAQ.md
      Gruntfile.js
      LICENSE.md
      package.json
      README.md
      STYLEGUIDE.md
  External Libraries
```

Figure 6 - Directory structure of Kibana

Figure 6 shows an overview of the directory structure of Kibana. As mentioned above, the `core_plugins` directory contains all components that Kibana consists of.

Approach to building, integrating, testing and releasing

The build process of Kibana contains various subcomponents. These consist of downloading and including external dependencies such as Node.js and ReactJS, converting ES6 into regular JavaScript code with Babel, setting environment parameters to production settings, and readying the `README.md` to exclude the section on snapshot builds.

Kibana has a many-sided test process involving browser and UI testing, server testing, and visual regression testing. Plugins are also tested during this process. To achieve this, many tools are utilized such as Karma, Mocha and ChromeDriver. Furthermore, a tool developed by Elastic called ESVM is used to test with different versions of Elasticsearch.

The processes for building, integrating, testing and releasing are all automated so that they can be activated using a single action. For the build, test and release processes this is done using a task runner tool called Grunt.

Releasing into the test environment for integration is handled by Jenkins following the practice of Continuous Integration. For the production environment, the releases are created and listed using GitHub's Releases.

Configuration management

The configuration of Kibana is managed from `config/kibana.yml`. Examples of what can be configured are the port for Kibana's back-end server, credentials for Elasticsearch, and the SSL settings. Besides that, configuration files for third-party tools and services such as NPM, Jenkins and ESLint are found under the root directory.

Deployment view

According to Rozanski and Woods [1], the deployment view describes the environment into which the system will be deployed, including the dependencies the system has on its runtime environment.

At this point in time, Kibana supports Linux, Darwin, and Windows. Since Kibana runs on Node.js, the necessary Node.js binaries for these platforms are included as part of the product.

Other third-party software requirements are a result of Kibana's JavaScript nature:

- **NPM** manages Kibana's dependencies. It is written in JavaScript. Kibana 6.0.0 requires v3.10.10.
- **Node.js** is an open-source, cross-platform runtime environment for developing server-side web applications in JavaScript. Kibana 6.0.0 requires v6.9.5.

Finally, a modern browser is required. Kibana 6.0.0 supports Chrome, Firefox, Safari, and IE11+.

Internationalization

Internationalization means designing and developing Kibana in such a way, that it enables easy localization for target audiences that vary in language or culture. It is currently being implemented in Kibana. Issue #6515 discusses Kibana's internationalization roadmap. It was opened in March 2016, so implementation started recently. The internationalization of Kibana is being developed in four phases:

Phase 1: Implementing the internationalization engine in the form of a class called i18n (abbreviation for internationalization). The i18n engine should manage all locale translations; registering all translations and loading the correct locale when required.

Phase 2: Integrating the angular translate module with the i18n class. Furthermore, generating a translation plugin that localization engineers can easily use to create translations.

Phase 3: Adding translation identifiers and English translation strings for Kibana's AngularJS/ReactJS views.

Phase 4: Creating core language packs that are supported by Kibana, and allowing language packs to be contributed by outsiders.

Currently, phase 1 is finished and both phase 2 and phase 3 are work in progress. This is a nice video showcasing how to translate Kibana with the IBM globalization pipeline.

Issues and integration

This section discusses the workflow for issues and pull requests for the Kibana project. We conclude by mentioning the integrators and their challenges.

Issues

At the time of analysis, Kibana had 1397 open and 4637 closed issues. The issues are categorized with colored labels to indicate the type of issue the domains affected by the issue. The most important labels are:

- **Black labels** are used to indicate *priority*. `P1` for high priority issues and `P5` for issues that for example are highly niche or in opposition to the core goals of the Kibana team.
- **Blue labels** are used to indicate *difficulty to implement*. `low fruit` is used for easy issues, `high fruit` for more complex issues.
- **Grey labels** are used to indicate *what version of Kibana the issue belongs to*
- **Yellow labels**, starting with a colon, are used to indicate *what component the issue belongs to*. Examples of labels are `:Data Table`, `:Filters` and `:Heatmap`.

Pull Requests

While anyone can submit a pull request, most are created by Elastic employees who work on Kibana. Administrators that open pull requests can also merge them, as long as they are reviewed by at least one peer. While most are discussed on Github, sometimes it seems as if some of the pull requests are closed suddenly without discussion. This leads us to believe that these pull requests are discussed elsewhere, for instance in slack discussions. Some pull requests also refer to discussions on Slack. Contributors also make great use of 'Work In Progress' pull requests, which are pull requests that are not yet ready for merge, but are meant to be submitted unfinished, so that people can start to discuss the topic and collaborate on the code.

Integrators

Based on the analyzed issues and pull requests, we are able to determine the main integrators. The top three integrators are:

1. @Ppisljar
2. @Thomasneirynck
3. @Kobelb

Several factors affect their decisions. First of all, to trigger a review, the @Elasticmachine bot asks an admin to verify the patch. Then they use a test to find out if the commit author has signed the contributor license agreement (CLA). When the author has not yet signed, this is requested before merging. To test if the build succeeds, they ask Jenkins to test. For more difficult merges, they ask developer specialists to look over the pull request and ask for permission to merge. This process gives the integrators a robust framework to base their merging decisions on.

The main challenge for integrators is to keep the codebase bug-free and style compliant. Kibana is a popular project, so the integrators have to make sure that developers from all over the world create code that matches their desires. Matching random pull request with the company vision requires strict attention.

Technical debt

According to Techopedia, technical debt is *"a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution"*. The goal of identifying technical debt in a system is to improve code maintainability and to reduce the amount of effort it takes to develop new functionality. In this section, we will discuss the state of technical debt in the Kibana project.

Analysis tools

To get an idea of the current state of technical debt in Kibana, we ran two code analysis tools to get some insights. We used Codebeat and CodeFactor. These tools analyze all project files separately and apply a rating (from A to F). Code complexity, code issues, and code duplication are examples of metrics that are taken into consideration. We chose these two tools because they provided free access for open source projects, and were hosted in the cloud to so that the analysis could be easily executed.

These tools gave us the following insights:

- *The Kibana project has quite some duplicate code* - The Codebeat documentation explains that they consider this one of the most serious issues, because it harms maintainability. Warnings are already triggered after five duplicate lines of code. Out of the 1315 warnings found by CodeFactor, 952 were about duplicate code.
- *Multiple functions have too many arguments* - Codebeat triggers an information message if functions need four arguments, a warning if functions need five arguments, an error for six, and a critical issue for anything over six. For the kibana project, multiple warnings were generated.
- *Multiple functions have a too large body* - The Codebeat documentation [6] gives a warning if methods have a length 40-60 lines, an error if the length is 60-80 lines, and a critical issue for anything over 80. 300 of the 1315 warnings found by CodeFactor were maintainability warnings.

Roughly half of the debt that was found by these tools is located in our test files. You could argue that certain aspects of technical debt are less urgent when located in test files. Your testing code is completely separated from your production code, functions that are too long could for example also improve the readability of test files, and possible performance degradations could be taken for granted. Nevertheless, we think that coding best practices should also be applied in testing code.

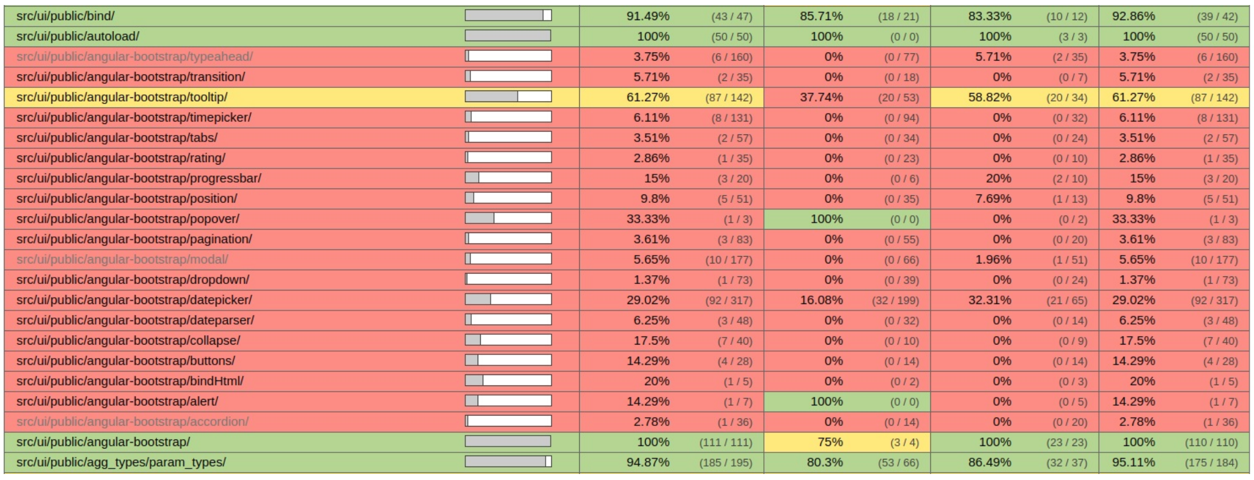
The largest shortcoming of analysis tools like Codebeat and CodeFactor is that they do not look how files relate to each other. They do, however, help to discover trends and find errors.

Code coverage

Code coverage describes the degree to which the source code of a program is executed by a particular test suite. Not having your software tested thoroughly is considered technical debt. According to the coverage report generated by Istanbul, the Kibana project has good test coverage. The overall coverage is 70 percent. In this analysis, we will review two sections that have bad coverage and thus are examples of technical debt.

UI components based on AngularJS

As discussed in issue #7591, the Kibana community is trying to remove the dependency on Angular-bootstrap, which in turn depends on AngularJS, by integrating its source code into Kibana. The UI components that are based on Angular-bootstrap have seen custom tweaks by the Kibana community. Nevertheless, the code coverage (see Figure 7) of these files is below 40%, which is way below that of other files found in the `src/ui` directory. Over time, functionalities have been added and the code has been refactored to better adhere to the styling conventions. This means that Kibana can no longer simply depend on tests written by the angular-bootstrap community. The decision to remove the dependency on Angular-bootstrap, and instead create a copy of the code to serve as a foundation for Kibana's own UI components, created testing debt.

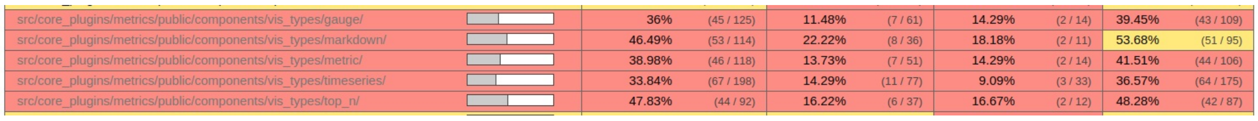


src/ui/public/bind/	91.49%	(43 / 47)	85.71%	(18 / 21)	83.33%	(10 / 12)	92.86%	(39 / 42)
src/ui/public/autoload/	100%	(50 / 50)	100%	(0 / 0)	100%	(3 / 3)	100%	(50 / 50)
src/ui/public/angular-bootstrap/typeahead/	3.75%	(6 / 160)	0%	(0 / 77)	5.71%	(2 / 35)	3.75%	(6 / 160)
src/ui/public/angular-bootstrap/transition/	5.71%	(2 / 35)	0%	(0 / 18)	0%	(0 / 7)	5.71%	(2 / 35)
src/ui/public/angular-bootstrap/tooltip/	61.27%	(87 / 142)	37.74%	(20 / 53)	58.82%	(20 / 34)	61.27%	(87 / 142)
src/ui/public/angular-bootstrap/timepicker/	6.11%	(8 / 131)	0%	(0 / 94)	0%	(0 / 32)	6.11%	(8 / 131)
src/ui/public/angular-bootstrap/tabs/	3.51%	(2 / 57)	0%	(0 / 34)	0%	(0 / 24)	3.51%	(2 / 57)
src/ui/public/angular-bootstrap/rating/	2.86%	(1 / 35)	0%	(0 / 23)	0%	(0 / 10)	2.86%	(1 / 35)
src/ui/public/angular-bootstrap/progressbar/	15%	(3 / 20)	0%	(0 / 6)	20%	(2 / 10)	15%	(3 / 20)
src/ui/public/angular-bootstrap/position/	9.8%	(5 / 51)	0%	(0 / 35)	7.69%	(1 / 13)	9.8%	(5 / 51)
src/ui/public/angular-bootstrap/popover/	33.33%	(1 / 3)	100%	(0 / 0)	0%	(0 / 2)	33.33%	(1 / 3)
src/ui/public/angular-bootstrap/pagination/	3.61%	(3 / 83)	0%	(0 / 55)	0%	(0 / 20)	3.61%	(3 / 83)
src/ui/public/angular-bootstrap/modal/	5.65%	(10 / 177)	0%	(0 / 66)	1.96%	(1 / 51)	5.65%	(10 / 177)
src/ui/public/angular-bootstrap/dropdown/	1.37%	(1 / 73)	0%	(0 / 39)	0%	(0 / 24)	1.37%	(1 / 73)
src/ui/public/angular-bootstrap/datepicker/	29.02%	(92 / 317)	16.08%	(32 / 199)	32.31%	(21 / 65)	29.02%	(92 / 317)
src/ui/public/angular-bootstrap/dateparser/	6.25%	(3 / 48)	0%	(0 / 32)	0%	(0 / 14)	6.25%	(3 / 48)
src/ui/public/angular-bootstrap/collapse/	17.5%	(7 / 40)	0%	(0 / 10)	0%	(0 / 9)	17.5%	(7 / 40)
src/ui/public/angular-bootstrap/buttons/	14.29%	(4 / 28)	0%	(0 / 14)	0%	(0 / 14)	14.29%	(4 / 28)
src/ui/public/angular-bootstrap/bindHtml/	20%	(1 / 5)	0%	(0 / 2)	0%	(0 / 3)	20%	(1 / 5)
src/ui/public/angular-bootstrap/alert/	14.29%	(1 / 7)	100%	(0 / 0)	0%	(0 / 5)	14.29%	(1 / 7)
src/ui/public/angular-bootstrap/accordion/	2.78%	(1 / 36)	0%	(0 / 14)	0%	(0 / 20)	2.78%	(1 / 36)
src/ui/public/angular-bootstrap/	100%	(111 / 111)	75%	(3 / 4)	100%	(23 / 23)	100%	(110 / 110)
src/ui/public/agg_types/param_types/	94.87%	(185 / 195)	80.3%	(53 / 66)	86.49%	(32 / 37)	95.11%	(175 / 184)

Figure 7 - Code coverage of Angular-bootstrap components

UI components based on ReactJS

The Kibana community recently decided to slowly transition away from AngularJS as a whole. As can be seen in Figure 8, testing debt that came with this process is present. The Kibana community still has a long way to go in redeeming this debt.



src/core_plugins/metrics/public/components/vis_types/gauge/	36%	(45 / 125)	11.48%	(7 / 61)	14.29%	(2 / 14)	39.45%	(43 / 109)
src/core_plugins/metrics/public/components/vis_types/markdown/	46.49%	(53 / 114)	22.22%	(8 / 36)	18.18%	(2 / 11)	53.68%	(51 / 95)
src/core_plugins/metrics/public/components/vis_types/metric/	38.98%	(46 / 118)	13.73%	(7 / 51)	14.29%	(2 / 14)	41.51%	(44 / 106)
src/core_plugins/metrics/public/components/vis_types/timeseries/	33.84%	(67 / 198)	14.29%	(11 / 77)	9.09%	(3 / 33)	36.57%	(64 / 175)
src/core_plugins/metrics/public/components/vis_types/top_n/	47.83%	(44 / 92)	16.22%	(6 / 37)	16.67%	(2 / 12)	48.28%	(42 / 87)

Figure 8 - Code coverage of ReactJS components

Comments

An interesting indicator that can help in analyzing technical debt is analyzing comments from a project. We analyzed `TODO` comments in the Kibana project to gain a better understanding of the technical debt that exists in the project:

1. 'Extract this into an external service' - found several times, producing these services

would solve a lot of technical debt.

2. 'We should probably display a message of some kind' - This one is found at various places in the code, which signals that developers do not want to spend time on this kind of chore.
3. 'Override bootstrap styles. Remove !important once we're rid of bootstrap.' - `TODO's` including Bootstrap are found often, so bootstrap is definitely a major cause of technical debt.
4. 'May need to verify this or refactor' is a `TODO` that is found sometimes, referring to a part of the code that needs attention.

We also searched the project for `FIXME` comments. A `FIXME` comment is a comment that elicits a part of the code that needs fixing. This term is used less frequently than the `TODO` comment, which makes sense, as `FIXME's` are more severe. In total, we found 20 occurrences in the codebase listing a wide range of subjects, mostly referring to a small bug that is not too urgent. This causes the developer encountering the problem to defer it to a later point in time, creating a `FIXME` (and some Technical Debt). An example is '*`FIXME: inline moveTo is buggy with excanvas`*'.

Evolution of technical debt and current challenges

In this section, we will discuss how technical debt has evolved in the Kibana project. We will also look at issues that are symbolic for challenges that exist in the current system. These include the removal of external library usage, the transition from AngularJS to ReactJS, and the transition from Kibana v4 to Kibana v5.

AngularJS to ReactJS migration: Kibana was originally written in AngularJS. However, since February 2017, ReactJS has also been in the `packages.json` file. As described in #10271, it is not the goal of the Kibana team to do an immediate complete rewrite to ReactJS. However, the goal is to provide a slow migration to ReactJS, focusing on new apps being built from scratch and features isolated in leaf components.

This will cause major changes to the codebase in the near future. As the transition occurs, developers are required to work with two different frameworks that solve very similar problems. The usage of AngularJS has become a form of technical debt, but no immediate action will be taken to get rid of it.

Removal of external libraries: The Kibana team is trying to get rid of noncrucial external libraries like Lodash or Bootstrap, because it makes the project dependent on external dependencies which might get outdated or unsupported. Bootstrap, for example, *"provides a lot of styles which we don't use/need, and of course, doesn't provide many styles which we do need"* as stated in #7364. This transition is still work in progress but it will be a great cleanup of the code once the team gets rid of it.

Kibana v4 to v5: The transition from Kibana version 4 to version 5.0.0 was a major makeover that added features and also improved the architecture. It made the codebase more robust and more open to additional improvements. This was caused mainly due to the new architecture, which splits up the code into different plugins. As since version 5.0.0 each option in the main menu is a plugin in the code, every component can be maintained, extended, and updated separately from the rest of the codebase. As a result, it has also become easier to add code to Kibana. One can simply create a new plugin that can be self-contained in a single directory. This can be considered a major payoff of the technical debt that had been around for quite some time in earlier versions.

Conclusion

In this chapter, we gave an overview of different aspects of Kibana from a software architectural point of view. We identified different stakeholders, which gave us a great overview of who is involved with the development of Kibana, and how they rank in terms of interest and power. We discussed several architectural views defined by Rozanski and Woods [1] to fully understand Kibana's inner workings. Hereafter, we discussed issues and integration, stating the workflow of how issues translate into pull requests, and finally into integration. Finally, we analyzed the Kibana project from a higher level and tried to map out its status. We did this by identifying technical debt and looking at how this debt has evolved over time.

Looking back, a couple of interesting conclusions can be made. First of all, we think that the project architecture, that has its business logic defined in plugins, is a good architecture. It makes the code modular and allows programmers to easily extend Kibana's functionality. This architecture, that was introduced with the release of Kibana version 5, has helped to make the code more maintainable.

A more troubling observation is the migration process of the front-end framework AngularJS to React. This migration process is slowly being rolled out, and currently results in different front-end frameworks used simultaneously. Our code analysis learned us that these components are also badly tested, and hence contain technical debt. The current combined use also introduces confusion for future developers that want to contribute to the project. We think it is better to bite the bullet and try to migrate as quickly as possible.

To conclude, the Kibana project is a well-managed open source project that has evolved over time. It is able to leverage its technically informed user-base to solve issues and technical debt. With the exponentially growing volumes of data that are being generated worldwide, we are confident that Kibana will play an even more important role in the future.

References

1. Nick Rozanski and Eoin Woods - Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. Igor Steinmacher et al - Overcoming open source project entry barriers with a portal for newcomers. ACM, New York, 2016.

Magento: an e-commerce platform for growing your business online

By Hans Schouten, Elvan Kula, Kin Lok Chow and Jan-Gerrit Harms

Delft University of Technology, 2017

Abstract

Magento is the fastest growing and most widely used e-commerce platform in Europe. It was first released in 2007 with an open source license. Magento differentiates itself from competitors by its remarkable amount of flexibility and large scope on customer targeting. This chapter analyzes Magento's software architecture and the way the developers tackle challenges related to security, maintainability and technical debt. We will look more closely into Magento's layered design, extensive use of design patterns, information flow and security measures.

Table of Contents

Introduction

The problem with e-commerce software platforms has always been that these solutions never exactly provide what businesses are looking for. The business calls for a tailored solution. One that delivers on power, flexibility and scalability to meet the requirements of any business. Magento is the exception.

Magento is the fastest growing and most widely used e-commerce platform in Europe [1]. It is an open source platform that can be used by web shop developers to grow their online businesses. The e-commerce solution currently has over 200,000 online retailers and is chosen by one of every four online businesses, which is more than any other provider [2]. Magento offers powerful features such as advanced marketing, search engine optimisation and catalogue-management tools to control the appearance, content and functionality of an e-commmerce website.

In 2007 the first release of Magento was developed by the company Varien with support of volunteers [3]. In 2015 Magento Commerce launched as an independent company and released Magento 2.0, with an aim to provide new ways to improve the user experience [4]. The important milestones in Magento's history are depicted in Figure 1.

Magento’s powerful features require an architecture that is modular, extensible, flexible and simple to maintain. Offering such qualities, dealing with a diverse user base and managing technical debt are the biggest challenges for Magento. This chapter will give insights into the way the developers of Magento deal with these challenges. First, the chapter will look into the stakeholders and context view of the project. Next, the layered design, information view and security measures of the system are described. This will be followed by an analysis of the technical debt in Magento.

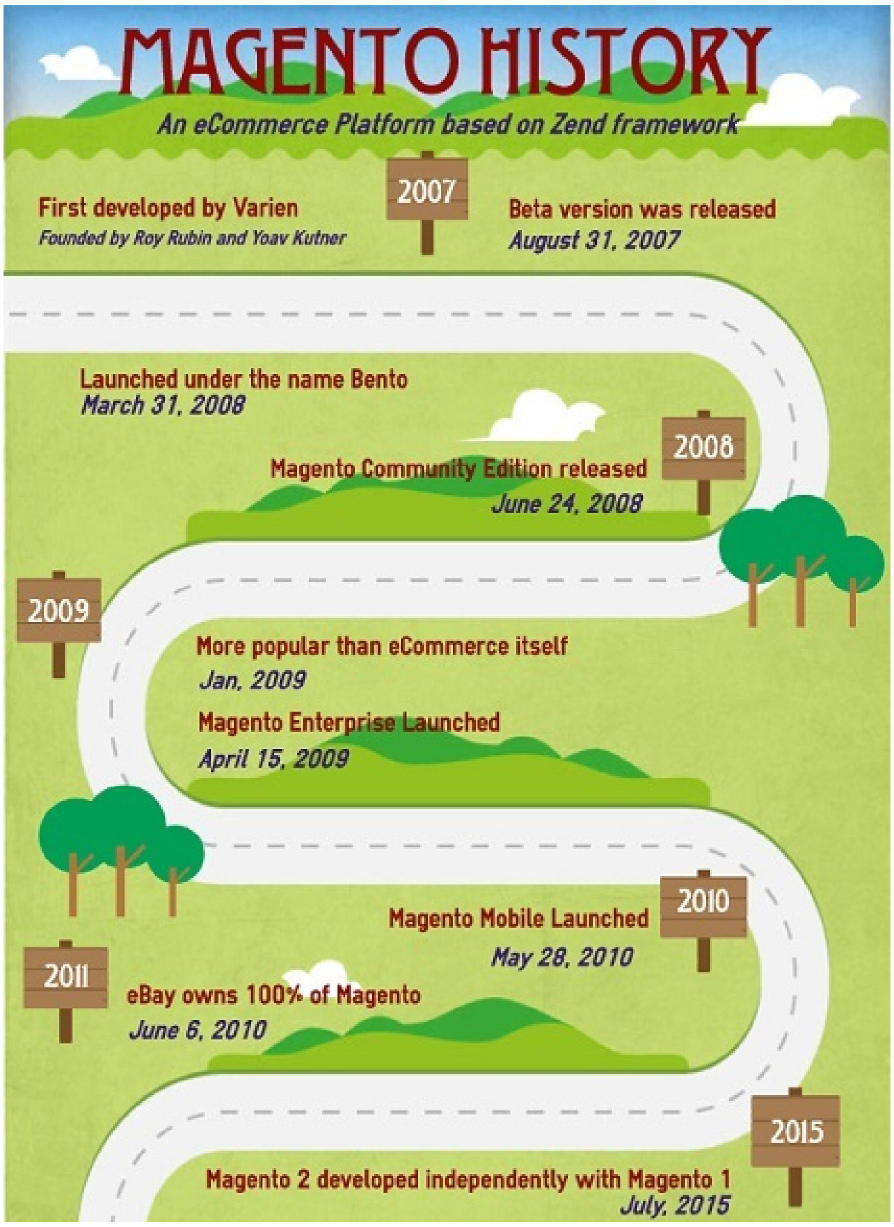

MAGENTO HISTORY
An eCommerce Platform based on Zend framework
2007
First developed by Varien
Founded by Roy Rubin and Yoav Kutner
Beta version was released
August 31, 2007
Launched under the name Bento
March 31, 2008
2008
Magento Community Edition released
June 24, 2008
2009
More popular than eCommerce itself
Jan, 2009
Magento Enterprise Launched
April 15, 2009
2010
Magento Mobile Launched
May 28, 2010
2011
eBay owns 100% of Magento
June 6, 2010
2015
Magento 2 developed independently with Magento 1
July, 2015

Fig. 1: Roadmap of milestones in Magento's history

Stakeholder analysis

This section identifies the key stakeholders of Magento and describes how they apply to the project. This will be followed by an analysis of their influence and associated levels of power and interest.

Key stakeholders

The main stakeholder is Magento Commerce. The organization and its employees exclusively fulfil the role of three of the eleven classes of stakeholders defined by Rozanski and Woods [5]: *acquirers*, *assessors* and *communicators*. The other stakeholders applicable to Magento are listed in the following table.

Stakeholder	Description
Developers	Magento is developed by a highly active community of nearly 400 contributors. Most of the developers are employees of Magento Commerce and focus on contributing to specific features. The external developers can be classified in two groups, namely the *frequent community contributors* and *third-party contributors* based on their commit frequency.
Testers	Magento's chief architect, Alan Kent, is responsible for internally running the automated tests of Magento's testing framework. The individual contributors take responsibility for raising concerns and fixing minor bugs.
Integrators	All integrators of Magento are employees of Magento Commerce. Oleksii Korshenko, Igor Miniailo and Max Yekaterynenko, were identified as general integrators who are responsible for evaluating all pull-requests in general. Ievgen Shakhsuvarov, Eugene Tulika and Vitalii Korotun were identified as specific integrators who are assigned to specific types of pull requests.
Competitors	Magento's competitors can be divided into two types: hosting and self-hosted solutions. Magento belongs to the self-hosted set of platforms. Competitors in the same area include WooCommerce, Zen Cart, osCommerce, OpenCart, X-Cart, SpreeCommerce and DrupalCommerce. Competitors that provide hosting are Shopify, BigCommerce, Yahoo Commerce and Volusion.
Maintainers	Alan Kent, is an important maintainer as his role is to keep an overview of the maintainability of the architecture.
Suppliers	As Magento is run on a local webserver, the user himself is responsible for the arrangement of the hardware and infrastructure on which the platform will run. However, there are numerous web hosting companies that offer Virtual Private Servers and Dedicated Servers that are specially designed for hosting Magento webshops [6]. The largest Magento hosting provider is RackSpace.
Support	Customers with a Magento Enterprise Edition can request direct assistance from Magento's customer support. Non-enterprise customers can request assistance from the Magento community via their forum [7] or from their network of 300+ solution and technology partners [8].
Users	The users are developers who want to implement a webshop or a similar application based on Magento. They need to easily understand the sample code and quickly identify those lines of code with shop logic. Code quality and correct use of technologies are important for them. Companies that hire developers to build these tailored applications are a more indirect type of users, the so-called clients. Some of Magento's largest clients are Burger King, Nestle, Murad and Coca-Cola [9].
End users	The end users are people who would visit the webshop in case it went live. They need an intuitive layout that allows them to shop with ease in

	a few clicks.

Table 1: Key Stakeholders of Magento

Stakeholders influence

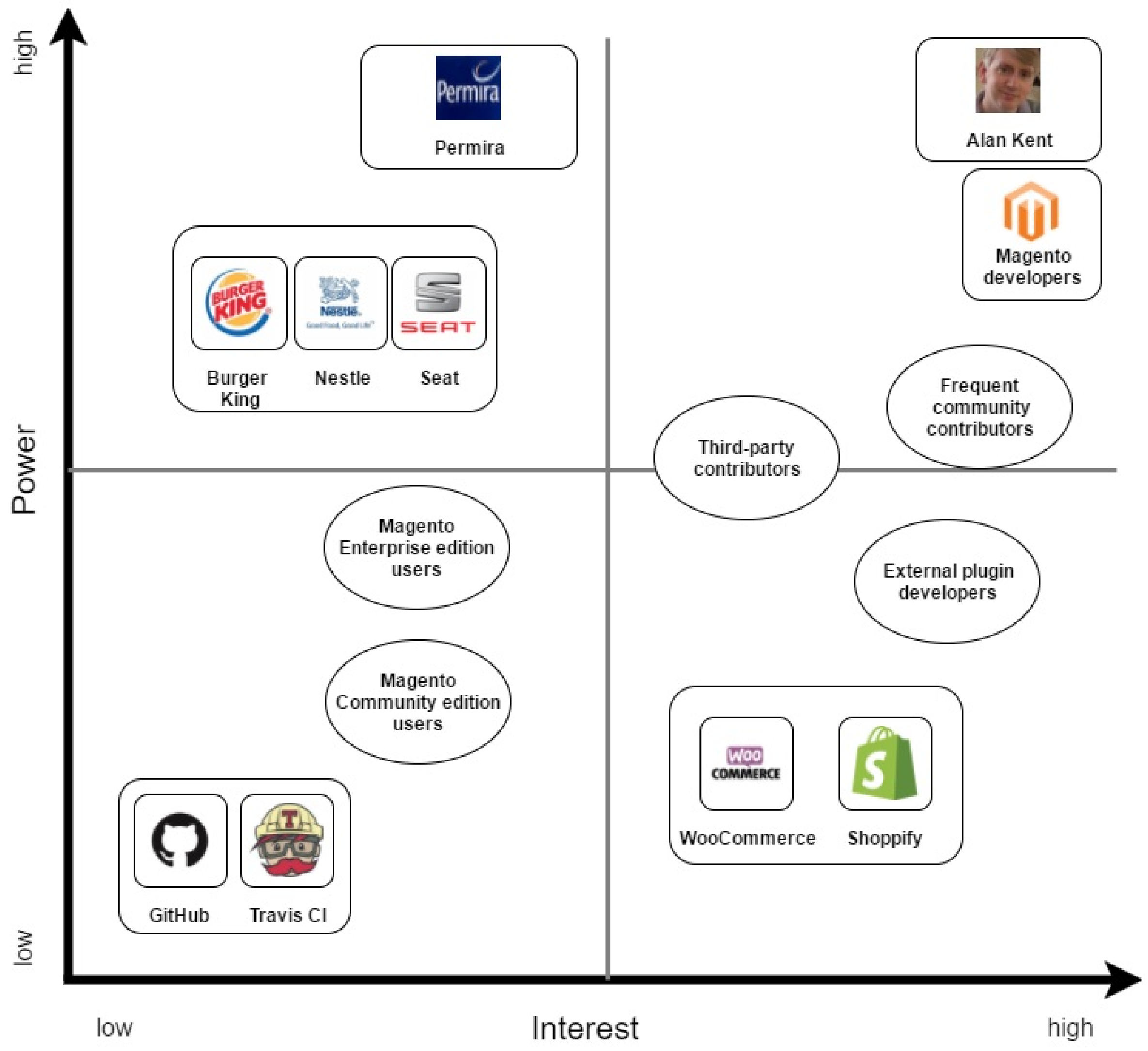


Fig. 2: A power-influence grid showing the positions of key stakeholders

The power-interest matrix shown in Figure 2 visualizes a measurement of the power of the stakeholders compared to their interest. The grid consists of four quadrants with the most powerful entities in the upper-right quadrant and the least powerful ones in the opposite lower-left quadrant. Below, we elaborate on responsibilities of some stakeholders and based on their power-interest ratio explain how Magento Commerce should engage with them:

- **High power, interested people** - *Manage closely*: The main contributors of Magento can be viewed as the most important stakeholders since they are the ones who make the project possible. They are responsible for fulfilling the user requirements and solving issues while maintaining a good quality of code. Their interest in a good functioning system is very high.

- **Low power, interested people** - *Keep informed*: In the lower-right quadrant we see the external plugin developers. These are the companies or individuals who create the free/paid modules and themes for the Magento site owners. Those with almost no power, but pretty high interest are the competitors of Magento. They have to keep a close eye on each other to stay in the market.
- **High power, less interested people** - *Keep satisfied*: Those with high power, but less interest are Permira and the largest clients of Magento (Burger King, Nestle and Seat). As the company's largest investor, Permira Funds holds a big stake in Magento Commerce. The largest clients also have a say in the future direction of Magento as they are responsible for a major part of the company's value creation. They have great influence over the way the system is designed and develops. As the system is developed it is normal for these large clients to be questioned about the type of enhancement or design that they prefer.
- **Low power, less interested people** - *Monitor*: In the least important category we find supporting services such as GitHub, Travis CI and the hosting providers. They are responsible for providing services regarding building, maintaining and hosting Magento's software. The single users are also located in this quadrant as they have less power compared to the larger clients.

Context view

A context view describes the relationships, dependencies, and the interactions between the system and its environment. This view is relevant for the system's architecture and defines the boundaries of the system and how it interacts with external entities across these boundaries. Figure 3 shows the context view of the Magento project.

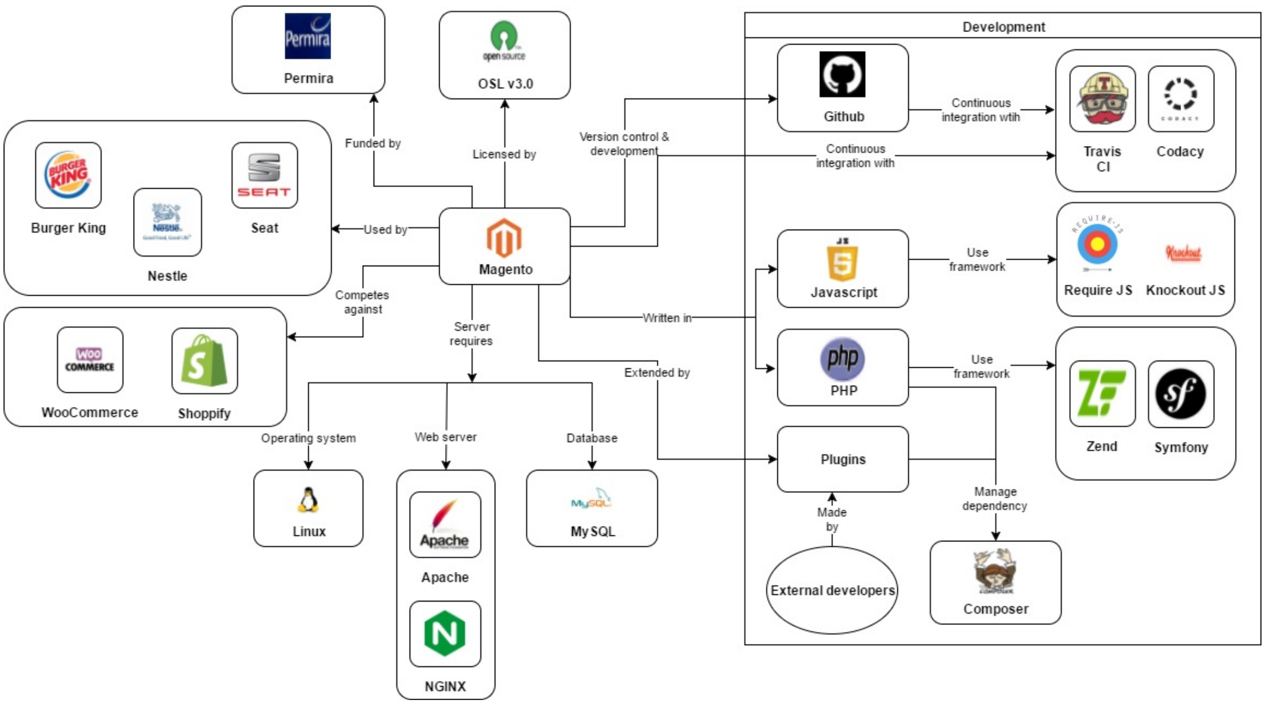


Fig. 3: An overview of the relationships between the Magento platform and its external entities

Development entities

Magento is written in PHP and its front-end is based on the JavaScript library Knockout.js, enabling developers to provide their webshops with dynamic UIs. KnockoutJS, other frameworks and scripts are imported via RequireJS to manage JavaScript dependencies easier and to improve page load time. For continuous integration Travis CI and Codacy are used. These services are not only used for continuous integration, but also function as separate integration tests to check compatibility.

Hosting entities

In order to host a Magento webshop, you need a MySQL database or any variant which is forked from the MySQL database, like MariaDB or Percona. The database should be hosted on a webserver, either on NGINX or Apache. The operating systems on these servers support most Linux distributions.

System scope

Magento differentiates itself from competitors by having a large scope of options when it comes to customer targeting [10]. The platform is a complete e-commerce solution that offers a lot of extra functionality in marketing, search engine optimisation and advanced search engine optimization features. On top of that, Magento offers a remarkable amount of flexibility to users. There are many possibilities of customizing the store with additional features and functionality with more than 7000 extensions in the Magento Marketplace.

Therefore, the scope of Magento is to provide a rich set of features for customer targeting and customization options, as well as the functionality that is already present in existing e-commerce platforms.

Development view

This section presents an overview of Magento's architecture by first describing the principles and guidelines that govern the development of the system. This will be followed by an overview of the as-designed development view and as-implemented view of Magento.

Architectural principles

Magento's architecture is mainly built on concepts that allow for maximum flexibility and extensibility of its software:

- **OOP Principles**

As a PHP framework, Magento takes advantage of Object-Oriented Programming (OOP) principles that provide simplicity and extensibility. The concepts of data encapsulation and inheritance are especially useful for Magento's front-end components [11].

- **Modularity**

The concept of modularity lies at the heart of Magento. At the highest level, Magento's architecture is composed of core components and optional independent modules. These modules are organized by feature and can be used to modify the appearance and behaviour of a webshop without altering other parts of the code.

- **Extensibility**

Product extensibility has always been taken into account from Magento's earliest design stages [12]. Magento 2 uses automatic *dependency injection* and *service contracts* to facilitate new implementations of existing functionality [13].

- **Stack of open-source technologies**

The Magento stack contains many open-source technologies for deployment and customization of storefronts.

Development guidelines

Test strategy

The Magento team has an extensive testing framework which is divided into six different levels of testing, namely:

- **Static tests** Static testing checks if the code adheres to coding standards that are set by the Magento team.
- **Unit tests** The Magento team proposes to use Test-Driven Development (TDD) for unit testing. The team only uses unit tests for testing PHP code and defines a separate category for other programming languages.
- **JavaScript tests** JavaScript tests are defined as unit tests for all Magento components that are written in JavaScript, which are mainly part of the front-end. The JsTestDriver library is used for running these tests in the browser.
- **Integration tests** The integration tests are run in different levels of isolation. Magento does not use the browser for the integration tests which makes the tests more granular than functional tests and a lot quicker.
- **Functional tests** The functional tests are run using the separately provided Functional Testing Framework (FTF) for smoke testing, acceptance testing and regression testing. The Magento framework uses a web browser and the selenium server to remote control user interaction.
- **API functional tests** API functional tests take the application as a black box and only test from the client's perspective (the Web API endpoints). The API calls are managed by a common class which is inherited by all test cases and uses one function to access the API using either REST or SOAP.

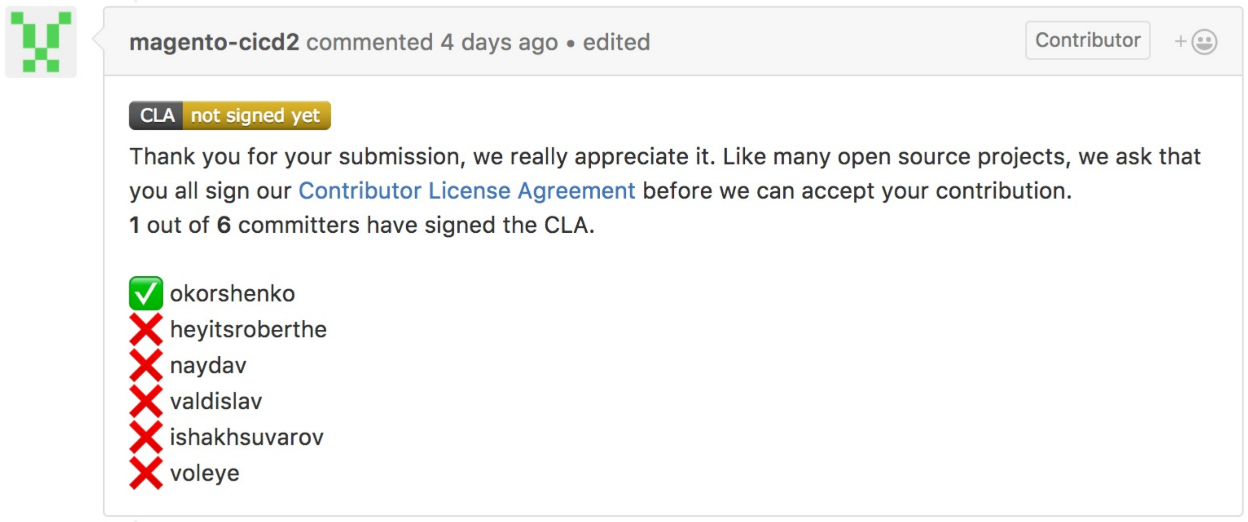
All the tests described above are located in the `<magento2 root dir>/dev/tests` folder. The team provides an easy integration with the PHPStorm IDE to ease the setup and running of tests [14].

Contribution process

All contributions are done via the *fork and pull model* with the integrators reviewing pull-requests on a first-in-first-out basis. One's code only has the chance of being merged if it adheres to the rules in the developers guide:

- Each code extension needs to adhere to the coding standards.
- The Magento project has its own Definition of Done (DoD) defining a set of acceptance criteria that is applied to any changes in the code base. These criteria revolve around readability, sufficient code coverage and solid documentation.

- Each commit needs to pass the automated tests that are in place.
- Everyone who wants to contribute to Magento's source code is required to accept the terms of the contributor agreement. This is automatically checked as illustrated in Figure 4.



magento-cicd2 commented 4 days ago • edited

CLA not signed yet

Thank you for your submission, we really appreciate it. Like many open source projects, we ask that you all sign our Contributor License Agreement before we can accept your contribution.
1 out of **6** committers have signed the CLA.

✓ okorshenko
✗ heyitsroberthe
✗ naydav
✗ valdislav
✗ ishakhsuvarov
✗ voleye

Fig. 4: To make a contribution to Magento's source code you need to sign their Contributor License Agreement

Original design

Architectural layers

Magento's core product code has a layered design. For Magento's customers specifically, a layered architecture provides the benefit of separating presentation logic from business logic. This simplifies the divided customization of store appearance on the one hand and store behaviour on the other. Architectural layers also provide developers a simplified model for the ideal placement of features and code in the system.

The design of Magento can be decomposed into clusters of components with a similar functionality. These clusters, or layers, enable the logic separation of different responsibilities. Figure 5 illustrates the layered architecture of Magento and shows the components of each layer. The diagram also demonstrates the connections between the four layers and the Magento framework, third party libraries, the supported database, and other technologies. [15]

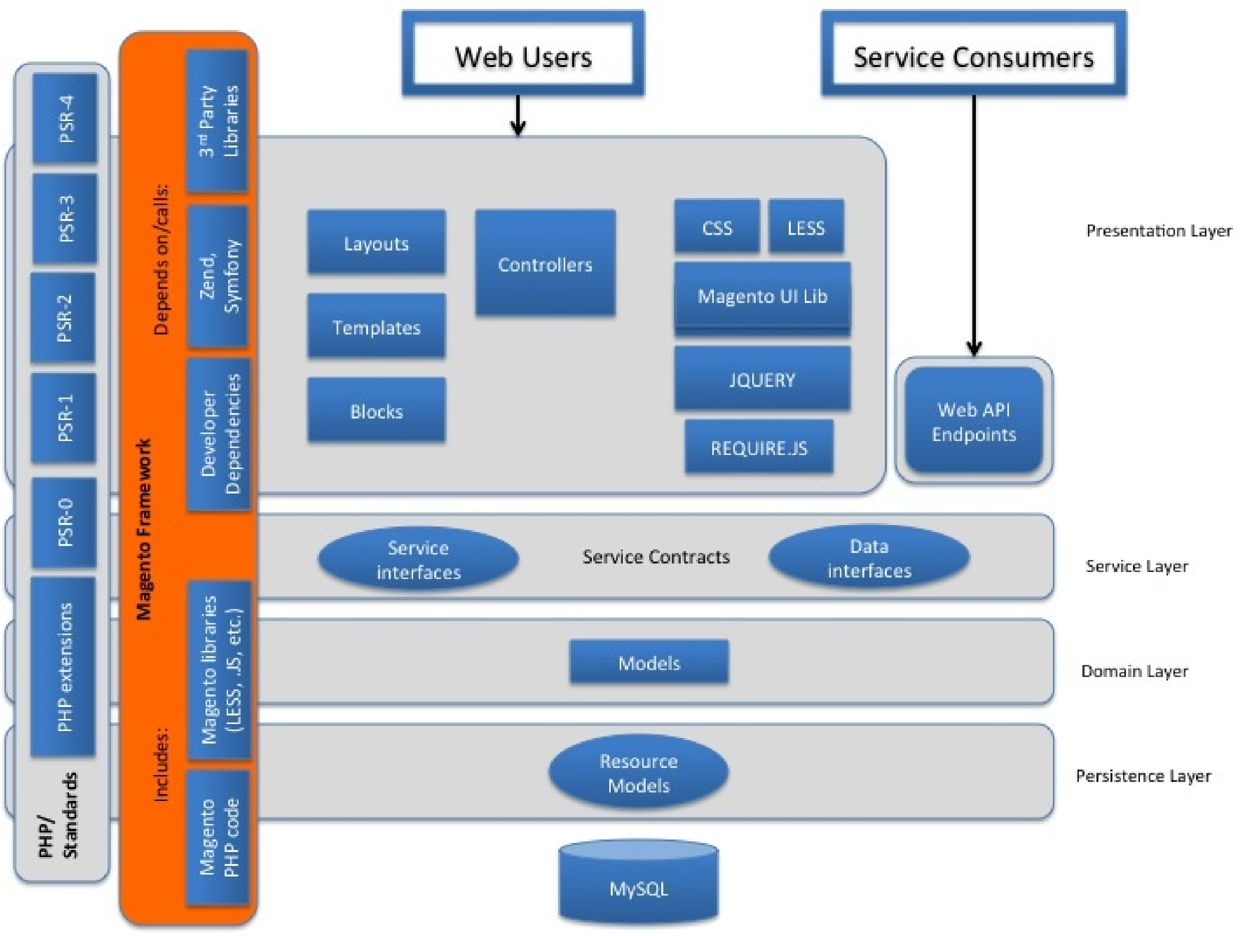


Fig. 5: Magento's architecture layered diagram [15]

1. **Presentation layer** - This layer is responsible for the interaction between users and the Magento Framework. Components in this layer are in charge of handling incoming user requests by executing code of the controller classes which in turn load UI components [16].
2. **Service layer** - This layer provides a bridge between the presentation layer and the domain layer [17]. It uses PHP interfaces to form service contracts for Magento components. These contracts define through data interfaces how the components of the higher presentation layer and the lower domain layer should interact. This layer also prescribes service interfaces to provide an easy way to access Magento's API Framework for external REST and SOAP API applications.
3. **Domain layer** - This layer holds all business logic that Magento modules perform on data that is received in requests. A special use case of the models is when access to resources (for instance a database) is required. Higher layers do not have direct access to these resources, but use the resource model abstractions of the domain layer instead.
4. **Persistence layer** - This layer is responsible for all access to storage resources. Since Magento supports different databases, it offers different database adapters to accommodate them all [18].

Architectural patterns

A design pattern is a reusable solution to a commonly occurring problem. Since Magento contains functionality that matches these problem descriptions, using design patterns is an elegant way to implement the Magento functionality. Therefore, Magento 2 contains various design patterns. The Object Manager alone already contains more than 11 design patterns. In this section, however, we will only cover the most important design patterns of Magento.

Model-View-ViewModel/Controller (MVVM/MVC) model

To have a well-defined standardization for modules, the core developers decided to implement the Model-View-Controller (MVC) design pattern in Magento 1. In Magento 2 this model transformed into a more complex variant that is "closer to a Model, View, ViewModel (MVVM) system". [19] The benefit of MVVM over MVC is its simplification of data binding and the manipulation of UI elements [20].

In Magento 2 a module corresponds to a URL. The URL is routed to the action method of a controller like in a classical MVC system, however in Magento's design the Controller has less responsibilities. As the Controller does not directly set variables of the View, the variables handle their own fetching from the Model instead. The HTML page that the controller returns to render has many sections called Containers which are constructed from a set of Blocks. A block corresponds to the ViewModel part of the MVVM pattern while the contained `pthml` file, a layout template for the Block, is the View. The implementation of this pattern is shown in Figure 6 and the way modules employ MVVM can be found in the part of the Logical view.

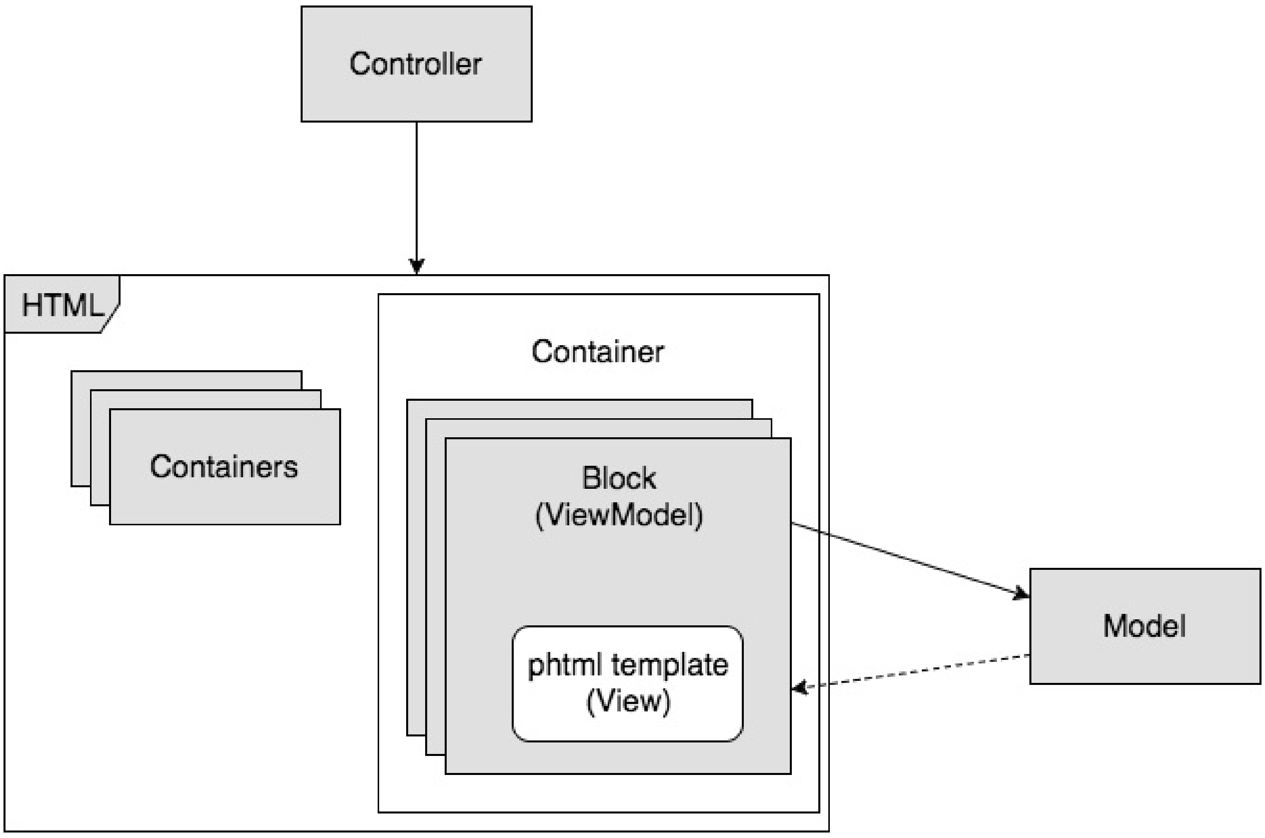


Fig. 6: Magento implementation of the MVC-MVVM pattern

Factory pattern

The factory pattern is responsible for the instantiation and automatic loading of classes. It is used in multiple locations in the Magento code. A class in Magento can be instantiated by a factory by calling a method with an abstract name that defines its class group and name.

The factory pattern can for example be found in the instantiation of objects by the Object Manager class [21] as visualized in Figure 7 [22]. The Magento development team tries to limit the dependence on and direct use of the ObjectManager in code. Factories are an exception since they need the ObjectManager to create specific types of objects.

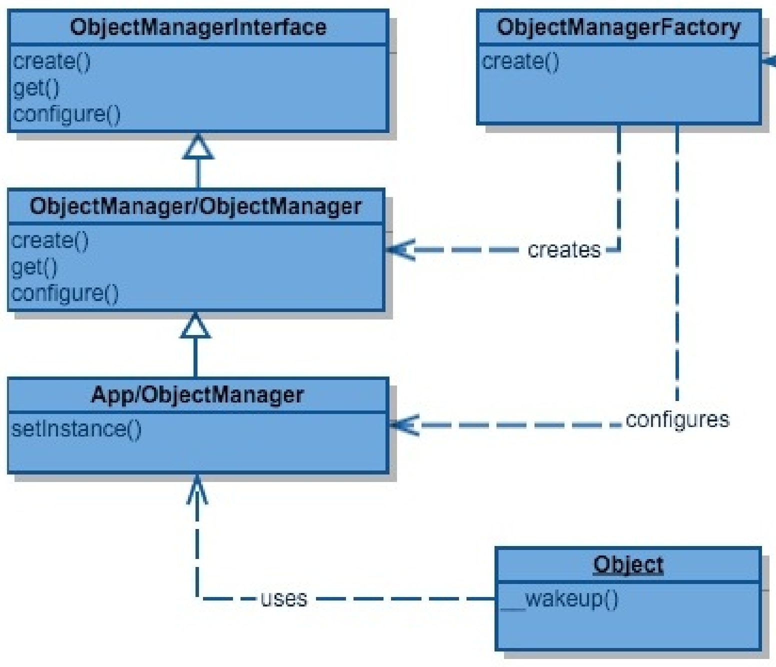


Fig. 7: Example of the Factory Pattern in Magento 2 [23]

Observer pattern

The event-driven architecture of Magento has been made possible by the observer pattern. The pattern is based on a one-to-many dependency between objects so that when one object's state changes, all its dependents are informed and updated accordingly [22]. Magento uses its XML data to define observers. If an event is fired with `Mage::dispatchEvent($eventName, $data)` , the data storage will be consulted and the appropriate observers for `$event` will be fired. In Magento 2, the observer pattern is for example implemented in the payment process as shown in Figure 8.

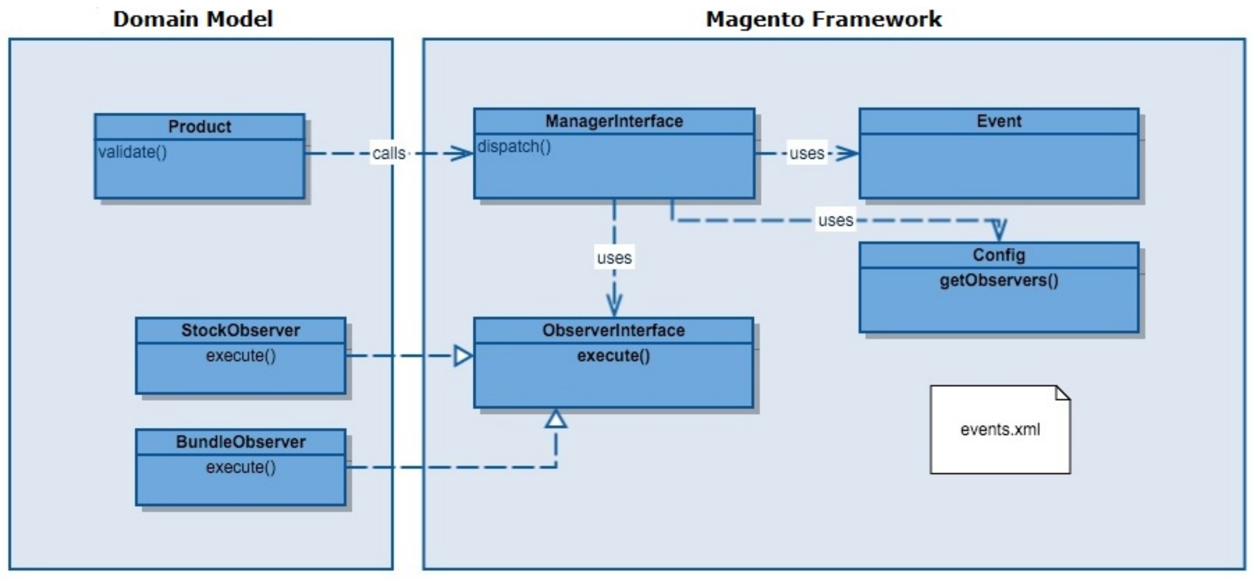


Fig. 8: Example of the Observer Pattern in Magento 2 [23]

Strategy pattern

The strategy pattern enables the application to vary an algorithm's behaviour independently from users at runtime. In Magento 2, this pattern is used in the algorithms behind building payment requests [22]. Figure 9 shows an example in which the pattern recognizes two types of functions, namely the CaptureBuilder and PartialBuilder. The CaptureBuilder is used in the default case, however under certain conditions the function will be interchanged to adapt PartialBuilder.

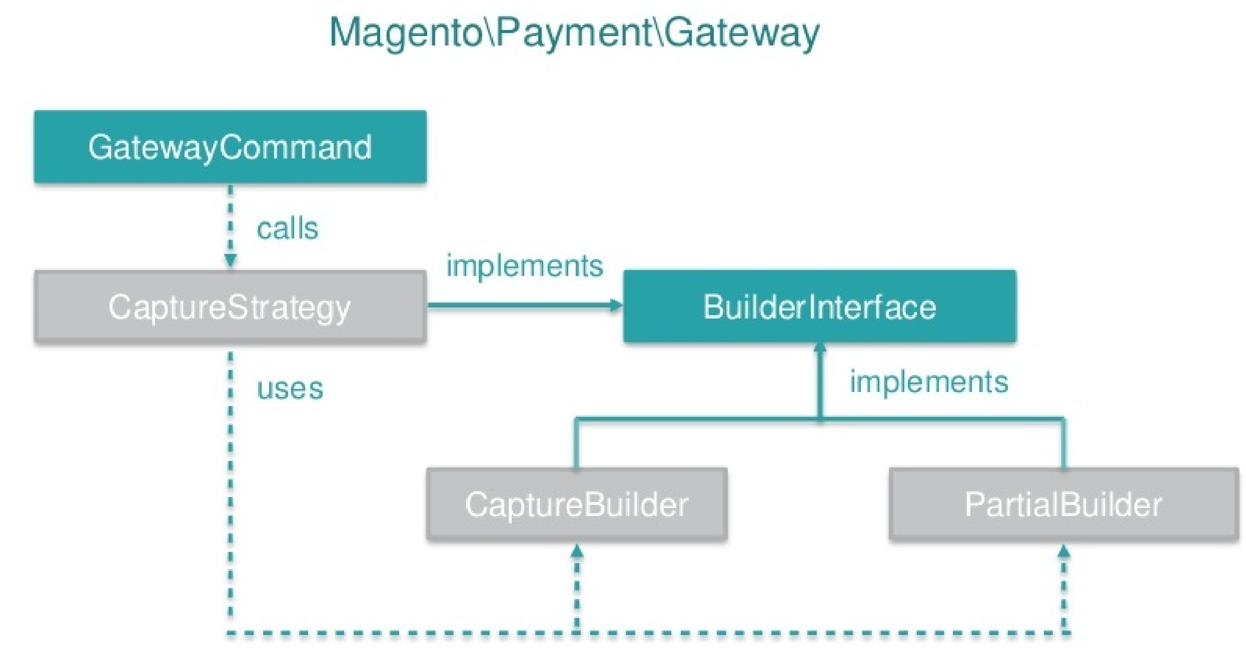


Fig. 9: Example of the Strategy Pattern in Magento 2 [22]

Decorator pattern

The decorator pattern allows responsibilities to be added dynamically to an individual object. In Magento 2, this pattern is implemented to add new behaviour to the factory used by the ObjectManager. In Magento 2, this pattern is implemented to add new behaviour to the factory used by the Object Manager [22]. This can be seen in Figure 10.

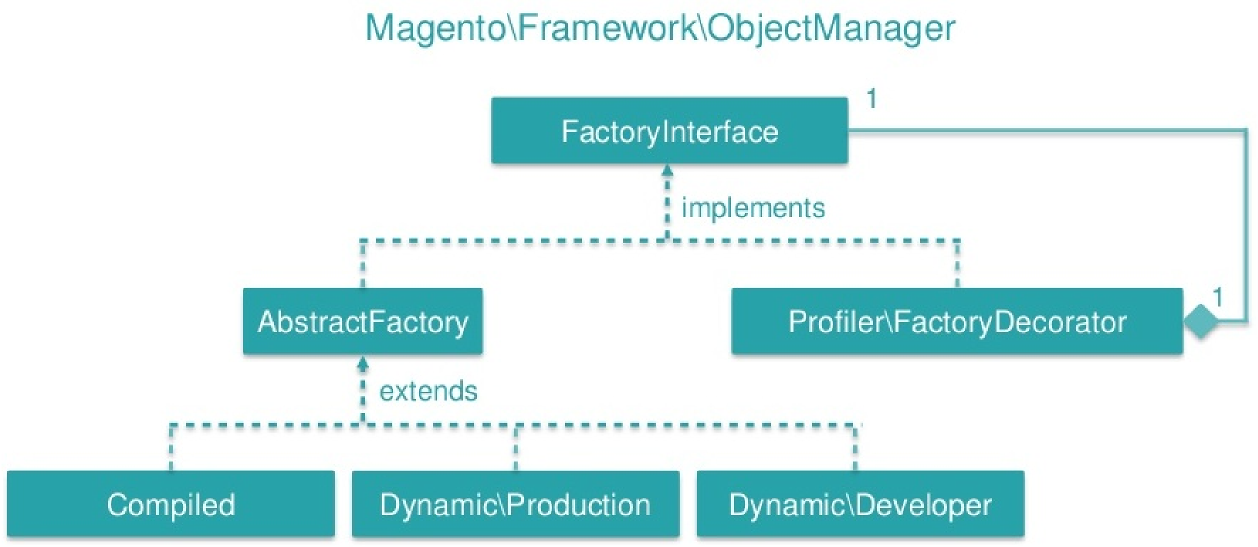


Fig. 10: Example of the Decorator Pattern in Magento 2 [22]

Actual implementation

The Magento Framework

The Magento Framework manages the interactions and information flow between application components. This includes processes regarding routing, indexing, caching and exception handling. The framework is positioned under the domain layer and surrounds the presentation, service and domain layers. More details about the framework's folder structure and its libraries can be found in the section in the Magento developer documentation.

Logical components

Magento makes use of what they call 'Magento components' to enable users to customize the server-side visual representation of their webshop. The 'Magento components' are divided into *modules*, *themes* and *language packages*. Below we will look more closely into modules and the way they employ the MVVM pattern.

Modules

As explained before, modules employ the MVVM pattern. These are the most important implementation details that this pattern entails:

- **module root**: All modules are located in the `<magento2 root folder>/app/code` folder. These modules can be basic features developed by the core development team, or extensions built by external developers. The naming convention is `VENDOR_FUNCTIONALITY`. The developers each have their own folder, so a vendor specific module can be found in `<magento2 root folder>/app/code/<vendor_name>`.

- **main config**: Configuration files are written in `.xml` format with the `<module root>/etc/module.xml` being the main configuration file. It specifies the name and version of the module. The same name also needs to be added to the `<magento2 root folder>/app/etc/config.php` file in which the module we would like to use should be specified.
- **controller**: In order to respond to a URL, the module requires a controller. This consists of two parts. The `<module root>/etc/frontend/routes.xml` specifies which routes the module should be able to respond to. Magento then transforms the URL to a controller class name and tries to find it at `<module root>/Controller/SECOND_URL_COMPONENT/THIRD_URL_COMPONENT.php`. If the controller exists, its execute function is called which determines how to respond (e.g. sending a HTML document or a redirect).
- **view**: Finally, the view is described by the *full action name layout handle XML file* which defines hooks for the page layout system to include the Block of this component. Inside this file we refer to a block class which should be created at `<module root>/Block/BLOCKNAME.php`. Usually, the Block class populates a template which will be rendered. This template is a `phtml` file located at `<module root>/view/frontend/templates/TEMPLATENAME.phtml`. The Block would usually get data from the model using requests or from other data sources and inject them into the view template.

In Figure 11 we can see an example of a module structure.

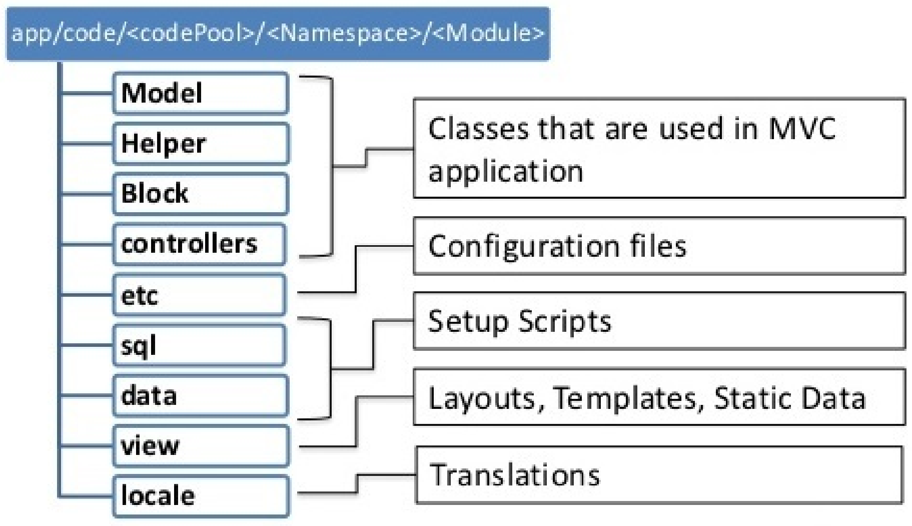

app/code/<codePool>/<Namespace>/<Module>
Model
Helper
Block
controllers
etc
sql
data
view
locale
Classes that are used in MVC application
Configuration files
Setup Scripts
Layouts, Templates, Static Data
Translations

Fig. 11: Example structure of a module [24]

Information view

Magento webshops need to process many requests, orders and purchasing data. In order to work with these data, the Magento development team provides a proper interface for data management. This section describes the way that Magento's architecture stores, manages and migrates information.

Data storage

The persistence layer is responsible for all access to storage resources in Magento. In order to provide a safer query execution and to simplify access to the database, it employs the PHP Data Objects (PDO) interface. The data-access abstraction that PDO offers is currently the recommended method for interacting with databases in PHP [25]. For communication with the higher Domain layer, the Persistence layer adopts the Object-Relational Mapping (ORM) technique [26]. In this technique the communication with the database is performed via resource models. A resource model is an extension of a Domain Layer model that allows database access in an intuitive way. It creates an abstract representation of a particular database table. Each instance of the resource model represents a single record in the table the model corresponds to. A resource model is responsible for performing functions such as [27]:

- **Executing all CRUD (create, read, update, delete) requests**. The model contains the SQL code for completing these requests.
- **Executing additional business logic**. For example, a resource model could perform data validation, start processes before or after data is stored or perform other database operations. The Magento ORM also allows the request of multiple records at once, by providing resource models of the type collection. Figure 12 shows an overview of resource models and collection models in Magento.

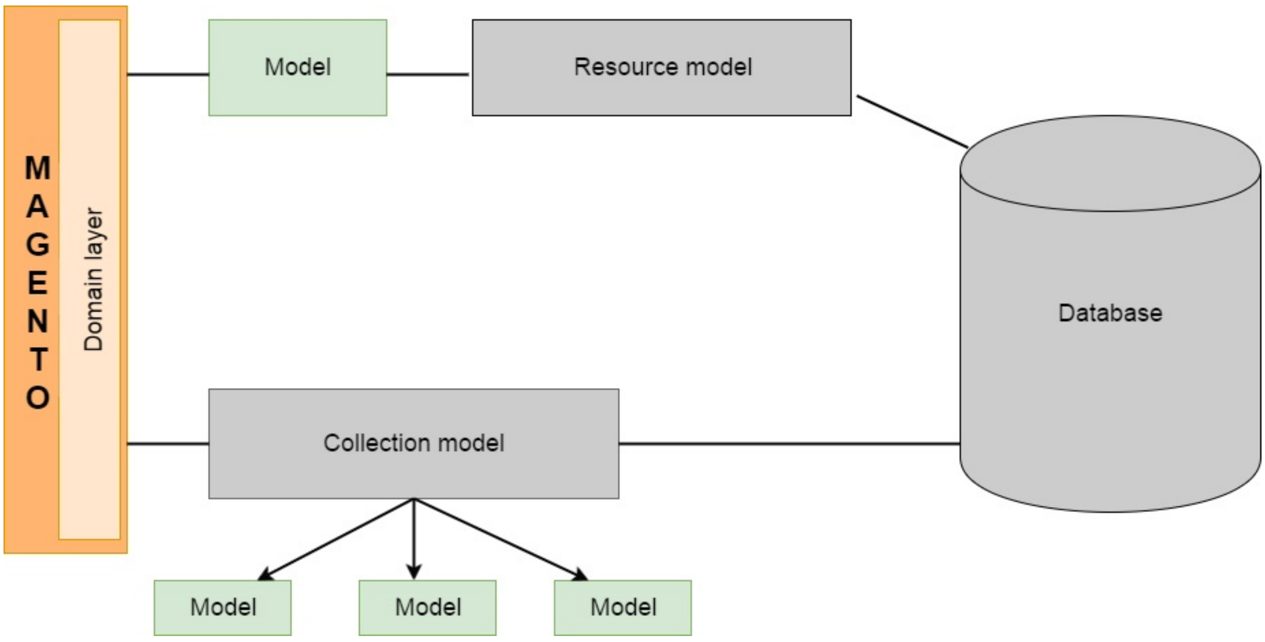


Fig. 12: Resource models represent records in the database

A simple resource model defines and interacts with a single table. However, some objects have many attributes or a set related objects with a variable number of attributes. In these cases, the objects are composed using Entity-Attribute-Value (EAV) models. In result, any model using an EAV resource has its attributes spread out over a number of MySQL tables. The Customer, Product and Catalog resource models of Magento use EAV attributes.

Data management

Magento makes use of a relational database management system with support for MySQL, MariaDB and Percona databases [28]. For purposes of flexibility, the Magento database employs EAV models [29]. In EAV, each modelled "entity" (e.g. a product) has a different set of attributes. Once a new product attribute is necessary, an EAV is added instead of adding an extra column to an ever-growing product table. EAV is very useful for a generic e-commerce solution since various stores may have different characteristics. EAV enables the developers to extend attribute sets without the need to redesign the database [30]. Although it offers a lot of flexibility, not many open source or commercial databases make use of EAV. Hence, the Magento developers have constructed an EAV system out of PHP objects that adopts MySQL as a data-store. In other words, they have built an EAV database system on top of a traditional relational database.

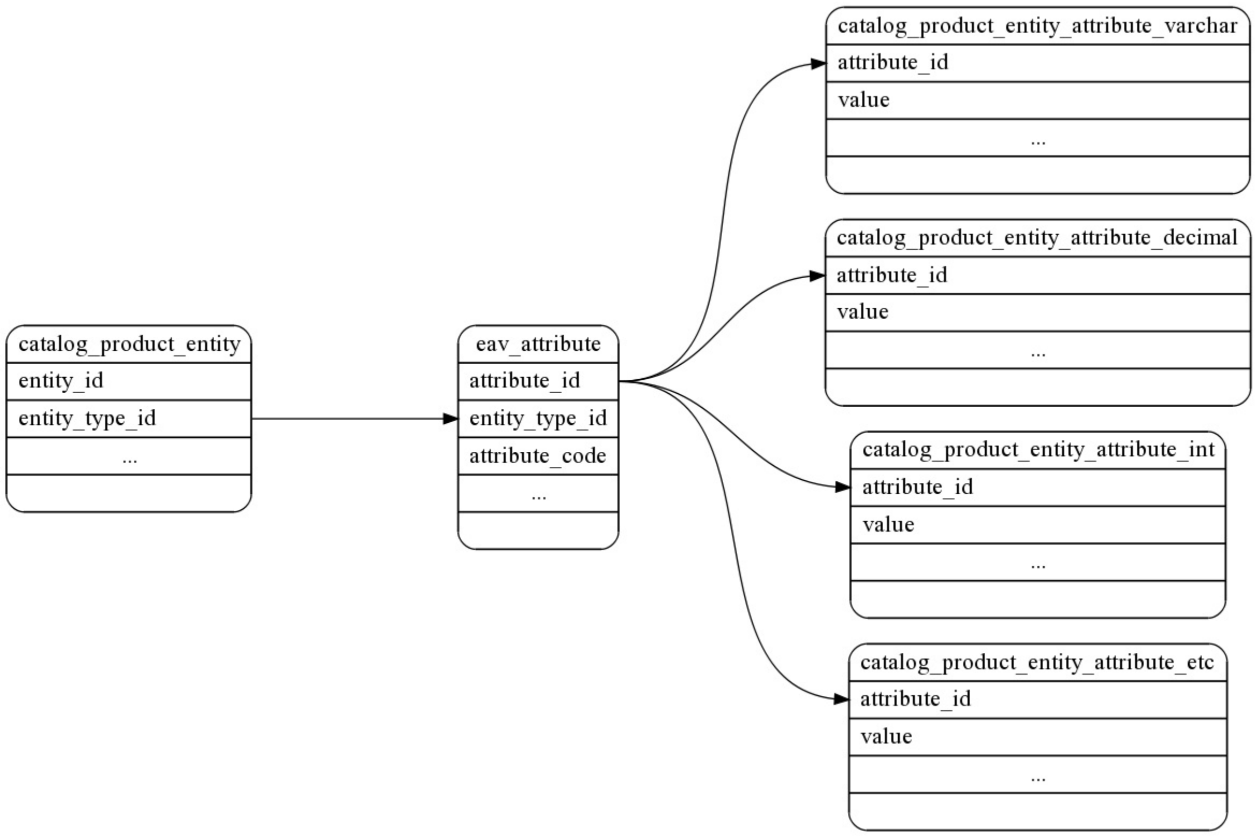


Fig. 13: Overview of Magento’s database tables [30]

Figure 13 is an overview of the database tables Magento interacts with to gain access to an EAV record for a *catalog_product* entity. Each individual product contains a row in *catalog_product_entity*. All the available attributes in the whole system are stored in *eav_attribute*, and the actual attribute values are stored in tables with names like *catalog_product_entity_varchar*, *catalog_product_entity_decimal*, *catalog_product_entity_etc*.

Security

E-commerce websites have always been a lucrative target for hackers, because of the sensitive data they contain, including credit card information. Especially with open source projects, security concerns present themselves as hackers know the internals of the application. That is why security should be the number one focus of all Magento developers and users. This section describes the vulnerabilities of the Magento system and the security measures that have been taken from the developers' side and that should be taken from the users' side.

Magento vulnerabilities

Based on the published patches and compared to competitors, Magento is quite secure [31]. According to CVE Details, an online security vulnerability data source, there was only one vulnerability reported in 2016, five in 2015 and two so far in 2017 [32]. According to the Magento security center, they published six security patches in 2016 and one so far in 2017. Figure 14 shows the percentages of the causes of these vulnerabilities.

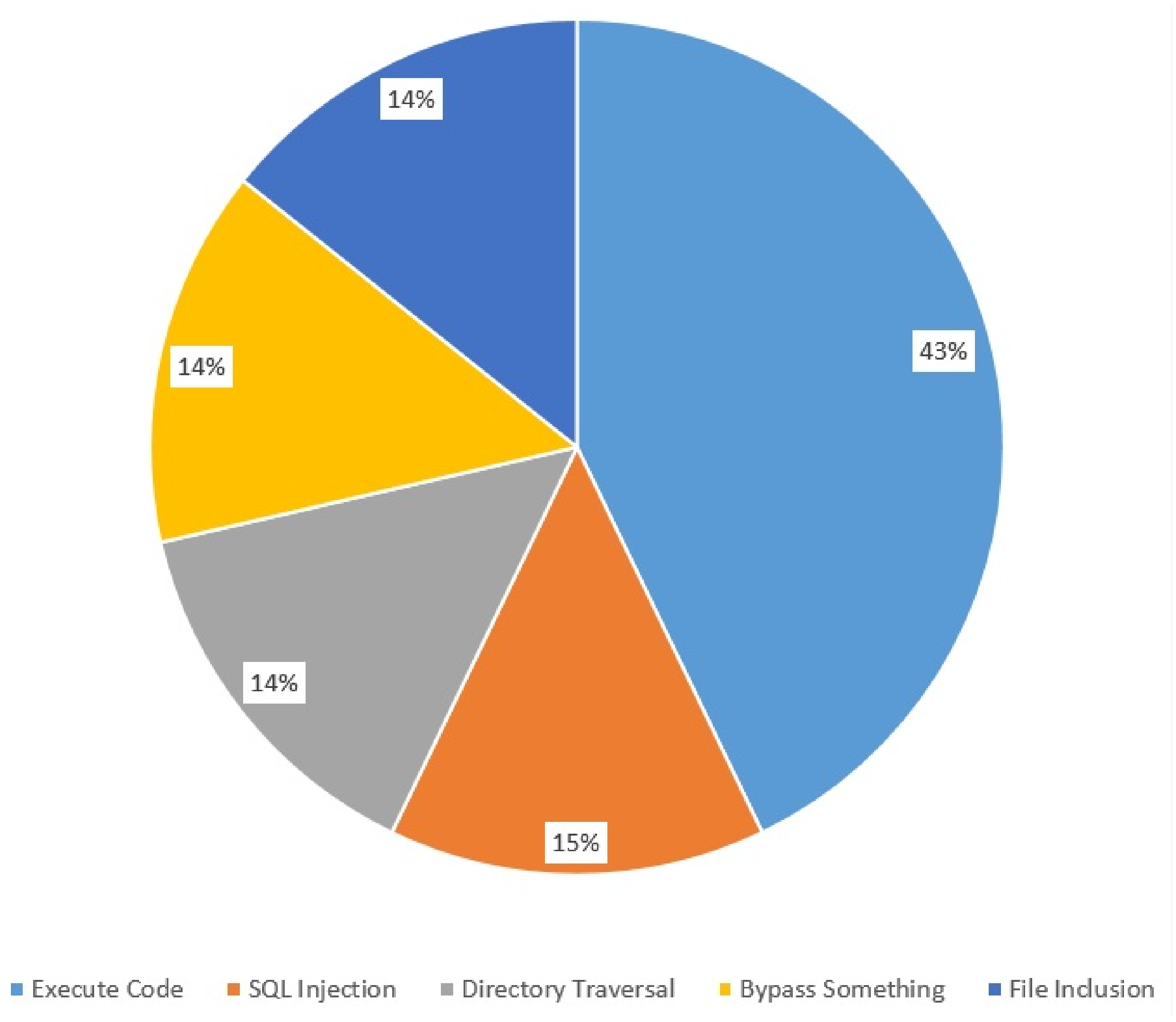


Fig. 14: Vulnerabilities by type

On the other hand, according to TrustWave's Global Security Report 2016, "Magento was the ecommerce target of choice for hackers, with Magento installations accounting for 85% of compromised ecommerce systems" [33]. This was mostly caused by weaknesses in the server environment. As illustrated in Figure 15, a Magento infrastructure contains several potential gateways for hackers [34].

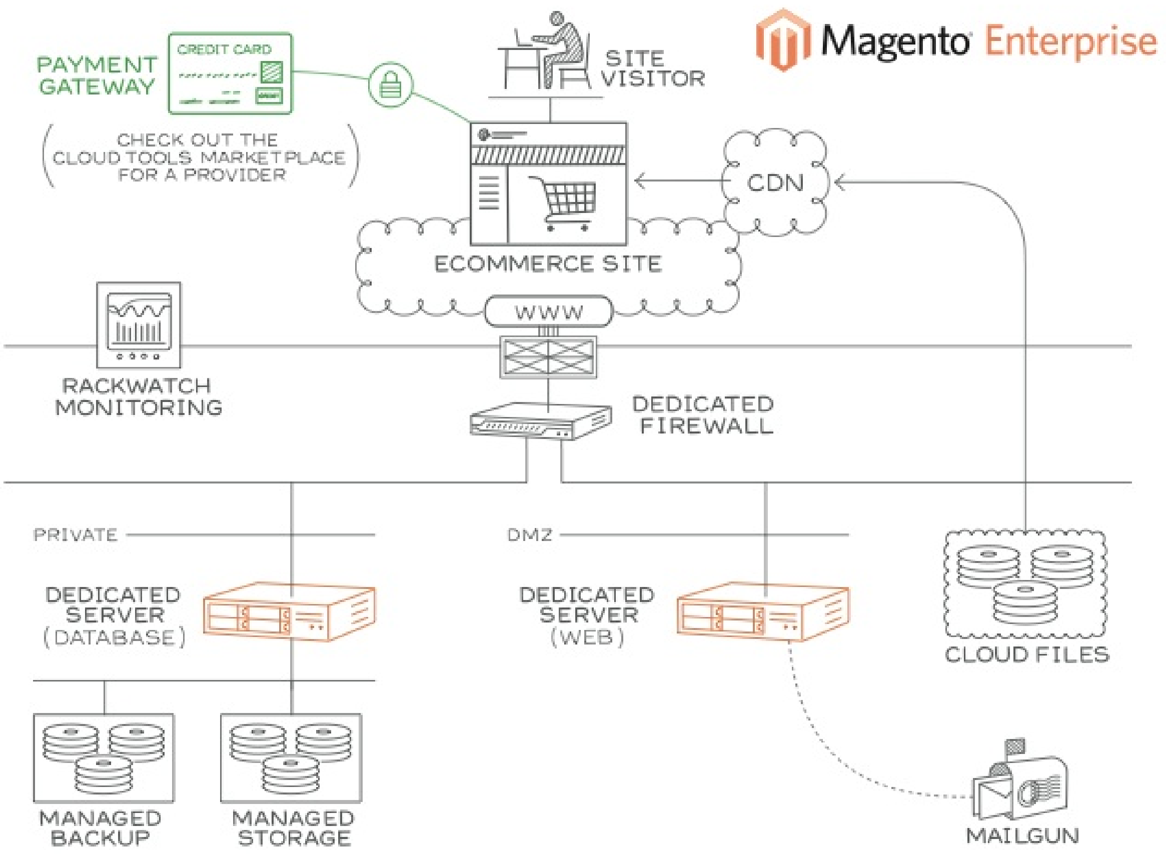


Fig. 15: A standard Magento infrastructure at Rackspace

TrustWave's report also notes that most software on Magento users' servers were outdated and not fully patched. Therefore, securing the server infrastructure is the most critical aspect of securing a Magento website. In order to do this properly the Magento team published a security guide which describes the best practices for server administrators to setup the third-party software correctly and securely.

Intrusion detection and containment

Magento provides multiple security measures for intrusion, detection and containment to prevent attacks. Table 2 shows the security measures for each phase.

Milestone	Measures for developers	Measures for users
Intrusion	Security fixes and patches are used to protect Magento.	Users should apply Magento's best practices for protecting their website and server environment.
Detection	Security tests, BugCrowd and the Magento community forum are used to keep track of detected breaches.	Monitor for signs of attacks using external tools that check the security status of your website. The most popular, Magereport, is developed by the Magento community.
Containment	Security updates are released in security patches with additional notes on their risk and applicable versions of Magento.	Users can sign up for security alerts and fixes to recover from attacks. To prevent future similar attacks follow Magento's Disaster Recovery Plan, also part of the security guide.

Table 2: Intrusion, detection and containment in Magento

According to TrustWave, it takes Magento 80.5 days on average from initial intrusion to a security breach being detected [33]. The median total duration between intrusion and containment decreased from 111 days in 2015, to 63 days in 2016. But this still leaves hackers nine weeks on average to attack Magento without the developers being aware of it.

Recovery from failures

Magento applies the following measures to secure their products [35]:

- **Enhanced password management**: Magento has improved the hashing algorithms (SHA-256) used in password management.
- **Prevention of cross-site scripting (XSS) attacks**: The Magento Framework follows protocols that manage the escape of data in output. These protocols allow clients to escape output for HTML pages (HTML, JSON, and JavaScript) and email. More information on measures against XSS attacks can be found in the Magento documentation.
- **Flexible file system ownership and permissions**: Since Magento 2.0.6., file system permissions are held in certain files that are writable in a development environment and read-only in a production environment. These permissions (particularly for production) can be further restricted using a umask, as explained in the Magento documentation.
- **Prevention of clickjacking exploits**: Magento protects your webshop from clickjacking attacks by using an X-Frame-Options HTTP request header. For more information, see X-Frame-Options header.
- **Use of non-default Magento Admin URL**: To prevent large-scale attacks that use automated password guessing and target default admin URL's like admin or backend,

Magento creates a random Admin URI when you install the product. This URI can be changed through the provided CLI. More information can be found in the Magento documentation.

Technical debt

Technical debt reflects on the long-term impacts of trade-offs that are taken during software development between productivity and maintainability. Since Magento is a very large system with nearly 200 external dependencies, technical debt is a serious threat that should be given a lot of attention.

Monitoring technical debt

In order to keep an overview of the system's technical debt, the Magento development team employs two automated testing tools.

Codacy static analysis

Codacy is a static analysis tool that performs automated code reviews and marks all violations. Based on the analysis results, the code is assigned a quality rating based on code style, security, duplication, code complexity and test coverage. Figure 16 depicts the current quality rating of the Magento development branch. The vast majority of warnings are low level issues on variable name lengths and naming conventions. The Codacy report indicates that Magento is quite healthy software project.

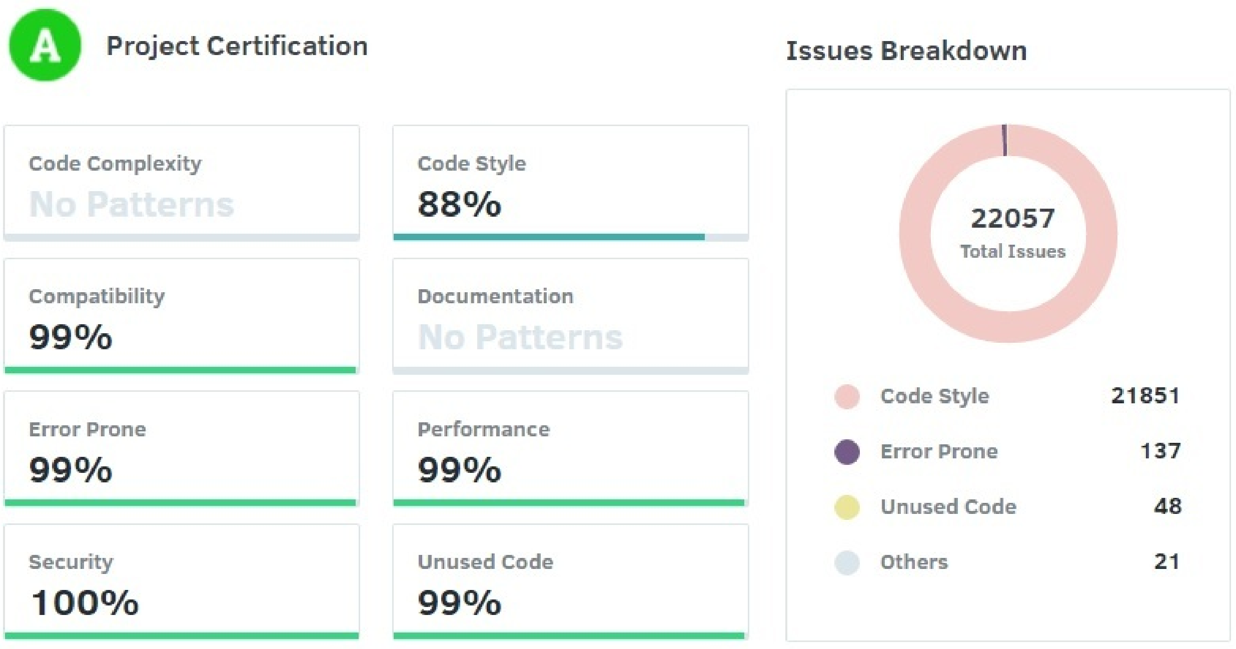


Fig. 16: Codacy Technical Debt overview

Travis Continuous Integration

Travis CI is used for performing automated builds of the latest version of the platform. Since PHP is an interpreted language, the Travis build does not actually compile the code, instead it runs an extensive test suite with different configurations. Since most webservers use different PHP versions, the entire test suite is repeated for the current stable PHP 5 and PHP 7 versions.

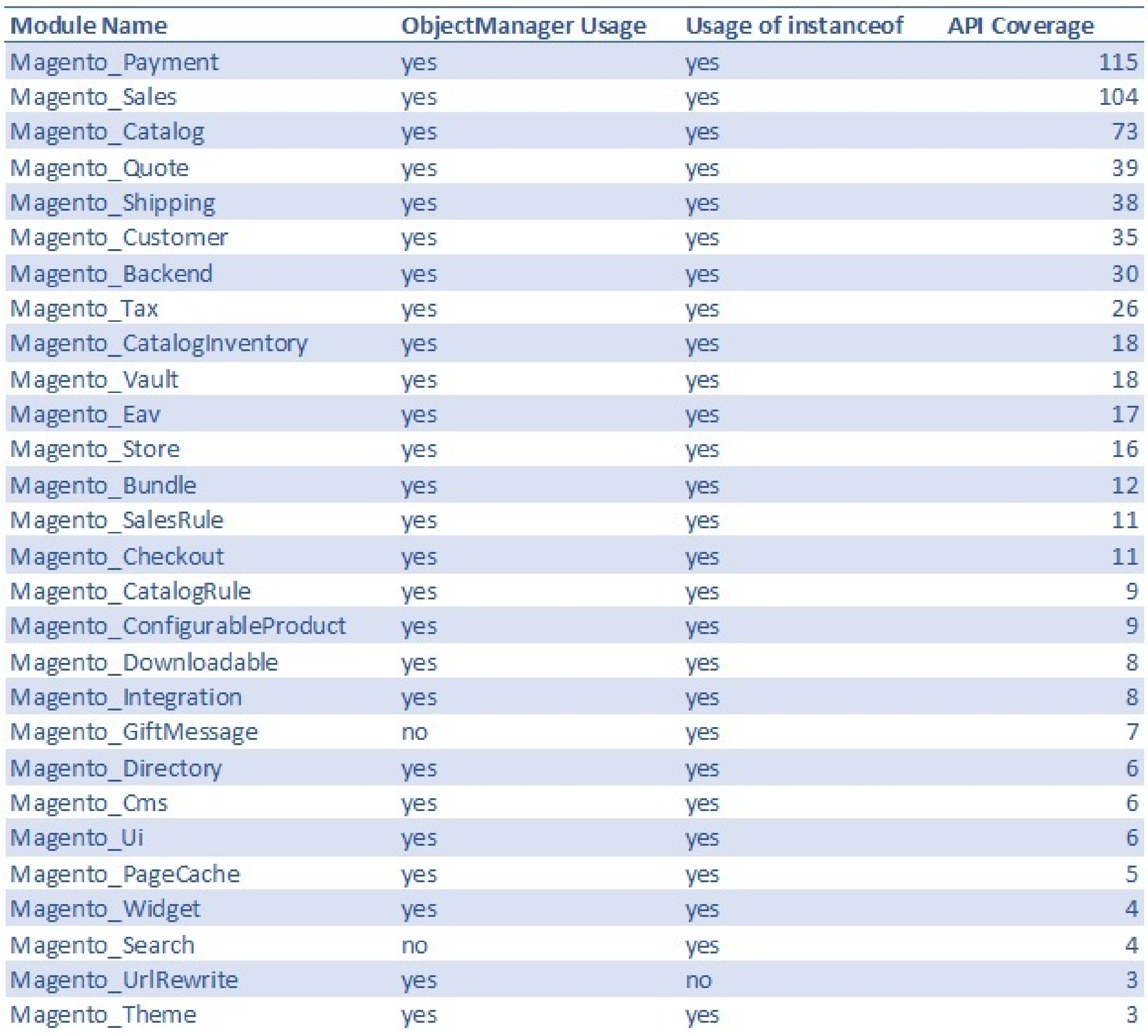
Currently, this is the outcome of an integration test for a typical commit:

> OK, but incomplete, skipped, or risky tests!
> Tests: 4783, Assertions: 12392, Incomplete: 69, Skipped: 77.

It indicates that there are a number of incomplete assertions and skipped test cases. This is generally not a good sign and indicates a source of technical debt.

Technical Debt Aggregation Tool

One developer from the Magento community developed the *Module Technical Debt Aggregation* tool that analyses the technical debt of each module in Magento 2 [36]. Figure 17 shows the output of the tool for 28 modules ordered by decreasing technical debt.



Module Name	ObjectManager Usage	Usage of instanceof	API Coverage
Magento_Payment	yes	yes	115
Magento_Sales	yes	yes	104
Magento_Catalog	yes	yes	73
Magento_Quote	yes	yes	39
Magento_Shipping	yes	yes	38
Magento_Customer	yes	yes	35
Magento_Backend	yes	yes	30
Magento_Tax	yes	yes	26
Magento_CatalogInventory	yes	yes	18
Magento_Vault	yes	yes	18
Magento_Eav	yes	yes	17
Magento_Store	yes	yes	16
Magento_Bundle	yes	yes	12
Magento_SalesRule	yes	yes	11
Magento_Checkout	yes	yes	11
Magento_CatalogRule	yes	yes	9
Magento_ConfigurableProduct	yes	yes	9
Magento_Downloadable	yes	yes	8
Magento_Integration	yes	yes	8
Magento_GiftMessage	no	yes	7
Magento_Directory	yes	yes	6
Magento_Cms	yes	yes	6
Magento_Ui	yes	yes	6
Magento_PageCache	yes	yes	5
Magento_Widget	yes	yes	4
Magento_Search	no	yes	4
Magento_UrlRewrite	yes	no	3
Magento_Theme	yes	yes	3

Fig. 17: Magento Module Technical Debt

The API coverage is particularly interesting, since it indicates the complexity and coupling of the various modules. The Payment module turns out to be the worst performing module. A further investigation confirms its complexity by the fact that this module consists of a staggering 272 files. Besides the Payment module, also the Sales and Catalog modules are high on the list. This shows that many parts of the website need access to the sales and catalog data. The *Module Technical Debt Aggregation* tool clearly indicates which modules need the most attention when reducing technical debt.

Testing debt

Testing debt is caused by the lack of testing or by poor testing quality. Testing is unfortunately still considered a luxury for many development teams. Luckily, at Magento, they realized this and significantly invested in automated testing.

Current code coverage

In order to measure how well the system is tested, we have used PHPUnit to generate a code coverage report in HTML format for Magento 2. The report can be found on our team website. The overall coverage of the modules in `app/code` is 37%. This is pretty low because these modules contain a lot of untested UI elements and auto-generated code, as illustrated in Figure 18.

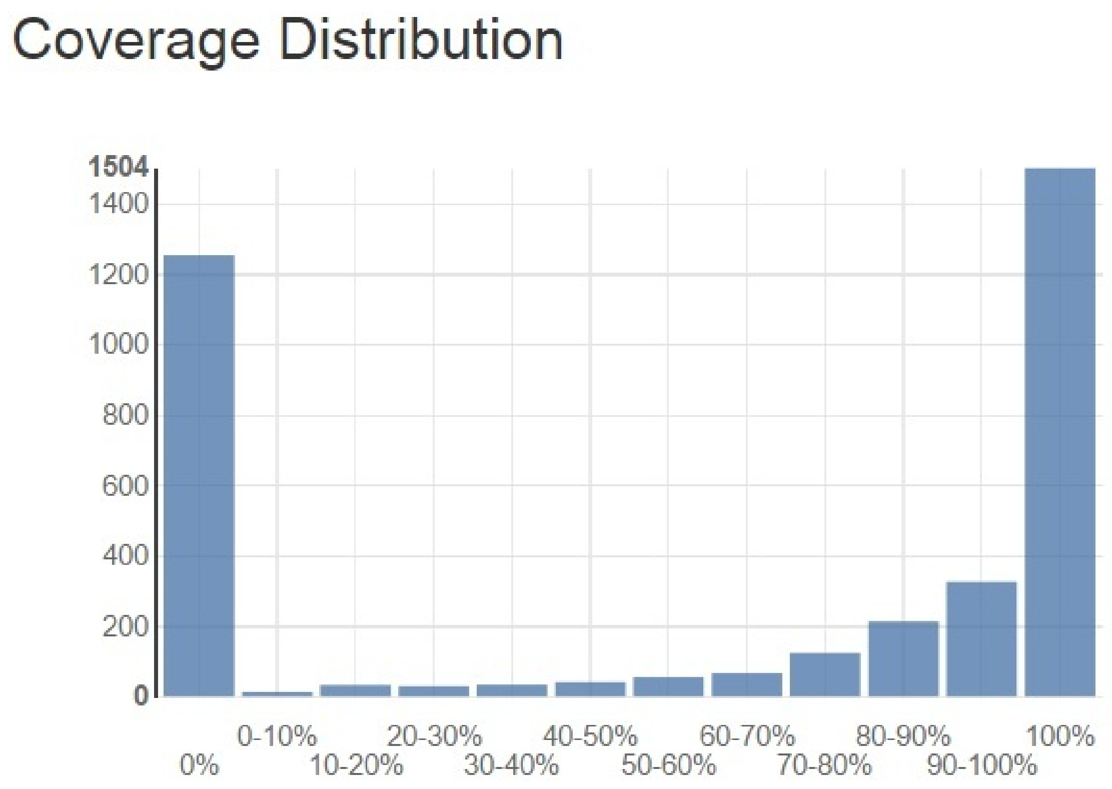
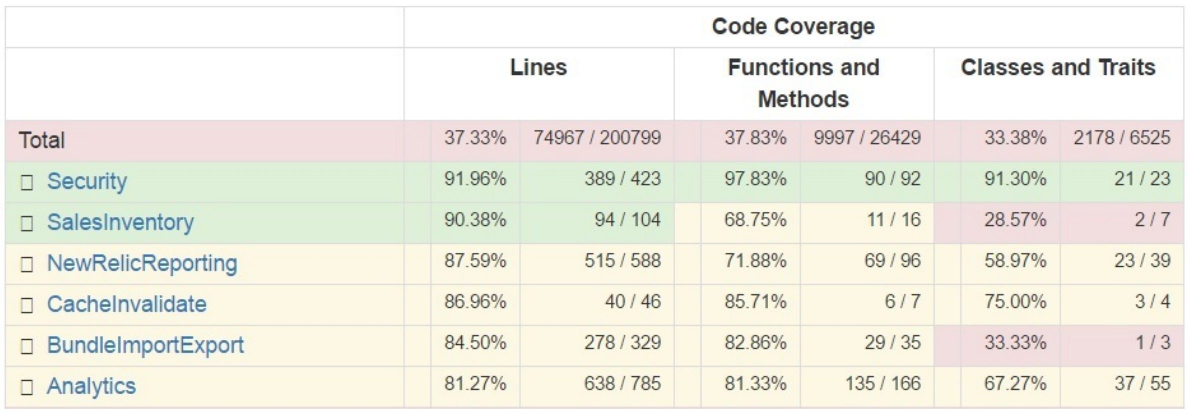


Fig. 18: Coverage distribution in Magento 2

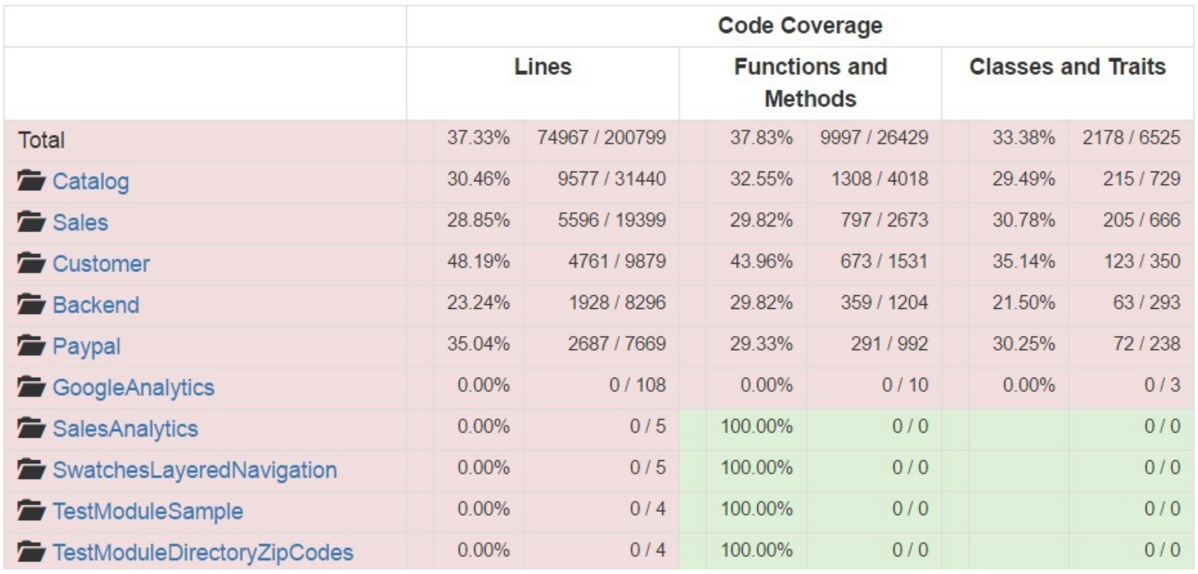
Magento's Testing standard states that new code must always be covered unless it is auto-generated code or does not contain new business logic. Therefore, each module's View folder is untested which drastically lowers the overall coverage. However, the majority of the modules has a well-tested Block, Helper and Model folder (with a coverage higher than 70%). If we leave out all View folders and auto-generated code, the code coverage ends up higher than 88%.



	Code Coverage					
	Lines		Functions and Methods		Classes and Traits	
Total	37.33%	74967 / 200799	37.83%	9997 / 26429	33.38%	2178 / 6525
Security	91.96%	389 / 423	97.83%	90 / 92	91.30%	21 / 23
SalesInventory	90.38%	94 / 104	68.75%	11 / 16	28.57%	2 / 7
NewRelicReporting	87.59%	515 / 588	71.88%	69 / 96	58.97%	23 / 39
CacheInvalidate	86.96%	40 / 46	85.71%	6 / 7	75.00%	3 / 4
BundleImportExport	84.50%	278 / 329	82.86%	29 / 35	33.33%	1 / 3
Analytics	81.27%	638 / 785	81.33%	135 / 166	67.27%	37 / 55

Fig. 19: The top 5 Magento modules with the highest code coverage

The modules that are best tested all have a coverage higher than 80%. Since modules shown in Figure 19 are responsible for Magento's core functionality, it is a good sign that they have been thoroughly tested.



	Code Coverage					
	Lines		Functions and Methods		Classes and Traits	
Total	37.33%	74967 / 200799	37.83%	9997 / 26429	33.38%	2178 / 6525
Catalog	30.46%	9577 / 31440	32.55%	1308 / 4018	29.49%	215 / 729
Sales	28.85%	5596 / 19399	29.82%	797 / 2673	30.78%	205 / 666
Customer	48.19%	4761 / 9879	43.96%	673 / 1531	35.14%	123 / 350
Backend	23.24%	1928 / 8296	29.82%	359 / 1204	21.50%	63 / 293
Paypal	35.04%	2687 / 7669	29.33%	291 / 992	30.25%	72 / 238
GoogleAnalytics	0.00%	0 / 108	0.00%	0 / 10	0.00%	0 / 3
SalesAnalytics	0.00%	0 / 5	100.00%	0 / 0		0 / 0
SwatchesLayeredNavigation	0.00%	0 / 5	100.00%	0 / 0		0 / 0
TestModuleSample	0.00%	0 / 4	100.00%	0 / 0		0 / 0
TestModuleDirectoryZipCodes	0.00%	0 / 4	100.00%	0 / 0		0 / 0

Fig. 20: The top 5 largest Magento modules and the top 5 Magento modules with the lowest code coverage

Figure 20 shows that the five least tested modules have 0% coverage. Further investigation confirmed that these modules only contain auto-generated code. The largest modules have a coverage around 30% and are mainly base modules (Customer, Sales, Catalog) which contain many view files.

Testing procedure

As explained in Magento's Testing Guide, the Magento 2 project distinguishes six different types of testing [37]. Each test type has different quality criteria that must be adopted by Magento developers in order to get their pull-requests merged. They can be found in the Magento2 git repository. Each PR will be measured by Codacy, Travis CI and integrators against the test level of the quality criteria. For each PR that includes new logic or new features another check by the core team is done to ensure adequate unit/ integration test coverage.

An improvement potential in Magento's testing procedure is the visibility of the code coverage in the continuous integration process. Currently, the contributors and integrators are expected to measure this on their local machine.

Evolution of technical debt

With the transition from Magento 1 to Magento 2, the platform inherited a part of Magento 1’s technical debt. Therefore, the development team started to work in a series of milestones to pay the technical debt and to build an architecture that better complies to the SOLID principles [38]. This whole refactoring process took a year. Here are the most significant improvements:

- **The ObjectManager**: This God class is probably Magento 2’s largest violation of the SOLID principles. The usage of ObjectManager is a bad practice since it increases the coupling between different modules or models. The development team is dedicated to decreasing its usage in the source code, however because of its deep roots in the architecture this process goes slowly [39].
- **Dependency injection**: Instead of the ObjectManager ("Mage" class in Magento 1) Magento 2 uses dependency injection to make modules independent of their dependencies [38].
- **Framework dependency**: Magento 1 was tightly wrapped around Zend 1 and highly dependent on it. To make Magento 2 [40] more independent, it uses its own adapter and interfaces for various frameworks.
- **Rewrite conflicts**: Magento 1 allowed modules to overwrite core functionality. Magento 2 changed this by introducing plugins [41]. Plugins do not overwrite the class, but make it able to make a call before or after a method.
- **Test automation**: With Magento 1 unit testing was not historically a requirement for developers. This was caused by the fact that Magento 1 was hard to test due to the static factory methods of the Mage class [42]. However, with Magento 2 the community was introduced to test automation and the Magento Automated Testing Standard. This standard was defined to unify different testing approaches developed by the community. Soon enough the code coverage improved and since 2015 the coverage of Magento has been stable above 75%.

To compare the technical debt in Magento 1 and Magento 2, we used the tool SensioLabs. In total Magento 1 had 655946 lines, which had 24347 violations compared to a total of 997592 lines but only 8421 violations for Magento 2. The report for Magento 1 and 2 can be found on the SensioLabs website, here and here, respectively.

Conclusion

We have analysed how Magento’s architecture offers flexibility and extensibility to its users. Magento’s design is based on a layered architecture with a clear separation in business and presentation logic. Many design patterns can be found in the core components. For purposes of flexibility, Magento employs a database layout based on Entity-Attribute-Value (EAV) models. In order to simplify access to the database, the platform offers the PHP Data Objects interface.

Our analysis of Magento's security showed that the platform's vulnerabilities are primarily caused by remotely executed code and weaknesses in the server environment. The team provides different security measures, such as security patches, monitoring tools and best practices.

Magento benefits from the improvements in technical debt that were introduced with the release of Magento 2. The development team is aware of the current technical debt and is dedicated to reducing it. The developers are actively decreasing the usage of the ObjectManager and often open pull-requests concerning code refactoring and documentation improvements.

In our analysis we have identified the visibility of code coverage and other software metrics as an improvement potential in Magento’s testing procedure. By making the measurement of these metrics part of Magento’s continuous integration process, feedback could be offered to developers more quickly, leading to a higher quality of code.

References

1. Ecommerce News. Magento the most used ecommerce platform in Europe. https://ecommercenews.eu/magento-used-ecommerce-platform-europe/.
2. Jodie Pride. WooCommerce or Magento for Your E-Commerce Store? https://www.ostraining.com/blog/wordpress/magento/.
3. Larry Morroni. The History of Magento and Its Rise to Ubiquity. https://morroni.com/blog/history-of-magento/.
4. Magestore. What is Magento 2? First step to build a Magento 2 site. http://www.magestore.com/magento-2-tutorial/what-is-magento-2/
5. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
6. Magento-hosting. https://www.yourhosting.nl/zakelijk/managed/magento-hosting/.
7. Magento Community. https://community.magento.com/.
8. Magento partners. https://magento.com/partners/portal/directory/?partner_type=1.
9. Magento. https://magento.com/.
10. Samy Felice. The numerous benefits of the Magento Ecommerce Platform. https://www.neotericuk.co.uk/benefits-magento-ecommerce-platform.
11. University of Southampton. Object Oriented Basic Concepts and Advantages. http://eprints.soton.ac.uk/250857/3/html/node3.html.
12. Magento. Global features that support extensibility.

http://devdocs.magento.com/guides/v2.0/architecture/global_extensibility_features.html.

13. Magento. Service contracts. http://devdocs.magento.com/guides/v2.1/extension-dev-guide/service-contracts/service-contracts.html.
14. Magento Testing Guide. http://devdocs.magento.com/guides/v2.0/test/testing.html.
15. Magento Layered Architecture diagram. http://devdocs.magento.com/guides/v2.1/architecture/archi_perspectives/arch_diagrams.html.
16. Magento Presentation Layer. http://devdocs.magento.com/guides/v2.1/architecture/archi_perspectives/present_layer.html.
17. Magento Service Layer. http://devdocs.magento.com/guides/v2.1/architecture/archi_perspectives/service_layer.html.
18. Magento Persistence Layer. http://devdocs.magento.com/guides/v2.1/architecture/archi_perspectives/persist_layer.html.
19. Introduction to Magento 2 — No More MVC http://alanstorm.com/magento_2_mvvm_mvc/.
20. R. Jaison. Six Benefits of Using MVC Model for Effective Web Application Development. https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application-development/.
21. Magento. PHP developer guide: Factory. http://devdocs.magento.com/guides/v2.0/extension-dev-guide/factories.html.
22. Max Pronko. Development Design Patterns in Magento 2. https://www.maxpronko.com/blog/magento-2-development-design-patterns.
23. Magenticians. Twelve design patterns in Magento. https://magenticians.com/12-design-patterns-magento/.
24. Magento 2.0. Prepare yourself for a new way of module development. https://www.slideshare.net/ivanchepurnyi/magento-20-prepare-yourself-for-a-new-way-of-module-development.
25. PHP The Right Way. http://www.phptherightway.com/.
26. Object-Relational Mapping. http://www.killerphp.com/articles/what-are-orm-frameworks/.
27. Magento. DevDocs: Persistence layer. http://devdocs.magento.com/guides/v2.0/architecture/archi_perspectives/persist_layer.html.
28. Magento. DevDocs: Technology stack. http://devdocs.magento.com/guides/v2.0/architecture/tech-stack.html.
29. Manish Prakash. Magento EAV database structure. http://excellencemagentoblog.com/blog/2011/09/07/magento-eav-database-structure/.
30. Alan Storm. Magento for Developers: Part 7—Advanced ORM: Entity Attribute Value.

http://devdocs.magento.com/guides/m1x/magefordev/mage-for-dev-7.html.

31. Brian Jackson. Complete Guide on Magento Security. https://www.keycdn.com/blog/magento-security/.
32. CVE Details. Magento: Vulnerability statistics. https://www.cvedetails.com/product/31613/Magento-Magento.html?vendor_id=15393.
33. ExtensionsMall. How secure is Magento? Not much, says Trustwave. https://www.extensionsmall.com/blog/how-secure-is-magento/.
34. John Engates. Building Secure, Scalable and Highly Available Magento Stores, Powered by Rackspace Solutions. https://support.rackspace.com/white-paper/building-secure-scalable-and-highly-available-magento-stores-powered-by-rackspace/.
35. Magento. DevDocs: Security overview. http://devdocs.magento.com/guides/v2.0/architecture/security_intro.html.
36. Vinai Kopp. Magento 2 Module Technical Debt Aggregation. https://github.com/Vinai/m2-tech-debt.
37. Alan Kent. Magento 2: Testing, testing and more testing. https://alankent.me/2014/06/28/magento-2-test-automation/.
38. Brideo. Being a SOLID developer in Magento 2. http://brideo.co.uk/magento2/SOLID-in-Magento-2/.
39. Alan Storm. Magento 2 Object Manager. http://alanstorm.com/magento_2_object_manager/.
40. Framework dependency in Magento 2. - http://www.coolryan.com/magento/2016/01/29/the-difference-between-magento-1-and-magento-2/
41. Magento. Developer Documentation: Plugins. http://devdocs.magento.com/guides/v2.0/extension-dev-guide/plugins.html.
42. Dan Homorodean. Beginning unit testing in Magento 1.x. https://magento.evozon.com/beginning-unit-testing-in-magento-1-x.html.

Mapbox GL JS

By Yoeri Appel, Lars Krombeen, Remco de Vos and Jos Winter.

Abstract

Mapbox GL JS is a JavaScript rendering library used to create interactive maps using WebGL. It is part of a large collection of open source tools created by Mapbox to design, develop and show (interactive) maps that are completely customised to suit the needs of the user. This chapter analyses the architecture of Mapbox GL JS using the context, stakeholders, development and information viewpoints, and could be used by future developers as reference to get started. Furthermore it provides an analysis of the technical debt present in the project.

Introduction

Human kind has been making maps for a extensive amount of time, with the first map dating back over 16500 years [2]. The way maps are made and viewed has changed a lot since then and is still changing today. Nowadays almost everyone has the possibility of accessing maps of anything anywhere over the Internet using computers or smart devices. As a consequence several companies and organisations have been formed around the services of providing maps and map data.

One of these companies is Mapbox which provides both geographic data, rendering clients, and other services related to maps. Mapbox GL JS is one of their open source client libraries for rendering and interacting with maps for websites. As part of the Mapbox ecosystem it of course integrates with the other services Mapbox provides.

As the Mapbox GL JS client is part of a much larger ecosystem this chapter first looks at the Mapbox and the relation between Mapbox and Mapbox GL JS. This is done by describing the context of Mapbox GL JS and its primary stakeholders. Furthermore Mapbox GL JS's architecture is discussed by describing its module structure, standardisation of design and testing, and build approach. Additionally, as maps are data heavy the flow of this data will be discussed in the information view as well as the usability for both developers and users. Lastly, the technical debt for the Mapbox GL JS library will be discussed to see what the quality of the project is.

About Mapbox and Mapbox GL JS

The Founding of Mapbox

Mapbox was founded in 2010 with the goal to provide an alternative to the popular Google Maps. In Google Maps little to no customisation was possible and there were barely any tools for cartographers to create maps how they envisioned it. Mapbox was founded with the goal to change that and provide (open source) tools for cartographers and developers to create the maps they desired.

All tools Mapbox develops are open source and free to use. Their core business is hosting services (such as storing and providing the user's custom geo-data) on their servers. Their goal is to ensure that software exists which supports the digital cartographers in the best way possible, but they do understand that not everybody needs and likes the same and that alternatives to their tools do exists. Since it is not their goal to make the best or most popular tools themselves, they learn from and work together with these alternatives to improve their own tools and let the users use, for example another data renderer, while still using the other Mapbox tools and services. This is the reason that there is not one Mapbox repository or project but every tool/library/specification is developed separately to keep everything modular. Mapbox GL JS is one of these modules and is one of the renderers developed by Mapbox that visualise your geo-data.

Mapbox GL JS and the Mapbox Architecture

An important part of Mapbox GL JS' context is its position and role in all the other Mapbox components. This section will give a general explanation of all relevant Mapbox components in order to better understand the role of the Mapbox-gl-js repository.

Mapbox's system revolves around Styles, tilesets and data sources. The relation between these components can be viewed in figure 1.

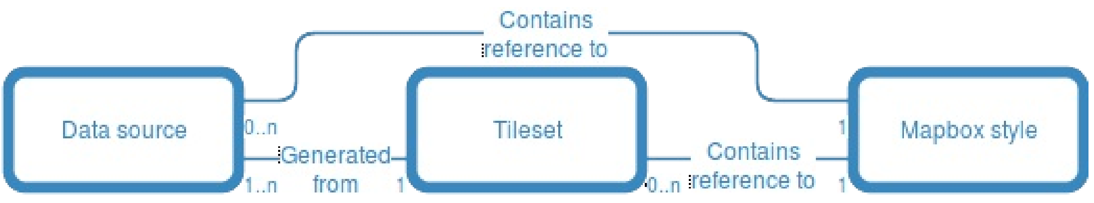


Figure 1. Diagram showing relation between Data sources, tilesets and styles in the Mapbox architecture

A tileset is an optimised way to save and transport data by splitting it in tiles. A Mapbox style defines where to find the data sources (in raw form or the optimised tilesets) and how to display this data. This allows the user to visualise every bit of data exactly the way they want (or not visualise it: hiding certain parts). Mapbox created tools to both create these styles and to render them with different methods. The most basic rendering tool is the static API which converts the style into a static image by running an algorithm on their server. Another way is to use the Mapbox plugin for Leaflet, which uses the basic browser techniques like svg and canvas to render the data. Furthermore, there is the Mapbox-gl-native repository which uses OpenGL to render the map and contains SDKs for Android, iOS, macOS, NodeJS and Qt. The last method is using WebGL inside a web browser and is developed in Mapbox GL JS repository.

Mapbox GL JS itself is a map interaction and rendering client for websites and web applications and thus does not provide the map data for rendering [3]. Within the Mapbox ecosystem the Maps API and Styles API provide respectively the raster and vector tilesets, and the Mapbox GL styles needed for the web rendering client [4]. When the Mapbox GL JS client is used completely within the Mapbox ecosystem it has a dependency on the Maps and Styles APIs. However, this is not necessary, as long as the data sources follow the specifications it is also possible to use other external or private sources [5]. However, since they earn their money with their servers, Mapbox promotes the use of the data they are providing on their services, which can be complemented with data the users uploads to the server on their accounts, and can be accessed simultaneously with the Mapbox data via the APIs mentioned before. The data Mapbox provides is gathered from both the leading open source and largest commercial providers, like OpenStreetMap, NASA and DigitalGlobe, as can been seen on their website. Figure 2 gives a graphical overview of the these relations.

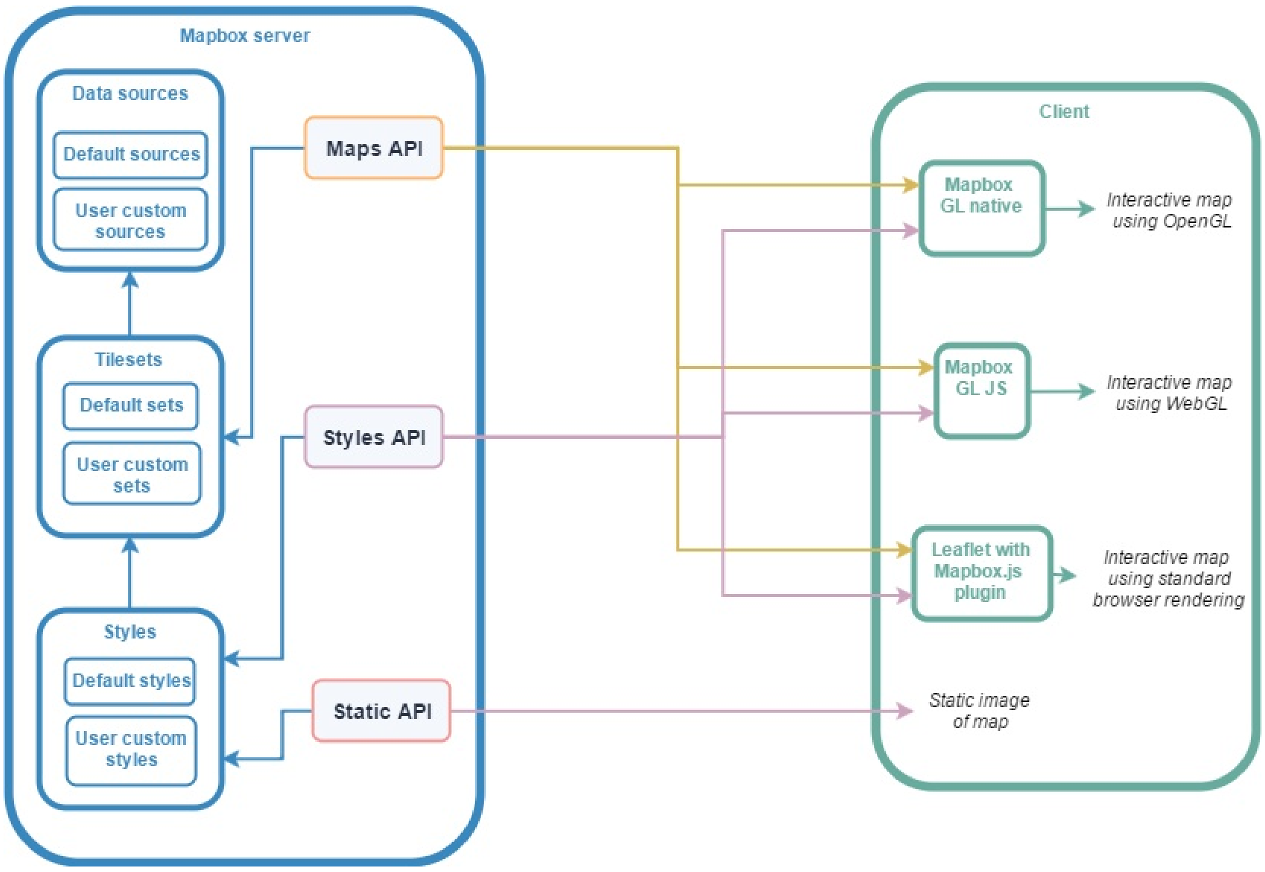


Figure 2. Simplified overview of the major parts of the Mapbox architecture related to Mapbox GL JS

Context View

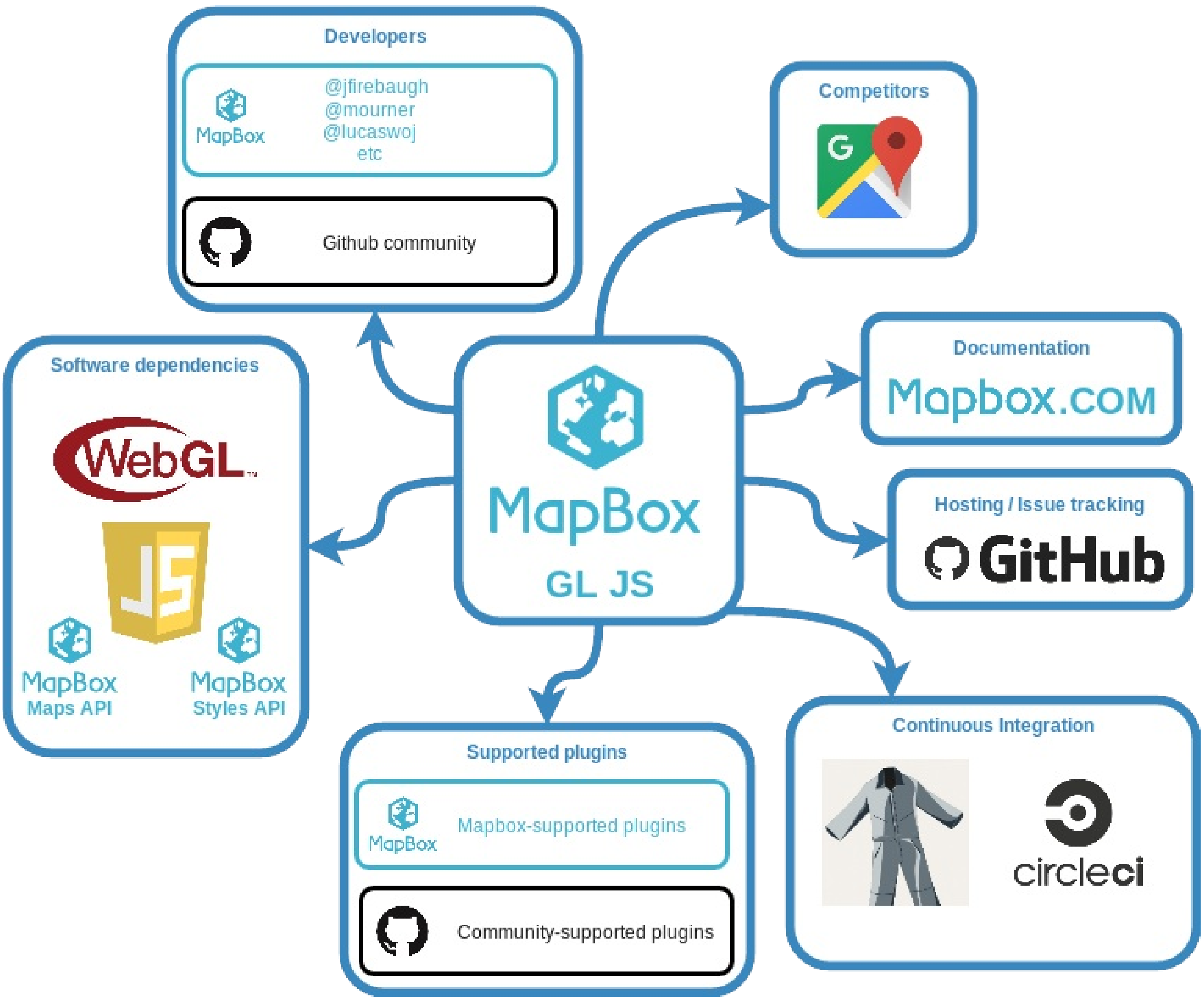


Figure 3. Diagram of the context view

As mentioned before, Mapbox develops as much as possible in open source projects and therefore the mapbox-gl-js repository is one of 500+ repositories in the Mapbox organisation. Consequently Mapbox GL JS is developed by both developers from Mapbox and people from the Github community who want to contribute. More information about which Mapbox employees are working on Mapbox GL JS can be found in the Stakeholders section. Github as the host for the code, forms a medium for people to contribute to the code and is used to track issues as well.

Just as all other Mapbox projects, Mapbox GL JS is developed by and developed for other developers. Mapbox GL JS is structured in such a way that plugins can be created to extend the default functionalities. Both Mapbox themselves and people from the Github community have created these plugins and they are promoted on their website.

The programming language used to build Mapbox GL JS is JavaScript and WebGL (the web version of OpenGL) is used to render the map inside the client's browser. Mapbox's Styles API and Maps API are used to determine what should be rendered, how it should be rendered and retrieve the data that needs to be rendered, as described in the previous section. This impacts the architecture heavily, because the architecture of a large JavaScript system is different than a strongly typed language like Java and the architecture is designed around the APIs and WebGL.

Mapbox' documentation is hosted on their site and Coveralls and CircleCI are used for continuous integration, the use of these specific tools does not impact the architecture, but using continuous integration in general helps to make sure that the quality of the code is at a certain level.

Due to Mapbox philosophy of creating open source tools that can be used by anyone, the Mapbox GL JS project does not have many competitors. When there is a tool that is an alternative to one of the Mapbox tools, this is not seen as a threat. This is shown by the fact that Mapbox collaborates with multiple companies that provide these tools in order to improve their own tools and/or create plugins for these tools so people can still use other parts of Mapbox with the alternative tools. In some extreme cases they even drop their own project and hire the developer of the alternative tool, like they did with Leaflet [9, 10]. Google is the only company that was found that has tools alternative to the Mapbox tools, but does not support anything of Mapbox, since Google Maps is not modular and Google wants people to only use their product. This is why we consider Google Maps as the only real competitor of Mapbox GL JS.

Mapbox GL JS is making use of a package manager (Yarn right now, npm in the past) to track dependencies and allow developers to easily install and test the code. Yarn and some other development dependencies (which are small and not really impactful to the architecture and therefore left out of this section) depend on NodeJS and Mapbox GL JS also uses some features of NodeJS during the development. This is why Mapbox GL JS also has a strong dependency on NodeJS.

Stakeholders of Mapbox and Mapbox GL JS

This section describes the relevant stakeholders of Mapbox GL JS. According to Rozanski and Woods [1] there are 10 different types of stakeholders. The most important types indentified for Mapbox GL JS are the developers and users. However, since Mapbox GL JS is part of the larger organisation Mapbox the stakeholders of Mapbox will be briefly mentioned, but the main focus will be on the Mapbox GL JS project. An overview of all the stakeholders can be seen in figure 4. Most stakeholders are within the organisation: e.g. Mapbox employs a team for the support of users. The teams as mentioned in figure 4 can be found on the Mapbox team page. Other relevant stakeholders of Mapbox which are important for Mapbox GL JS are the investors, grouped under the acquirers type together with the CEO of Mapbox.

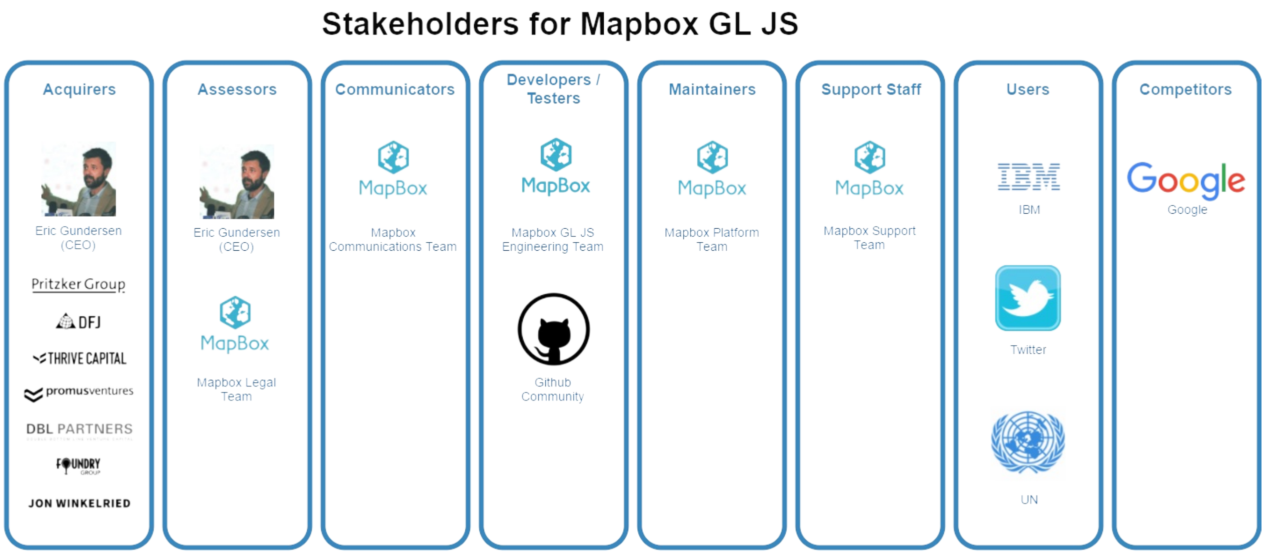


Figure 4. Stakeholders of Mapbox

The rest of the section focusses on Mapbox GL JS. The users of Mapbox GL JS are JavaScript developers that want to use the Mapbox plugin on their website. The gallery shows some usage examples. Their showcase includes some Mapbox customers and which industries Mapbox is powering.

Within the Mapbox GL JS project there is a hierarchy between the developers. Firstly, there is the open-source community that can create issues, making Mapbox aware of problems, and can propose pull request to fix issues. The open-source community has made significant contributions to the project. Secondly, there are the developers that pick up issues or work on existing projects and create pull requests when they finished their task. Lastly, there are the integrators: the developers that review and merge pull requests. These integrators are responsible that the code works and that the changes in the pull request are in compliance with the project standards. The integrators are the assessors of the team.

A summary of the identified stakeholders for Mapbox GL JS in addition to the stakeholders of Mapbox can be found in table 1.

Type	Stakeholders
Developers	Github open-source community, @jfirebaugh, @lucaswoj, @mourner, @anandthakker, @mollymerp, @tmcw, @ChrisLoer
Assessors/Integrators	@jfirebaugh, @lucaswoj, @mourner, @anandthakker, @mollymerp
Users	Thousands of users world-wide (showcase) e.g. IBM, Twitter, MasterCard, The World Bank, Runkeeper, The Guardian, Airbnb

Table 1. The main stakeholders of Mapbox GL JS

The Architecture of Mapbox GL JS

Development View

An important part of a software system is the software development environment as this has influence on the design, build and testing. Especially, for complex systems it is important that this has been set up correctly to maintain and guarantee productivity and quality. The development view studies code structure and dependencies, test and build and deployment management, design constraints, and code conventions. This section will take a look at these aspects for Mapbox GL JS.

Module structure model and organisation

In this section the different modules and their dependencies of the mapbox-gl-js repository will be discussed.

Madge, a developer tool for generating a visual graph of your module dependencies, was used to help determine the dependencies between the different modules. Since Mapbox GL JS is build on NodeJS, technically every file is a module, but the resulting graph was too big be displayed on one screen. Modules in different layers with similar abstraction layers were grouped together to generate a more comprehensible graph. The layers are largely based on the folders in which the source files are grouped.

- **User Interface** This layer is on top and contains all classes that interact with the user of the map, e.g. the code to zoom in or rotate the map.
- **Style** The style layer contains all classes that represent and process the Mapbox stylesheets.
- **Render** The render layer contains all classes that are responsible for rendering the geo-data on the screen using WebGL.
- **Mapdata** The mapdata layer contains all classes for representing the data that needs to be rendered.
- **Data types** The datatypes layer contains classes that are specialised datatypes that are used in the other parts of the system.
- **Style-spec** The style-spec layer was originally a separate repository but was merged into mapbox-gl-js. It defines the specifications that the Mapbox style created by the user has to satisfy to be considered valid.
- **Utility** The Utility layer contains all classes that provide general utility to the other parts of the system.

An overview of the layers and its dependencies can be found in figure 5.

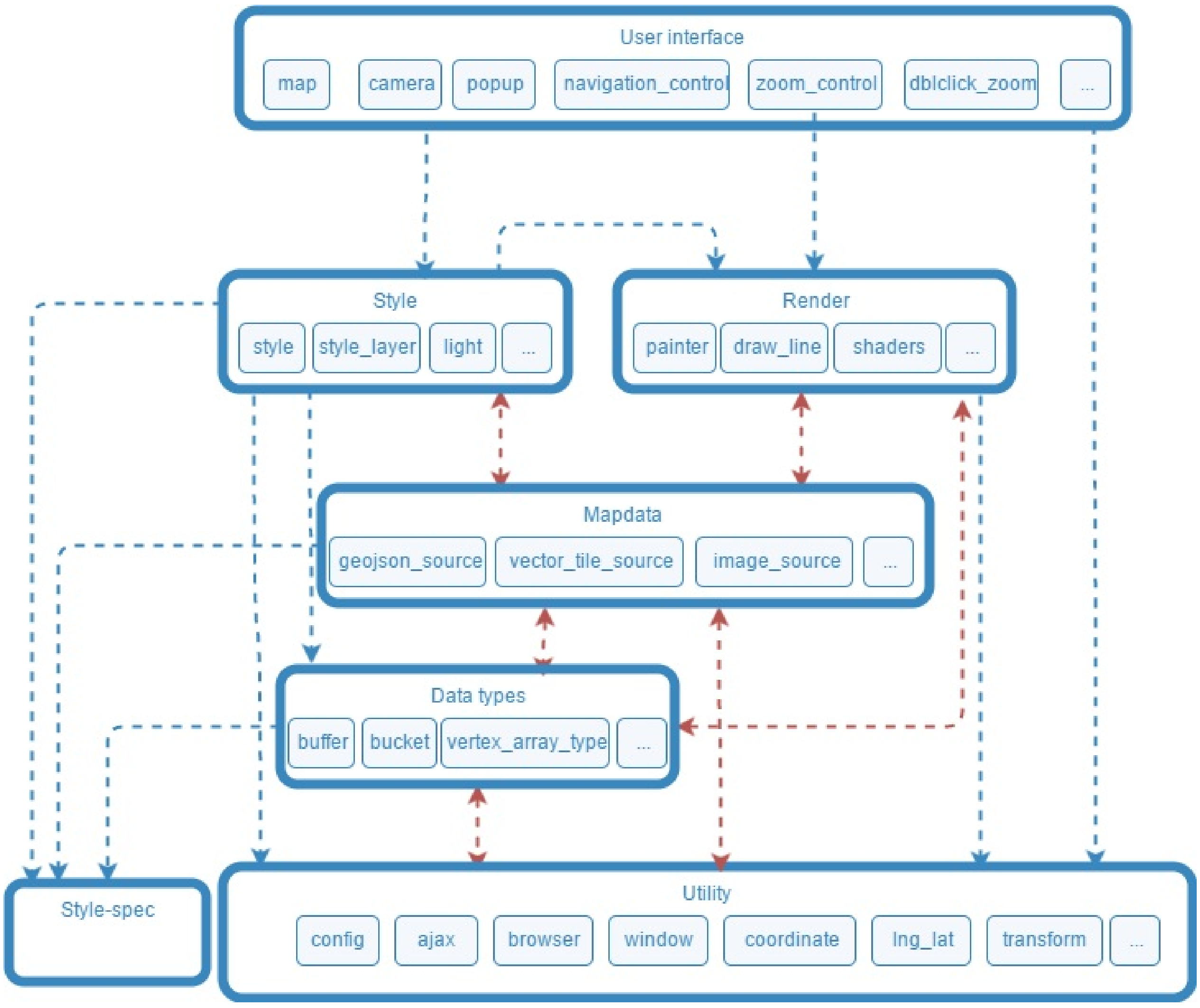


Figure 5. The module structure of Mapbox GL JS

Standardisation of Design

Because multiple software developers are influencing the Mapbox GL JS system it is important to standardise the key aspects of the design of the software to make it as maintainable, reliable and technical cohesive as possible. In the long run this decreases the technical debt and therefore the development time of future features and bug fixes in the system. Design standardisation can be achieved by using design patterns in the software and by standardising the process and communication around the software development.

Code Contribution Conventions

Not all Mapbox design standards are directly described. There are some design standardisation rules concerning the software mentioned in the contributions guide [6]. These design standards provided in the contributions guide mostly define functional standards which influence the functionality of the system for its users and these rules can be found in table 2. Furthermore there are conventions documenting the contributed code [7] and these rules are also displayed in table 2.

Specified in	rule
CONTRIBUTION.md	`error` events are used to report user errors instead of the standard `Error` class. However, the `Error` class is used to indicate non-user errors.
CONTRIBUTION.md	`assert` statements are used to check for invariants that are not likely to be caused by a user error. These `assert` statements are automatically stripped out of production builds.
CONTRIBUTION.md	A certain set of ES6 features are used so the system is functional on all the predefined platforms/browsers. The most notable used features which are important to the design standard are usage of classes and usage of computed and shorthand object properties (A detailed list of these features is described in the contributions guide).
CONTRIBUTION.md	Another set of ES6 features are not to be used, in order to maintain support for the predefined platforms/browser which also consists of older browsers. This may change in the future. Some notable features that may not be used are default parameters, REST parameters and iterators (A detailed list of these features is described in the contributions guide).
CONTRIBUTION.md	The Mapbox GL JS developer should use rebase merging as opposed to basic merging to merge branches
CONTRIBUTION.md	Use the Github label labeling system as specified in CONTRIBUTION.md
docs/README.md	PRs only only containing documentation improvement should be made towards the special mb-pages branch instead of master
docs/README.md	The documentation should follow the JSDoc rules [11]

Table 2. Overview of the rules regarding code contribution conventions

Architectural- and Design Patterns

Due to the dynamic typed nature of JavaScript and the absence of a standard way to implement interfaces and abstract classes a lot of design patterns cannot be used. The most central architectural pattern used in the system is the model-view-controller pattern which divides the application into three interconnected parts. The system is divided in the following parts:

1. The `Map` instance which contains information about the camera position and the data

that has been loaded into the map. (Model)
2. The UI handlers which update the map based on its functionality and the actions of the user. (Controller)
3. The active html canvas and css code active in the browser of the user. (View)

The observer design pattern is being used in the `Dispatcher` class, an instance of this class can broadcast to all subscribed `WorkerSource` instances in its pool. There also has been some discussion in the Mapbox repo about using the factory design pattern to create new objects which makes it easier for beginners, less prone to errors, and less verbose. To decrease the amount of duplicated code a lot of standard functionality is being reused and has been implemented in several utility classes which can be found in the util folder but also in the symbols folder.

Most of the SOLID principles cannot be used due to the absence of interfaces and abstract classes. The one principle that can be used is the single responsibility principle. The single responsibility principle is being used in the `Map` class which extends the `Camera` class which extends the `Transform` class. These classes inherit from each other to divide the functionality and the responsibilities between the classes. The `Map` class is responsible for the functionality which makes it possible to programmatically change the map and firing event when users interact with it. The `Camera` class is responsible for managing the animations and movement which are called by the user and the system. The `Transform` class is responsible for managing the position and the other camera options such as the pitch, the zoom and the bearing of the map.

Standardisation of Testing

The Mapbox GL JS repository contains several different and important test suites to ensure consistency and quality. This section will discuss the different tests, data and tools used by the team. The tests can be found in the folder tests which contains subfolders for different groups and types of tests. Furthermore there are several conventions tests must cohere with, which can be found in the test readme.

Test Suites

There are two different test suites associated with the project which are both run with yarn. The first one is `yarn test` which runs the quick unit tests, the second is `yarn run test-suite` which runs the integration tests. These two test suites consist of running several other more specific test suites. Tables 3 and 4 illustrate the different test suites in the mapbox-gl-js repository, their purpose and their dependencies on other test suites.

Test Suite	Purpose	Dependencies
test	Quick unit tests as well as syntax checking using lint and type checking using Flow.	test-unit, test-plugin, test-flow
test-suite	Integration tests for testing and validating the output of combined functionality: e.g. validate that a specified style generates a correct static image map.	test-render, test-query

Table 3. Mapbox GL JS test suites

All the test suite scripts are defined in the scripts section of the `package.json` and can be run with yarn.

Test Suite	Purpose	Tested Source
test-plugin	Runs the test in the test/plugins folder using the Tap framework.	The test verifies that the `docs/_data/plugins.json` is valid JSON.
test-unit	Runs all the tests in the test/unit folder using the Tap framework. The tests are unit tests which test functionality of the tested classes.	Individual classes in the src/data/, src/geo/, src/source/, src/style/, src/style-spec/, src/symbol/, src/ui/ and src/util/ folders.
test-render	Runs the `test/render.test.js` which runs `test/integration/lib/render.js`. Renders PNG's from the input and compares these to an expected PNG.	Different combinations of source files for combined behaviour.
test-query	Runs the `test/query.test.js` which runs `test/integration/lib/query.js`. Generates JSON based on the input and compares these to an expected JSON.	Different combinations of source files for combined behaviour.
test-flow	Runs the Flow to check for type mismatches.	All the files which are marked as needed to be type checked.
test-cov	Uses the nyc framework(#test-nyc) to create a test coverage report of the test-unit, test-render and test-query test suites.	See the individual test suites.

Table 4. Mapbox GL JS test scripts

Test Data

The test/integration/ folder contains all the data and information for the render and query integration tests. In integration/lib/ the files `render.js` and `query.js` can be found. These classes take all the subfolders in the integration/render-test/ and integration/query-test/ folders respectively and use these to create the tests and input data for testing, and obtain the expected result for comparison. Each subfolder is a specific group of tests which relate to each other or a function. Each of these groups again has subfolders for the specific test input data and expected results.

The other folders in the test/integration/ folder form the data input for some of tests as well as return data for created mocks.

Testing Tools and Infrastructure

Mapbox GL JS makes use of several testing tools for performing their tests. These include libraries for checking the code, providing functionality for mocking, testing framework and external infrastructure as part of continuous integration.

Testing Tools

The team uses several different testing tools to ensure code quality and correct functionality. In table 5 below the different testing tools and their purpose are further described. A distinction between two categories can be made for the used testing tools: testing libraries and code quality libraries. The libraries Tap, nyc, and Sinon.js are included and used for testing purposes and expanding the functionality of testing. Additionally, the static code analysis tools Flow and node-lint are included for checking and enforcing code quality.

Testing Tool	Testing Purpose	Tool Information
Tap	Tap is the Test-Anything-Protocol library for Node.js and provides a framework for writing and running tests.	www.node-tap.org
Sinon.js	Sinon.js is a library used to augment the standard testing object with the use of spies, stubs and mocks. At the end of each tests the spies, stubs and mocks on global objects are restored in the way the testing framework is setup [5].	www.sinonjs.org
Flow	Flow is a static type checker for JavaScript which uses type interference and type annotations.	www.flowtype.org
node-lint	Node-lint is a Node.js package which makes it possible to run JSLint from the command-line and is used for syntax validation.	www.github.com/jpolo/node-lint
nyc	nyc is Istanbul's command-line interface and is used for generating test coverage reports.	www.github.com/istanbuljs/nyc

Table 5. The testing tools used by the Mapbox GL JS team

Continuous Integration

CircleCI is the platform used for continuous integration. Each pull request triggers a minified and development build, and the `test-flow` and `test-cov` test suites are run. These tests also generate a test coverage report. As part of the checks of a pull request, passing all tests on CircleCI is a requirement for the pull request to be merged. The coverage report is send to Coveralls, a platform used for keeping track of test coverage statistics relating to the repository, files, lines of code, and coverage statistics over time. Each pull request thus results in a code coverage report, however, this is not required for a pull request to be approved. Because the developers are actively following their testing standards we can observe in Coveralls that the already high testing coverage has been slowly increasing from 85% to 89% in 2015 and 2016.

Build Approach

There are two build approaches provided for Mapbox GL JS: running the local build scripts with yarn or npm, and automated builds based on tagged releases.

Build Scripts

There are several different build scripts defined in the mapbox-gl-js repository. The build-dev and build-min build the application from the source files. Table 6 illustrates the different build scripts, their purpose and their output.

Build Script	Purpose	Output
build-dev	Builds a development version of the repository from src/index.js	dist/mapbox-gl-dev.js
build-min	Builds a minified version of the repository from src/index.js	dist/mapbox-gl.js
build-benchmarks	Builds the benchmarks in the repository bench/benchmarks.js	bench/benchmarks.js
build-docs	Generates the documentation of the repository.	Outputs to the docs/ directory.
build	Runs the build-docs build script.	

Table 6. Mapbox GL JS build scripts

The builds for the code use browserify to bundle all the single Node.js files working with require into one single file which can be used by webbrowsers. Additionally, the build-dev and build-min scripts run a test script with Tap to validate that the files have actually been build.

Automated Build and Deployment

The CircleCI configuration file is setup to trigger a deployment script, `ci-scripts/deploy.sh`, when there is a tagged release in the GitHub repository. The script is triggered when there is a passing build on CircleCI, which in turn requires the build and all tests to pass.

The deployment script in turn runs the build-dev and build-min scripts which create respectively the development and minified build of the repository. In turn these are uploaded to Amazon Web Services in a folder for that release and are available on the Mapbox website. In total four different files are uploaded: the minified mapbox-gl.js, mapbox-gl.js.map, the development build mapbox-gl-dev.js and mapbox-gl.css.

Information View

Rozanski and Woods define the information view as a description of the way that an architecture stores, manipulates, manages and distributes the data of the system. This section will target how data is stored, accessed and eventually how the data flows through the Mapbox GL JS plugin. The focus will mainly be on the flow of the geo-data and stylesheet in the system and until they are rendered as the entire system revolves around this data.

Mapbox GL JS is executed on the clients website. The JavaScript developer can add data like vector tiles to the map or change the style of the map by changing the stylesheet. If the JavaScript developer does not want to use a custom style they can use a default one. This data and other data is defined and stored on the client's website. The data that is eventually used depends on two API calls from the Mapbox servers. The APIs provide the data of the map that is to be rendered and it contains stylesheet templates that the JavaScript developer can use if he defined the use a default style. Figure 6 shows an overview of the main flow of data. The flow starts when the JavaScript developer makes the call to Mapbox GL JS and ends when the map has been created and is returned. It shows where the data is stored, retrieved, manipulated and managed.

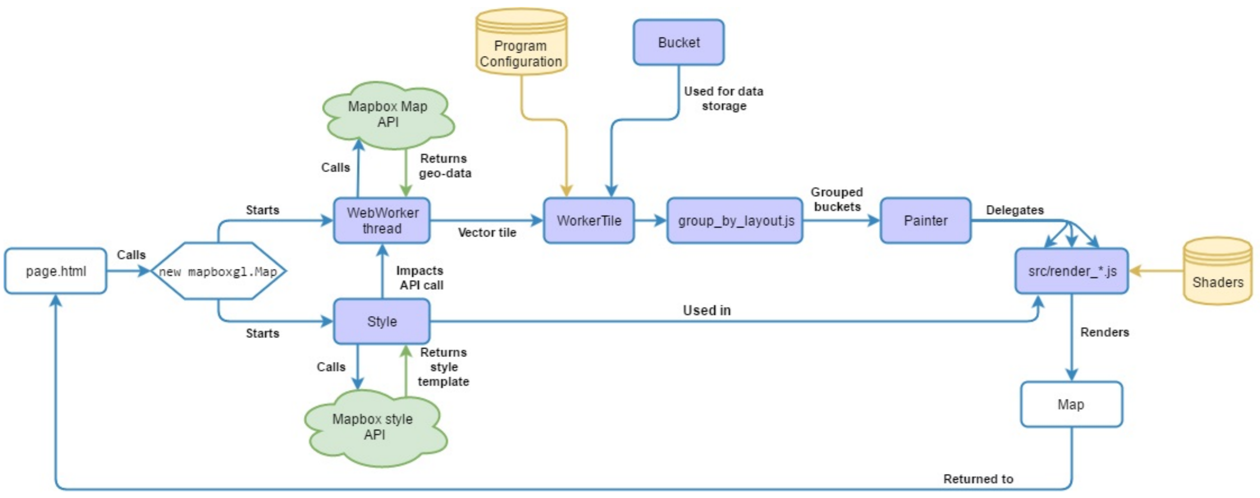


Figure 6. Overview of the flow of data in the Mapbox GL JS code. Yellow cylinders indicate local data storage, blue boxes indicate classes that are used and green clouds indicate external data which is fetched using an API call.

The flow makes clear that the data is stored at two local places and on the Mapbox server, and is manipulated using a simple flow. The progam configuration handles that the tiles and style layers are applied to the map. This way the data that the JavaScript developer defined is added to the map using a custom defined style if the JavaScript developer defined it.

The initial call is made from the page which instantiates a Mapbox object after which two flows are created which are combined at the rendering steps and are as follows. The first flow is responsible for fetching the geo-data. This is done by making a call to the Mapbox Map API. The API call is different based on the defined style. Once the tiles are loaded the tiles and the style which the developer defined are added to the vector tiles. The vector tile data is modified in different classes as can be seen in figure 6. The classes are part of the Mapbox GL JS plugin and should not be modified by the JavaScript developer. The results that are passed to the layer responsible for rendering consists of vector tiles which are grouped by their program configuration. The second flow fetches a stylesheet from the Mapbox Style API and communicates it with the first flow to make the proper API calls for the map data. If the user indicated that he wants to use a stylesheet the flow fetches the style and passes it to the renderer. Once the flows are combined the rendering layer renders the vector tiles group by group and applies the fetched and defined styles to the map. Additionally, the renderer uses shaders which are locally stored but the shaders should not be modified by the JavaScript developer. The map is now rendered and is returned to the client web page which displays the map.

Usability Perspective

Applying the usability perspective on the information view ensures that the system allows the users that interact with it do so effectively [1]. The usability perspective focuses on the end users of the system and thus addresses the concerns of JavaScript developers that have to work with the Mapbox GL JS plugin. The usability perspective can be applied on the information view by looking at the quality of the information e.g. the provision of accurate, relevant, consistent and timely data [1].

The developers are only concerned with the stylesheet for their website. The website can use a default style sheet which is given by Mapbox or add their own data sources which can be called using `style: 'mapbox://styles/mapbox/light-v9'`. Adding own data or changing the style can easily be done by following the API or examples provided on the Mapbox website. Once instantiated the map will be fetched automatically for the website using an API call.

The developers that make use of Mapbox GL JS are also concerned with the concerns of their website users. The website users are the people that can interact with the rendered map. The website user requires that the map is rendered quickly when he interacts with the map. Additionally, the map should be a correct representation of the real world. The website users are not concerned with the style since the website provides this. They are, however, concerned with the geo-data which is retrieved from OSM. The data from OSM can be modified by anyone and therefore erroneous data can slip in. This is a similar feature that Wikipedia has and has minimal wrong data since people correct each other and prevent people from changing data to something that is incorrect.

Figure 6 has shown that the data is going through a channel where it is transformed or used and passed to the next class. This makes it easy for a Mapbox GL JS developer to see what happens where and what he has to change if he wants to add new functionality or debug the program. However, the JavaScript developer is not concerned with which classes are called since he only calls the constructor of the map.

The way the information is sent through the code makes it usable for the JavaScript developers and the website users. The only downside for the website user is that the data can be incorrect. The performance is determined by the website which can add as much data as required. When the JavaScript developer adds too much custom data the rendering can become slow which lowers the usability.

Technical Debt

In this section the technical debt of the mapbox-gl-js repository is discussed. The technical debt was determined using both software tools and inspecting files manually. Additionally, the documentation and testing debt will be covered. Lastly, the pull requests and issues on Github will be analysed to see if the developers are recognising, discussing and managing the technical debt in the repository.

SonarQube Results

The SonarQube tool estimated the technical debt to be 7 days. The technical debt is defined as the amount of development hours it will take to fix all the found issues related to security, reliability and maintenance. However, the measure of time is not the best way to define the technical debt since 7 days is relatively low if the project contains 500k lines of code. An other metric called technical debt ratio is defined as the ratio between the actual technical debt and the effort it would take to rewrite the whole source code from scratch [8]. The technical debt ratio for Mapbox GL JS is 0.8% which indicates that the technical debt is relatively low and managed properly.

Documentation Debt

The API documentation consists of all the documentation, specifications and examples necessary for developers to start using Mapbox GL JS and can be found on the website. In contrast to the API documentation, the documentation for the rest of the source code seems to be rather lacking. Although most of the classes have documentation for the class definition (although for some this is also missing), a lot of methods have little to no documentation. For new developers trying to contribute to the project itself this is an issue as it takes a lot of time to figure out how the code works, what it does, and what it is supposed to do. However, the developers of Mapbox GL JS are planning on writing about their architecture.

Testing Debt

Using the already active code coverage tools in the repository the code coverage for the Mapbox GL JS code was evaluated. The testing efforts are prioritised and focused on testing the most important and core classes. Examples of these important classes are the `Camera` and `Map` class which are vital to the functionality of the system and which are frequently changed. Some less important parts of the code aren't well tested which increases the testing debt of the system. Some code is less important because hardly any issues and bugs are detected and fixed in it and therefore the code doesn't change frequently. However, the testing debt on this less important code is mitigated because of the unimportance of those classes and methods. The developers and testers of Mapbox GL JS prioritise their testing efforts on the most important classes and methods of the system and regression testing when fixing bugs. The testing debt in the Mapbox GL JS project can be decreased by creating more tests for uncovered parts of the code. In the Mapbox GL JS project there is already an existing restriction that when new functionality is added the developer should also create accompanying tests.

Evolution of Technical Debt

Using the same tools used in the SonarQube Tool Analysis section, technical debt can also be measured by analysing the separate releases of Mapbox GL JS.

There have been 33 major releases of Mapbox GL JS before March 2017. The technical debt grew gradually over time, but this is expected since the repository grew as well. The maintainability rating was always rated with an A, therefore it can be concluded that the technical debt grew proportionally with the code.

Discussion about Technical Debt

Technical debt is not necessarily a bad thing as long as the developers are aware and it is managed. After looking at the discussions in some of the Github issues/pull requests and searching the source code for certain keywords that would indicate unfinished or bad code, it can be concluded that the developers of Mapbox GL JS are discussing technical debt and leave little to none unfinished/bad code behind. However, not all the technical debt that was found is discussed and the debt that is discussed was discovered using SonarQube. Of course, technical debt is not an exact measure and there might be more discussion on technical debt outside GitHub, which cannot be seen. However, from the things which could be seen, it can be concluded that the technical debt is managed well and they especially focus on keeping the code/variable names consistent and clear.

Conclusion and Recommendations

In this chapter the mapbox-gl-js repository and its architecture were analysed. Firstly, the philosophy of Mapbox, the purpose of Mapbox GL JS, its context view and the stakeholders are discussed. After that architecture of mapbox-gl-js is discussed with analyses of the modules, testing methods, build approach and the information flow through the system. Lastly, the amount of technical depth was analysed. Based on all the analyses we have the following recommendations for the Mapbox GL JS developers concerning their project:

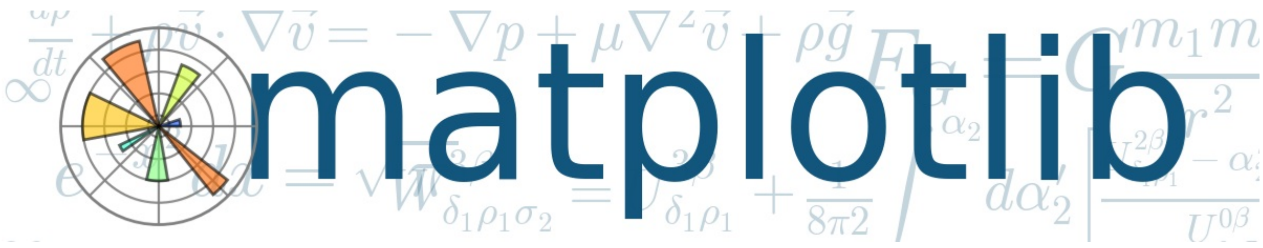
- Better documentation of the architecture and code should be added to the repository to ensure that new developers easily understand the architecture and how they could best contribute. The developers recently started on an architecture document which they should keep refining.
- Continue with prioritising the testing efforts to manage the testing debt. Any time that is left could be used to test the less important untested or badly tested code. If the testing standards are being following the coverage will improve over time similar to the past few years.

References

1. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. Ice Age star map discovered: http://news.bbc.co.uk/2/hi/science/nature/871930.stm (Visited: Februari 2017)
3. Mapbox GL JS fundamentals: https://www.mapbox.com/help/mapbox-gl-js-fundamentals/ (Visited: March 2017)
4. Glossary: tileset: https://www.mapbox.com/help/define-tileset/ (Visited: March 2017)
5. Mapbox awesome-vector-tiles: https://github.com/mapbox/awesome-vector-tiles (Visited: March 2017)
6. mapbox-gl-js contributing: https://github.com/mapbox/mapbox-gl-js/blob/master/CONTRIBUTING.md (Visited: March 2017)
7. mapbox-gl-js readme: https://github.com/mapbox/mapbox-gl-js/blob/master/docs/README.md (Visited: March 2017)
8. SonarQube Documentation: https://docs.sonarqube.org/display/SONAR/Documentation (Visited: March 2017)
9. Leaflet Creator Vladimir Agafonkin Joins MapBox https://www.mapbox.com/blog/vladimir-agafonkin-joins-mapbox/ (Visited: April 2017)
10. Announcing MapBox.js 1.0 with Leaflet https://www.mapbox.com/blog/mapbox-js-with-leaflet/ (Visited: April 2017)

Matplotlib - The Python 2D Plotting Library

By Andreas Maruli C Pangaribuan, Helmiriawan, Rizky Dharmawan, and Sindunuraga Rikarno Putra.
Delft University of Technology



The Matplotlib logo taken from their website

Abstract

Matplotlib is a data plotting library that is often considered as the grandfather of visualization libraries in Python. The project is pioneered by scientist John Hunter back in 2003 and has since been developed by the open source community. This chapter aims to offer an outside point of view of the Matplotlib software architecture. At first, an analysis of the external entities of the system is given. Next, a closer look at the underlying architecture of Matplotlib is presented. Lastly, a deeper analysis into the system is done to identify technical debts that exist in Matplotlib and some possible solutions.

Table of Contents

1. Introduction

Matplotlib provides a convenient tool for creating 2D plots of data using Python. It is an open-source project that is fully supported by the Python scientific computing community. Matplotlib is now used by a variety of people for analysis and research purposes and was even used by NASA for data visualization of the Phoenix spacecraft exploration on Mars.

Matplotlib was initially developed by a group of self-taught programmers with a scientific background. The founder of Matplotlib, John Hunter, started developing it to visualize biomedical data during his post-doctoral research in Neurobiology [1] to visualize electrocorticography (ECG) data. At that time, the available proprietary data visualization tool was a limited resource, and John Hunter wanted to find an alternative tool that was available for all his team. MATLAB was the most popular alternative at that time, but he found that as a software program, it has limited capabilities in managing large biomedical data from various sources. Therefore, he opted to create a new tool that is similar to MATLAB using Python, which as a programming language has an advantage in handling data.

Matplotlib eventually evolved from a small project by a group of researchers into a widely used open-source tool. It is now one of the core component for the scientific Python stack. Matplotlib being a project that was initially developed by people without vast experience in software engineering makes its architecture and evolution interesting. The goal of this chapter is to analyze the architecture of this interesting project and present an outside overview of the system. To analyze the architecture of Matplotlib, we will refer to the guidelines made by Nick Rozanski and Eoin Woods [2].

2. Stakeholders

To start off our analysis, we will look at the stakeholders involved with Matplotlib. A stakeholder is an entity of a system architecture that consists of an individual or an organization that has importance and interest to realize a system. Figure 1 shows a summary of identified stakeholders of Matplotlib based on the classification proposed by Nick Rozanski and Eoin Woods [2].

Overview

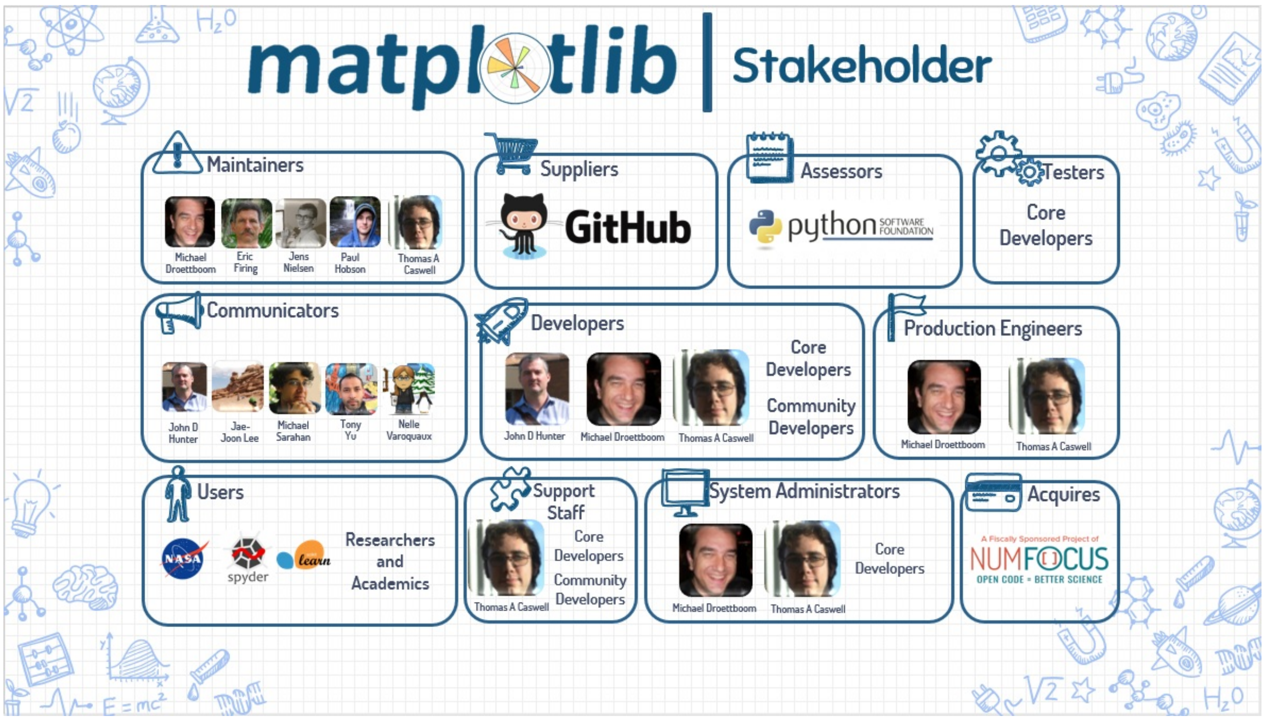


Figure 1. *The Stakeholders of Matplotlib*

There are no corporate sponsors for Matplotlib, but the main funding is provided through donations via the NumFOCUS organization, which can be classified as an **Acquirer**. The Python Software Foundation (PSF) can be categorized as an **Assessor** of Matplotlib because they oversee the system's conformance to standards and legal regulation. Meanwhile, the development of Matplotlib is coordinated via GitHub, which can be classified as a **Supplier**.

The **Developers** of Matplotlib are divided into core developers and community developers. John D. Hunter is the founder and initial lead developer of Matplotlib, but he passed away in 2012. Now both Michael Droettboom and Thomas A. Caswell act as lead developers. They are accompanied with 15 other core developers in developing Matplotlib. Many of the core developers also act as **Maintainers**, since they review all contributions to make sure it doesn't break the system. Matplotlib's **Support Staff** consists of a few core developers such as Thomas A. Caswell, Paul Hobson, and Eric Firing, and also some community developers, such as Nelle Varoquaux.

The **Users** of Matplotlib are varied as it is a generic data plotting tool, but based on depsy, quite a lot of them are from the research and academic background. Some large projects also use Matplotlib, such as NASA and Spyder. Some Python libraries such as `scikit-learn` and `seaborn` are also users since they use parts of Matplotlib inside their library.

Users can read the documentation to find out how to use Matplotlib, which is written by the **Communicators**, mainly John D. Hunter, Jae-Joon Lee, Michael Sarahan, Tony Yu, and Nelle Varoquaux.

Apart from the proposed definition above, there are also **Event Organisers** who organize annual events. An example is the SciPy conference, which gathers people who develop open source scientific projects using Python. In such events, the participants not only showcase their latest projects, but also learn and collaborate with other developers.

Power Interest Grid

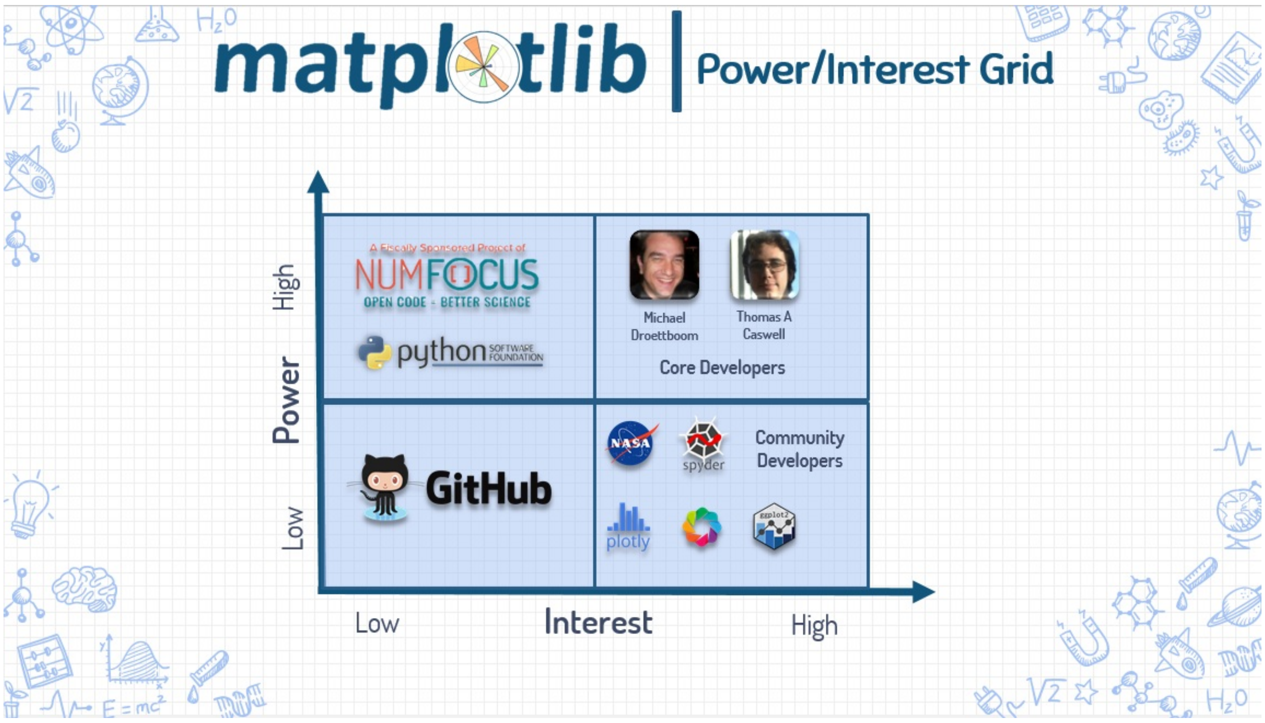


***Figure 2.** Power-Interest Grid*

The power versus interest relation of the stakeholders are shown in Figure 2 and are classified into 4 categories :

- **Low power and low interest** : GitHub is a stakeholder who does not has any control over Matplotlib, and does not has a significant role in the development of Matplotlib.
- **Low power and high interest** : Users, competitors, and community developers of Matplotlib are stakeholders who follow the latest development of Matplotlib and are active in the discussion of Matplotlib, but they do not have significant power to directly change the Matplotlib system.
- **High power and low interest** : NumFOCUS and the Python Software Foundation (PSF) are stakeholders who directly affect the development of Matplotlib. NumFOCUS though only provides funding without restricting the development of Matplotlib, while the

PSF develops Python which in turn directly affects Matplotlib itself. For both stakeholders, Matplotlib is only one of the few projects they affect and are not their main interest.

- **High power and high interest** : Michael Droettboom and Thomas A. Caswell are the lead developers of Matplotlib. Along with other core developers, they have significant interest and power in developing Matplotlib.

3. Context View

Next we will look into the boundary that separates between Matplotlib and its environment, a.k.a the system's runtime context. This describes what the system does and doesn't do, and how the system interacts with external entities [2].

3.1. System Scope

According to the introduction of Matplotlib at its website [3], Matplotlib is defined as **"a library for making 2D plots of arrays in Python"**. The scope of the software is clearly defined here. It is constrained to focus on one task, which is 2D plotting, and on a specific platform, which is on the Python programming language. From the history of its development, we can also conclude that Matplotlib was designed to be a free plotting library that is easy to use out of the box and is mainly targeted at the academic community.

The Matplotlib website [3] also states the design philosophy which John Hunter held to when developing Matplotlib.

> Matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn’t need to instantiate objects, call methods, set properties, and so on; it should just work.

This is further elaborated by a list of requirements that John Hunter had in mind when looking for a visualization tool.

> When I went searching for a Python plotting package, I had several requirements:
>
> - Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
> - Postscript output for inclusion with TeX documents
> - Embeddable in a graphical user interface for application development
> - Code should be easy enough that I can understand it and extend it
> - Making plots should be easy

3.2. External Entities

The connections between Matplotlib and its environment is summarized in Figure 3.

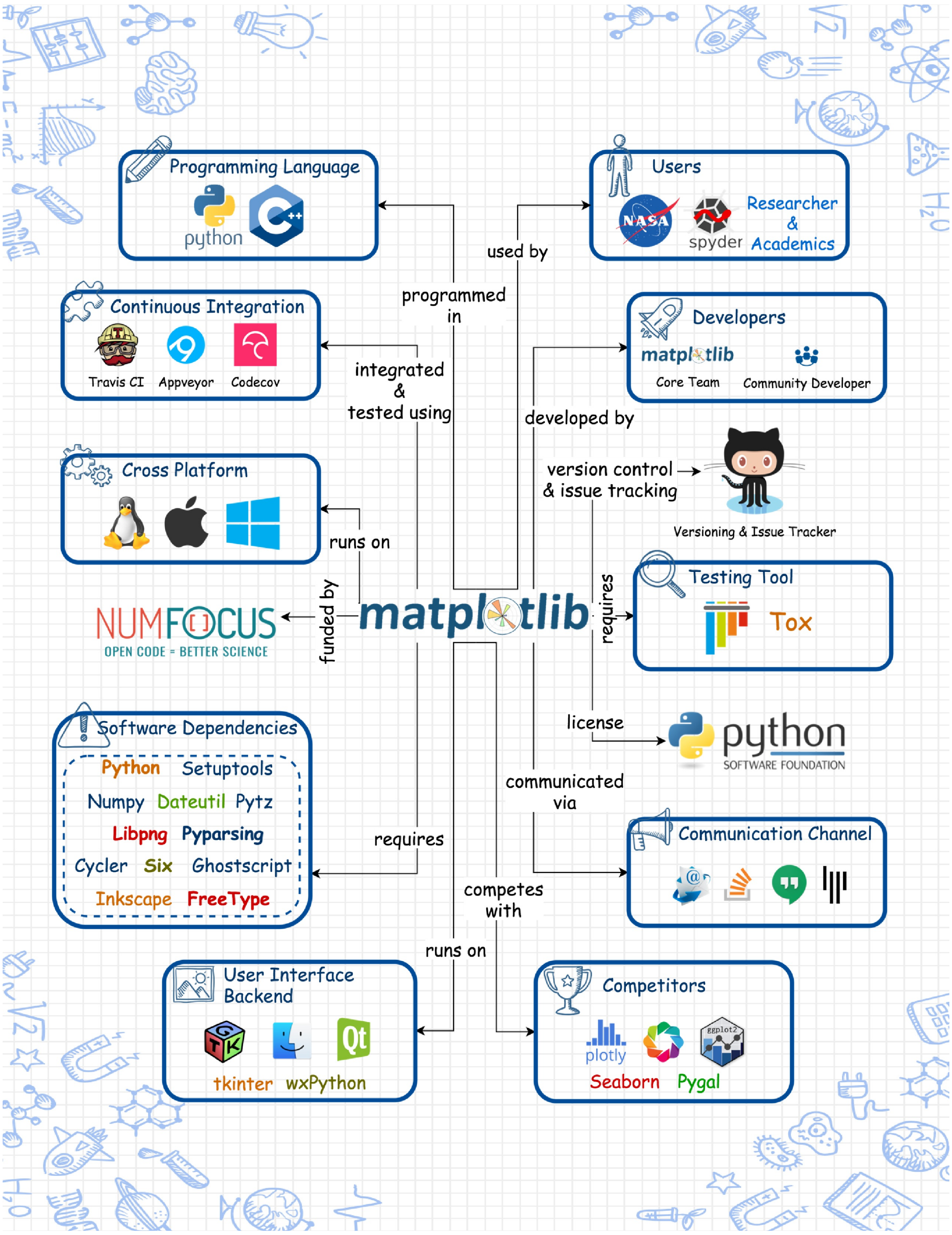


Figure 3. *Context View Diagram*

The external entities can be split into three groups:

A. Competitors

When Matplotlib was initially created, there was no other reliable Python visualization library available. But now Matplotlib faces quite a few competition for visualizing data in Python. The competitors are mainly split into two types:

Data plotting tools are the same as Matplotlib and want to make plotting data easier for people. Most of these tools extend Matplotlib such as `seaborn` that builds on top of Python by adding more beautiful default plot designs, and `ggpy` that adopts new features from the `ggplot2` R package.

Modern visualization tools on the other hand try to tackle newer visualization problems out of the scope of Matplotlib. An example is `bokeh` and `plot.ly` that both focuses on generating interactive visualization that can be embedded on the web.

B. Software Platform & Dependencies

Matplotlib is developed as a Python package and is hosted on the Python package repository (PyPI). A small part of the Matplotlib code is also written in other languages such as C++ and objective C for low level optimization.

Platforms : Matplotlib is well tested for compatibility on all three major operating systems (Windows, Linux, and OSX). Matplotlib enables image generation across platforms by supporting most UI rendering platforms (such as linux's `gtk` and OSX's `macosx`) and image formats (`jpg` , `svg` , etc).

Dependencies : Most dependencies are standard libraries such as `pyparsing` and `dateutil` , and also image format libraries such as `libpng` . Two libraries which Matplotlib has strong dependencies on are `numpy` which is used for numerical operations and `six` for backward compatibility with Python 2. A complete list can be found on the Matplotlib website [3].

C. Development & Community

A platform that plays an important role during development is `GitHub` , which is used for code versioning, issue tracking and project management. All code gets tested and integrated via continuous integration tools. Testing of the code uses the `pytest` and `Tox` package, and code coverage is tested using `Codecov` .

The main communication channel used in the development of Matplotlib is through their GitHub repository and their mailing list. More open discussion is done via `Google Hangouts` and the Matplotlib `Gitter` channel. The Matplotlib developers also provide support to users through `Stackoverflow` .

4. Functional View

In order for Matplotlib to handle 2D plotting, it is conceptually split into three layers, which can be viewed as a stack [1]. The higher layers depend on the lower layers, while the lower layers are independent of higher layers. The three layers can be seen in Figure 4.

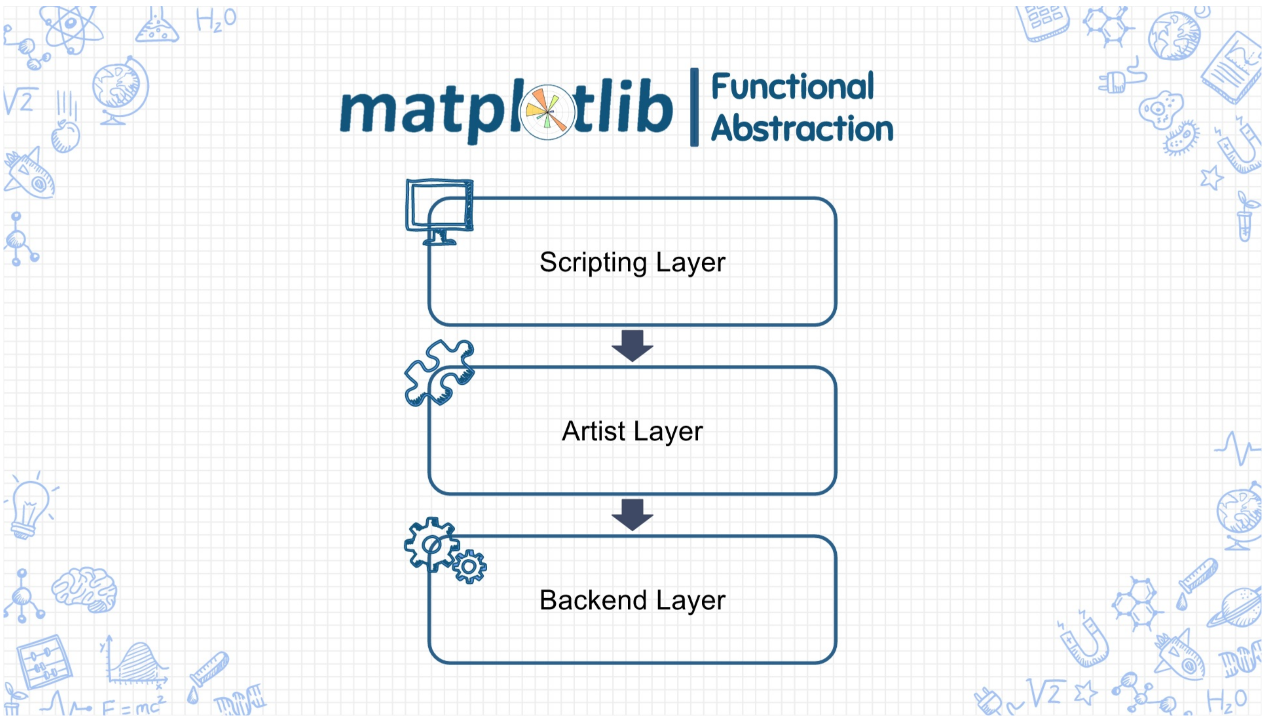


Figure 4. *Functional Abstraction of Matplotlib*

Backend Layer

Matplotlib encapsulates functionalities that interact directly with the environment it is run on into the backend layer. There are three main components in this layer:

- `FigureCanvas` : This component handles the concept of a surface that is drawn into to make the plots, a.k.a "the canvas".
- `Renderer` : This component does the drawing of the plots on the surface, a.k.a "the paintbrush".
- `Event` : This component handles user inputs such as mouse or keyboard events, a.k.a "the viewer".

Through this abstraction, Matplotlib can be extended to work on different platforms including various UI rendering platforms and image formats. Additionally, the backend achieves image consistency between platforms through usage of a C++ 2D graphics library called Anti Grain Geometry.

Artist Layer

From the previous analogy, the artist layer is the object that knows how to use the paintbrush (`Renderer`) to draw on the canvas (`FigureCanvas`). Every image component inside a plot made by Matplotlib (axes, legends, etc) is an instance of the `Artist` class and communicates with the backend through the `draw` function. In fact, most of the heavy-lifting is done in this layer and it comprises most of the code inside Matplotlib.

There are two types of `Artist` objects, `Primitive Artists` which draws basic objects such as `Line2D` and `Circle` , and `Composite Artists` which consists of multiple `Artists` . The most important `Artist` object in Matplotlib is `Axes` which is responsible for composing the 2D data plots by combining multiple other `Artist` objects.

Scripting Layer

With the artist layer, programmers are actually already able to create 2D plots. The scripting layer however encapsulates the lower level image component renditions with a layer that is designed to be easy to use by the average user. This is done to comply with the initial design goal of Matplotlib which is to create a 2D plotting tool that can be used interactively like in Matlab.

The scripting layer is accessed through the `pyplot` module and contains methods to create commonly used plotting graphics such as the histogram or the scatterplot. This layer also handles additional plot arguments such as setting the color of the plot or the plot labels, through the use of Python's `**kwargs` which captures all keyword arguments that are passed together with a method call. `pyplot` then forwards it to the correct `Artist` component to be configured.

5. Development View

We will now take a closer look at the code structure and development process of Matplotlib to derive the architecture used to implement the designed functionality of Matplotlib as well as the strategies employed to standardize its design and development.

5.1. Codeline Organization

Code functionality inside the project can be split into four large sections by their roles as can be seen in Figure 5.

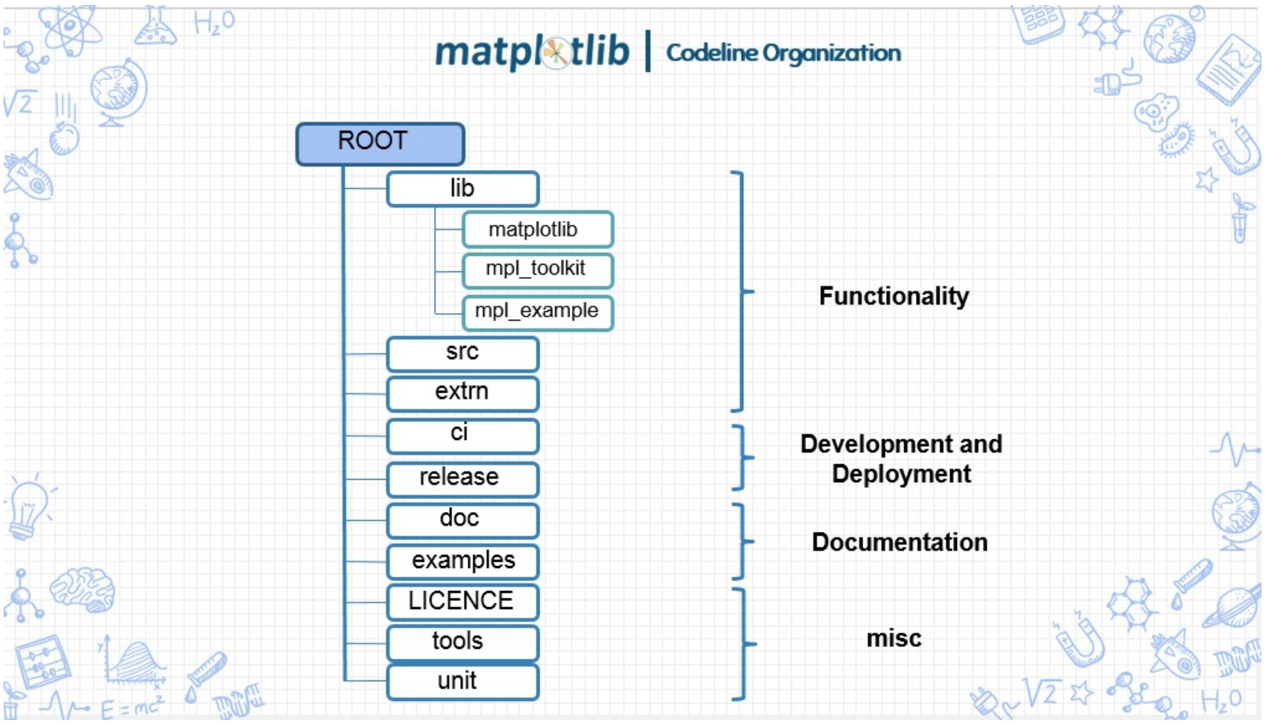


Figure 5. *Codeline Organization of Matplotlib*

The **Functionality** section contains code responsible for the functionalities of Matplotlib and the dependencies required. The main part of the Matplotlib library is contained in the `lib` directory which depends on both code in `extern` (external libraries packaged with Matplotlib) and `src` (c++ code made by the developers) for performance improvement. The `lib` directory is split into three main modules: `matplotlib` contains the core module for implementing 2D plotting, `mpl_toolkits` contains the toolkits module for extending functionalities outside of Matplotlib's scope, and `mpl_examples` contains a symbolic link to the `examples` directory for ease of access in regression testing.

The **Development and Deployment** section contains code used in the development and deployment process of Matplotlib. These are mostly used for developing Matplotlib and do not directly affect the functionality of Matplotlib. There are two directories: `ci` contains files needed by the continuous integration platforms and `release` contains compatibility code for deployment of Matplotlib across platforms. The main part of this category is actually in the scripts at the root directory that are used in the development process such as `setup.py` and `tests.py` .

The **Documentation** section contains code used to generate the documentation of Matplotlib. There are two directories: `doc` contains code to generate the documentation which is hosted at their website and `examples` contains example uses of Matplotlib.

The **Miscellaneous** section contains the rest of the code not contained in the previous three sections. The `LICENSE` directory is for legal purposes, while the `tools` and `unit` directory contains incidental scripts used for specific purposes only.

5.2. Module Organization

The Matplotlib source code is organized into several modules that encapsulates a coherent piece of functionality. Matplotlib modules are partitioned into three main categories as can be seen in Figure 6. These modules are organized in different abstraction layers where the top layer depends on the layers below it.

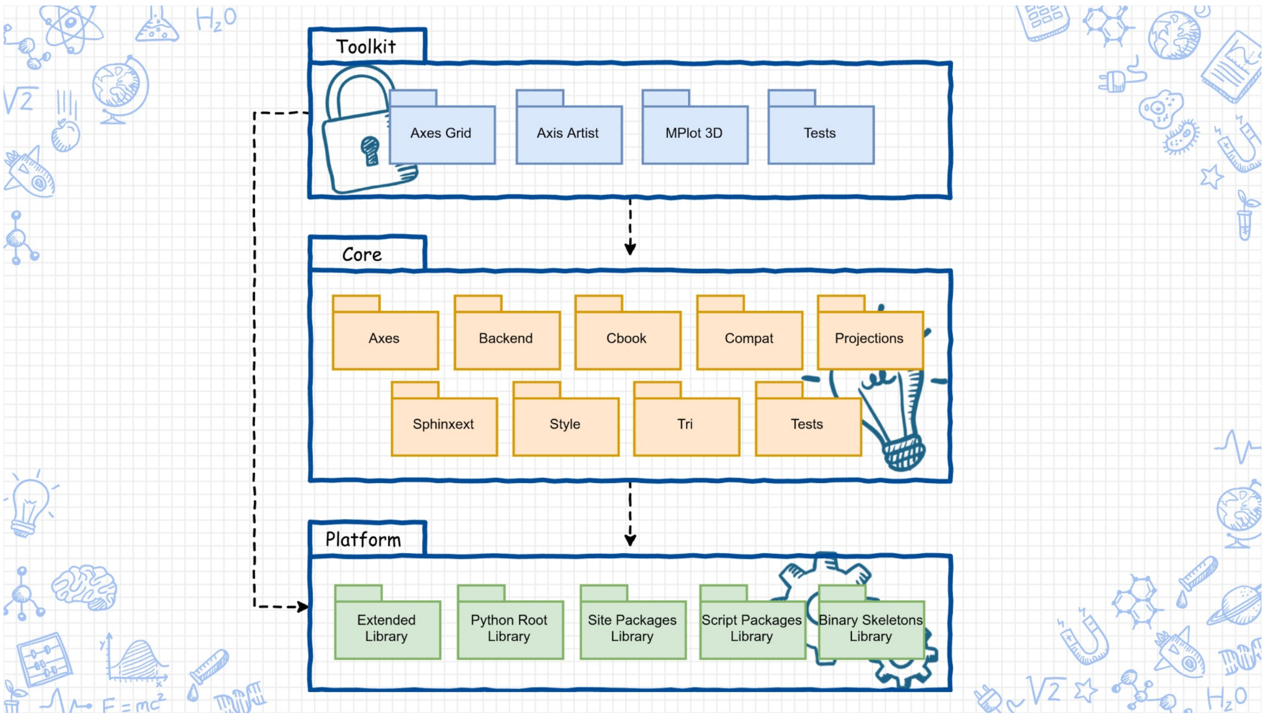


Figure 6. *Module Structure Model of Matplotlib*

The **core module** comprises the core Matplotlib functionality for implementing 2D plotting. The main functional components as described in the Functional View is implemented here. Most of the backend layer is encapsulated inside the `backends` module, while the scripting layer is accessed from the `pyplot` submodule which is autogenerated using `boilerplate.py`. Most of the code here is from the artist layer and is spread in various submodules.

Most of the submodules are in the root directory, but a few submodules have been grouped together inside their own directory which creates a larger module. The most important of these modules are `axes` which is responsible for drawing the plotlines, `backends` which contains handlers for different platforms, and both `tri` and `projection` which handle low level image transformations.

The **toolkit module** enriches the basic functionalities of Matplotlib and depends on the core module. There are currently two major projects in this module: `axes_grid` which adds functionalities to `axes` such as combining multiple `axes` or adding angles to `axes`, and `mplot3d` which adds pseudo-3D plotting to Matplotlib.

The **platform module** consists of supporting modules from external parties such as the basic Python language libraries (Python root library, Python extended library, script package library) and site package libraries such as `scikit-learn` and `numpy`.

The diagram above can be considered as an oversimplification as the connections between modules are much more complex. A closer look at the interdependencies of the submodules is done at the technical debt section.

5.3. Common Design Models

In this section, common designs that are used and standardized in the development of Matplotlib are described.

Common Process Standardization

Since Matplotlib is not a continuously running system, the use of logging is mostly for debugging purposes. Generally, there are two types of logging in Matplotlib, debugging traces which are handled by a class called `Verbose`, and warnings which are handled by the `warning` Python package.

Matplotlib requires intensive numeric calculations to efficiently plot 2D data. Instead of developing from scratch, Matplotlib uses third party libraries. For matrix operations and data handling, Matplotlib uses `numpy`, while for geometric calculations, it uses `antigrain geometry` and `qhull`. Commonly used functions that are shared among many classes such as datetime handling, are stored inside a utility module called `cbook`.

Design Standardization

Matplotlib is a community effort which is developed through GitHub. Development standards are communicated through their developers's guide.

Matplotlib uses PEP8 as a standard style for Python code. It is a set of coding standards created by the Python software foundation. To support Python 2 and 3 from a single code base, Matplotlib uses the six Python library. Meanwhile, design quality is maintained through Pull Request reviews and discussions in GitHub, and larger design goals are compiled into the Matplotlib Enhancement Proposal (MEP).

Testing Standardization

Matplotlib uses standard Python testing libraries for their testing process. The tests mainly consists of unit tests which tests small components of the code, and also a Matplotlib specific "image comparison test". This test generates specific images using the Matplotlib code and then compares the results with baseline images generated previously.

To be able to do the testing internally, developers are required to install `pytest` and `mock`. Ghostscript and Inkscape are also required for the image comparison test. To guarantee changes to the code do not introduce unexpected failures or conflicts, Matplotlib implements continuous integration using Travis CI for unix environments and Appveyor for windows. Both CI platforms are integrated to GitHub and run on every new Pull Request. Matplotlib also uses CodeCov to check the code coverage when there is a change in the test code. To accommodate testing on different versions of Python, Matplotlib uses `tox`.

6. Evolution Perspective

To understand more about the architecture of Matplotlib, we will take a look at its evolution over time. Matplotlib was originally developed as a visualization tool for medical research, but as time goes on, Matplotlib became a popular Python plotting library used by generic users. This interesting evolution of Matplotlib will be analyzed by comparing the state of Matplotlib in each minor and major releases.

Feature Evolution

Matplotlib uses semantic versioning for numbering their releases, which clearly separates major releases, minor releases, and bugfix patches. Figure 7 shows the timeline of Matplotlib version releases based on Matplotlib's version release documentation [3].

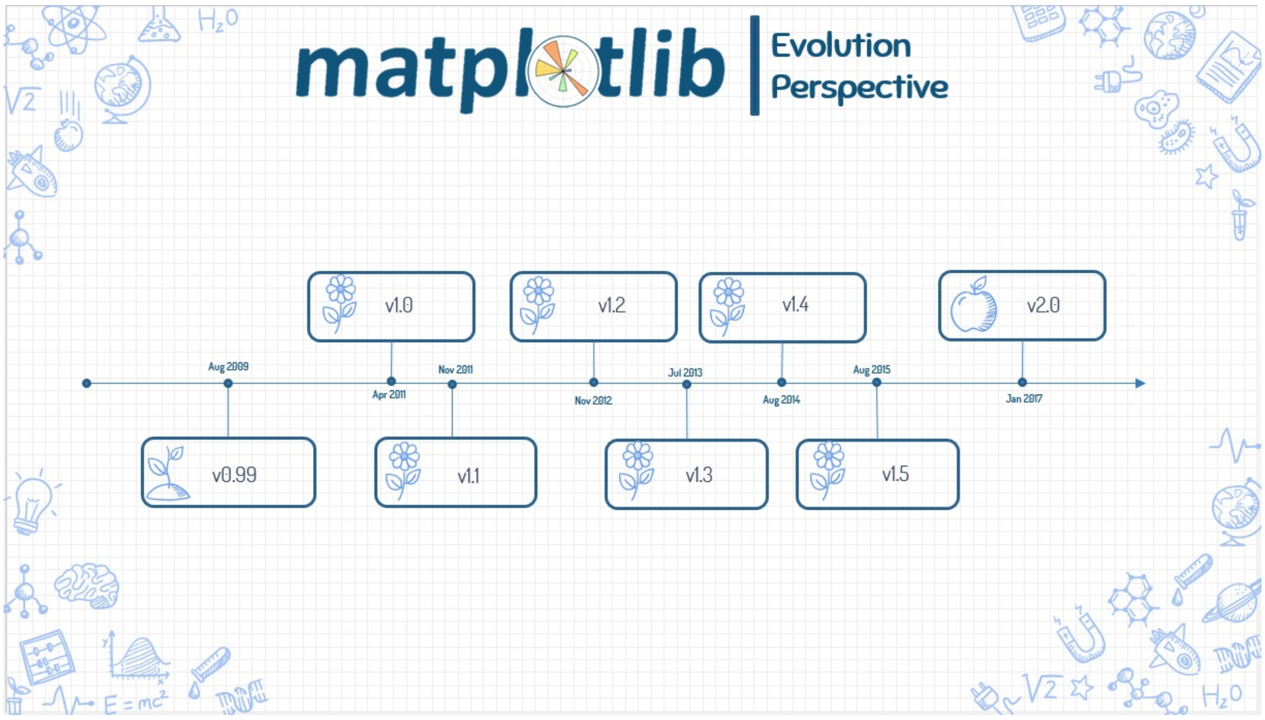


Figure 7. *Matplotlib version history*

- `v0.99` : Added new features (mplot3d, axes grid toolkit and axis spine placement).
- `v1.0` : New backends (HTML5/Canvas), performance enhancements, and new features (complex subplots, triplots, multiple 'show' calls).
- `v1.1` : Introduced animations, new backends (qt4, IPython), new features (sankey diagrams), and improvements to legends and mplot3d.
- `v1.2` : Support for Python 3.x, new backends (pgf/tikz), new features (tri-surface plots, streamplots) and updated shipped dependencies (pytz and dateutil).
- `v1.3` : New backends (webagg), new features (sketch style) and a new setup script.
- `v1.4` : New backends (nbagg), added style package and new plotting features.
- `v1.5` : Interactive OO usage, support for pandas, new colormap and plotting features, and new tool manager.
- `v2.0` : Overhaul of the default styles (font size, colours, mplo3d), fast text rendering, and support for retina displays.

Matplotlib has changed over time to support the latest versions of Python while also dropping support for older versions. For example in `v1.1` Matplotlib supports only Python 2.4 to 2.7, and then with release `v1.2` Matplotlib supports Python 2.6, 2.7 and 3.1. On the current release `v2.0` , Matplotlib only supports Python 2.7 and 3.4+ which are the two major Python versions currently used. Each update also usually brings support for new backends and plotting features. The more recent updates has focused mainly on supporting web backends which shows that Matplotlib is adapting to the trend of web based interfaces.

Another interesting part is how Matplotlib evolves to comply with user's needs. Matplotlib is mainly built for 2D plotting, but along with newer user's needs, some releases (`v0.99` , `v1.0` , `v1.1` , and `v1.3`) make adjustments to Matplotlib to support pseudo-3D plotting. Matplotlib also just recently changed the default plot styles since it was considered not very pretty by the community.

Source Code Evolution

We used cloc to count the lines of code in Matplotlib over time. The graph can be seen in Figure 8.

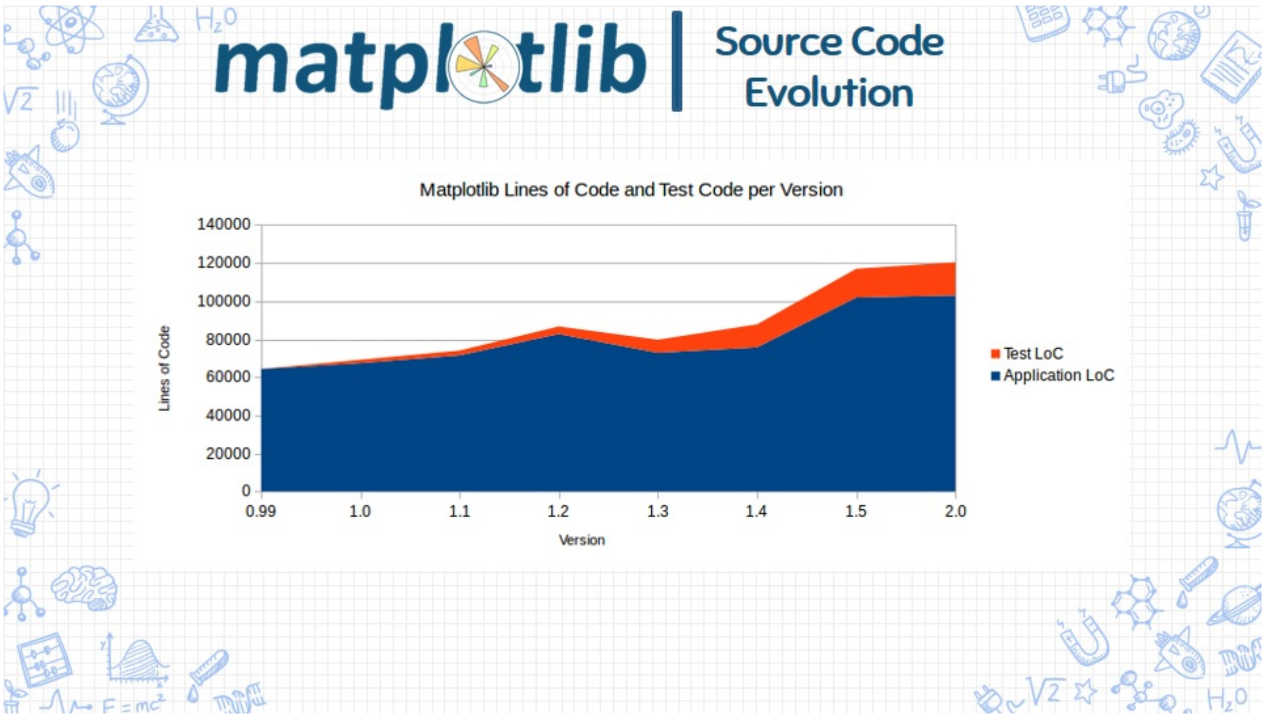


Figure 8. *Matplotlib Lines of Code per Version*

In general, the lines of code increases gradually over time, but there is a slight decrease of code from `v1.2` to `v1.3`. According to the Matplotlib Enhancement Proposal 11 (MEP11), this was because of a refactor. Matplotlib changed `dateutil`, `pytz`, and `pyparsing` into optional dependencies to decouple these third-party libraries from the application code.

It can also be seen that Matplotlib did not have any test code at `v0.99` but then started to add code testing in `v1.0`. Since then, in average the test code increases around 40% per release, while the application code only increased around 5% per release. This trend shows that over time, testing has become one area of focus in the development of Matplotlib.

7. Technical Debt Analysis

After understanding the overall architecture of Matplotlib, we now take a deeper look at the current Matplotlib source code along with recent changes to the code to identify technical debts that exist in the system. To do this, we did both static code analysis and manual code inspection. We also tried analyzing discussions happening in the community, but not much discussions were found that directly addressed technical debts.

From the analysis, there were some findings. One nice finding is that nearly all code in Matplotlib was classified as rank A by Radon with an average of 2.63, which is below the maximum threshold of 10 proposed by McCabe [5]. There were only 7 files that were marked as having a large cyclomatic complexity. We also found that there were no major outdated dependencies and most of the code were already written in idiomatic python. However, there are still a few technical debt that was identified and needs more attention.

Ambiguous Project Structure

When compared to other similar Python libraries such as `scikit-learn` or `bokeh`, Matplotlib's project structure can be considered messy. This is mainly because some files and directories are not self explanatory. For example, the `unit` directory in Matplotlib implies that it contains unit tests, but it actually contains problem specific testing scripts. The separation of the Matplotlib source code is also not clear between the `src`, `lib`, and `extern` directory.

These ambiguous code affect the developers of the system, potentially making development more tedious. There are currently also too much scripts in the root directory which would make it confusing for new contributors, but this is currently already being handled by the developers (#8276).

A simple step in improving this would be to put the contents of the `unit` directory as well as most scripts in the root directory into the `tools` directory. All functional code (`lib`, `src`, and `extern`) and build tools (`ci`, `release`, and some scripts in the root directory) can also be encapsulated into their own directory and renamed accordingly to better reflect their contents.

Messy Module Dependencies

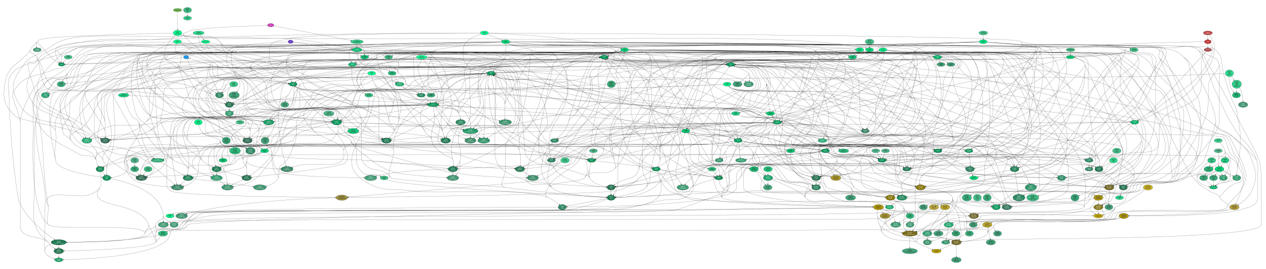


Figure 9. *Dependency Graph of Matplotlib*

The dependencies between internal modules of Matplotlib generated by pydeps is shown in Figure 9. Modules from core is shown in green, modules from toolkit in yellow, and the other colors indicate external dependencies. It can be seen that the dependencies inside Matplotlib are not so structured and that modules are not well separated. The separation between toolkits and core is also not so clean.

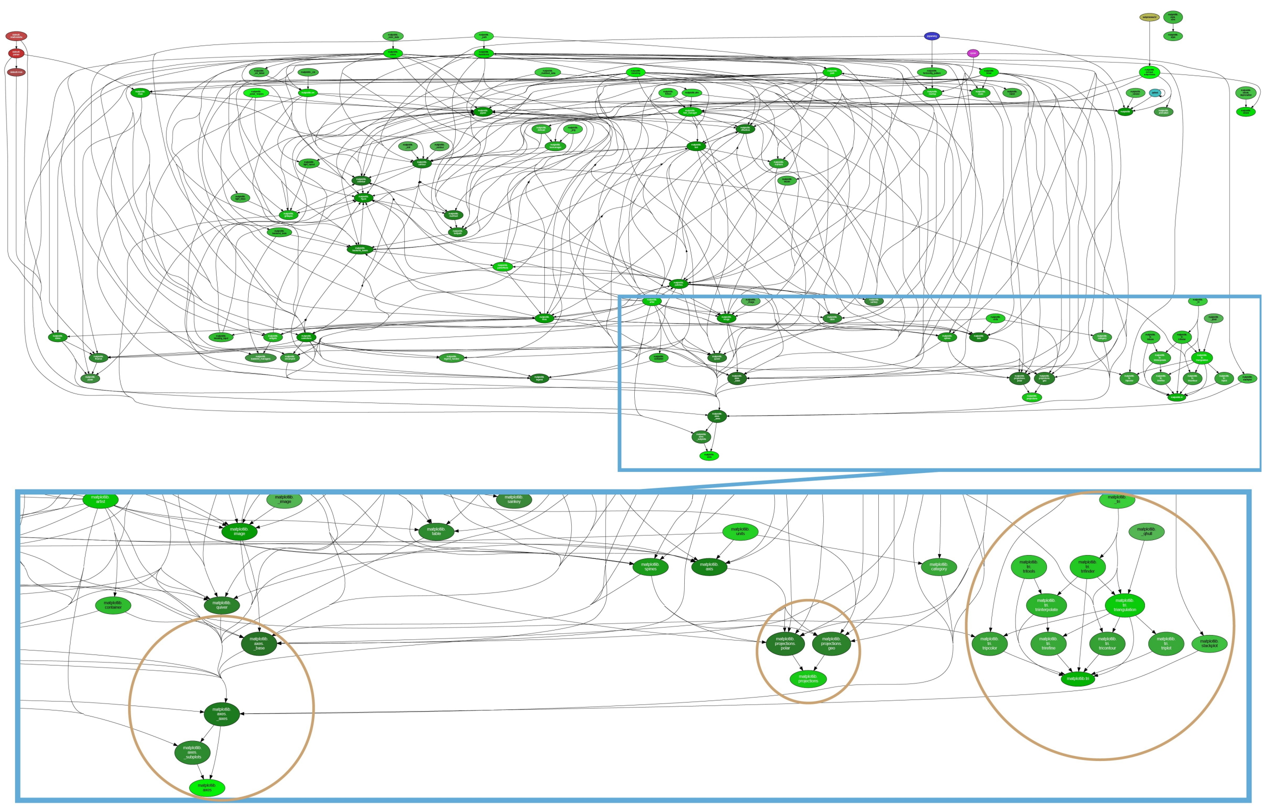


Figure 10. *Trimmed Dependency Graph of Matplotlib Core*

A trimmed dependency graph of the core module without `backends` and `sphinxext` is shown in Figure 10. A closer look at the bottom right side shows that there is a well separated module group which is `tri` . To the left of that, there are also two smaller groups which are `projections` and `axes` . This shows that there are some attempts at encapsulating large concerns. However, overall the module organization in Matplotlib needs more separation of concerns. Fixing this issue would need a large refactoring of the whole Matplotlib project.

Code Inconsistencies

Matplotlib has just recently migrated from using the `nose` testing framework into `pytest` . Despite this, some traces of the previous framework still exist in the system. The main trace we found is that the `testing` directory in each Matplotlib module which was previously created to extend the functionalities of `nose` , still exists until now. The Matplotlib website also still lists `nose` as their testing framework, but this is apparently already corrected in the code of the documentation and the website is not updated yet.

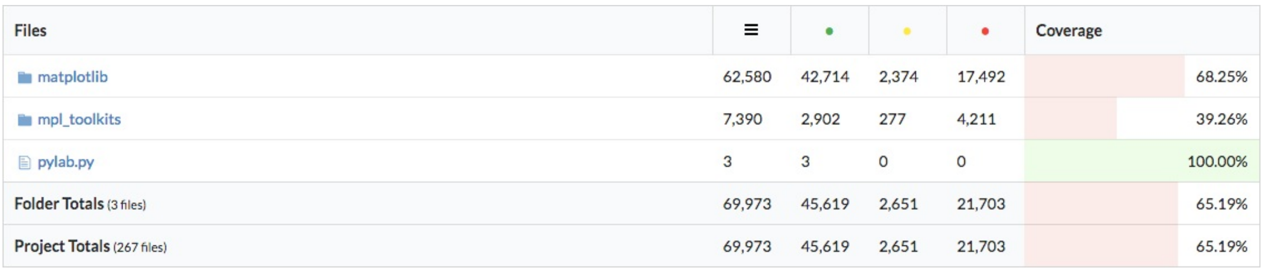
For logging purposes, Matplotlib has implemented their own utility class called `Verbose` , but many classes in Matplotlib still use a local debugging flag to control their debug loggings, and some even has local functions to handle this functionality. An example of this is in the `backend_qt5agg.py` file which uses a boolean flag named `DEBUG` .

There were also still 54 `TODO` and 13 `FIXME` that exist in the source code of Matplotlib at the time of our analysis, which shows there were some issues that are still not addressed yet.

This shows that the code inside Matplotlib is not very strongly standardized and maintained, which in the long run would make it harder for developers to extend the code since different classes have different code smell and conventions.

Low Code Coverage

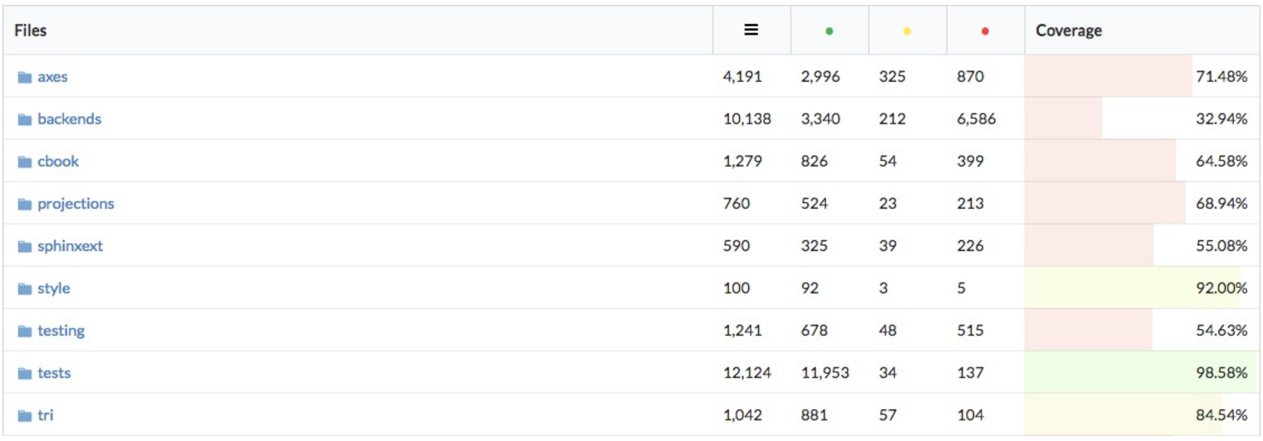
Code coverage is a measure used to indicate how much code has been covered by a test. Low coverage implies that the program has a high chance of containing undetected *bugs*. High coverage does not necessarily signify all actions will be correctly processed by the code, but at least it indicates that the likelihood of correct processing is good [4].



Files	≡	🟢	🟡	🔴	Coverage
matplotlib	62,580	42,714	2,374	17,492	68.25%
mpl_toolkits	7,390	2,902	277	4,211	39.26%
pylab.py	3	3	0	0	100.00%
Folder Totals (3 files)	69,973	45,619	2,651	21,703	65.19%
Project Totals (267 files)	69,973	45,619	2,651	21,703	65.19%

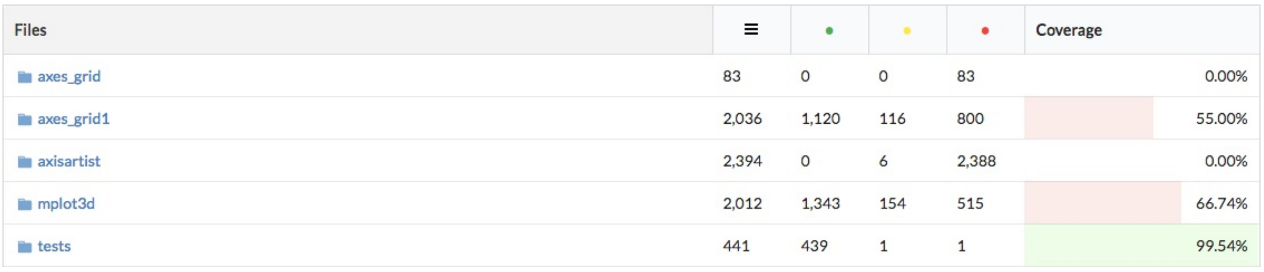
Figure 11. *Code Coverage of Matplotlib*

Matplotlib uses `Codecov` to measure code coverage automatically. At the time of analysis, Matplotlib obtains 65.19% code coverage (Figure 11). This is still considered as a low code coverage as it is below the high threshold of 80% [4]. The full results of our investigation for each module can be seen in Figure 12 for the Core modules and Figure 13 for the Toolkit modules.



Files	≡	🟢	🟡	🔴	Coverage
axes	4,191	2,996	325	870	71.48%
backends	10,138	3,340	212	6,586	32.94%
cbook	1,279	826	54	399	64.58%
projections	760	524	23	213	68.94%
sphinxext	590	325	39	226	55.08%
style	100	92	3	5	92.00%
testing	1,241	678	48	515	54.63%
tests	12,124	11,953	34	137	98.58%
tri	1,042	881	57	104	84.54%

Figure 12. *Code Coverage of Core Modules*



Files	≡	🟢	🟡	🔴	Coverage
axes_grid	83	0	0	83	0.00%
axes_grid1	2,036	1,120	116	800	55.00%
axisartist	2,394	0	6	2,388	0.00%
mplot3d	2,012	1,343	154	515	66.74%
tests	441	439	1	1	99.54%

Figure 13. *Code Coverage of Toolkit Modules*

A potential part of the code for improvement is the `backends` directory which currently only has 32% coverage albeit being a core part of Matplotlib. The `backends` is responsible for low level adjustments of Matplotlib to various graphic renderers, therefore improving code coverage in this part of Matplotlib would make future changes to Matplotlib more resistant to platform errors.

Long Testing Time

Another source of technical debt is how long it takes to run tests. The runtime of integration tests are not strictly limited, but for unit tests which are supposed to be run repeatedly, according to Martin Fowler, most experts agree that it should be fast and typically run less than a minute. A long unit test runtime means that the rate at which people can develop will be slowed down. The tests in Matplotlib when run in our own laptop achieved a runtime of around 7 minutes, while the average runtime of tests for continuous integration (in Jenkins) is around 20 minutes. This makes it hard to do rapid development of the Matplotlib code since it takes too long and the test script that is provided doesn't separate between unit and integration tests.

A possible solution is to separate out the integration tests, such as the Matplotlib image comparison test, into their own test script. This will reduce the number of tests that needs to be run frequently. If this is not enough, Martin Fowler also suggested that unit tests should be separated into a "compile suite" that is run on every compilation and a "commit suite" which is run before a commit (or usually on a pull request).

8. Conclusion

Matplotlib is one of the oldest yet reliable and well-known visualization library in Python. During our investigation of Matplotlib, we identified some interesting findings that we believe will be useful to understand the current state of Matplotlib.

John Hunter initially designed Matplotlib as a Python tool strictly for 2D plotting and targeted it towards people of an academic background. Now Matplotlib has evolved into a thriving community project while still staying true with the initial design goals. As an open source project, most of the stakeholders are developers, with the NumFOCUS organization providing financial support through donations. From our short experience contributing to the project, we have identified Thomas A. Caswell who is also one of the lead developer, as the person to contact regarding the development of Matplotlib.

From the analysis, we learned that the functionality of Matplotlib is conceptually split into three layers, which at a glance is simple. However, digging deeper into its code, the architecture of Matplotlib is actually quite complex, and at its current state there are still quite a few technical debts. For instance, the interdependencies of the modules are too unstructured which makes the separation of concerns unclear. This makes it hard for most developers to contribute or change specific components of the system. There are also quite a few inconsistencies and lack of standardization in the code of Matplotlib, and some ambiguity in the project structure.

We have found that along its development, Matplotlib has improved a lot in terms of code tests, but currently the code coverage of Matplotlib is still too low. More tests need to be added especially to the `backends` module. The test script of Matplotlib is also not separated well between different types of tests, resulting in a very long test runtime which inhibits the speed of development.

To conclude, Matplotlib has evolved a lot from the research tool created by John Hunter, to the vastly used data plotting library it is today. Albeit the technical debts that still exist until now, Matplotlib has shown promising growth over the years. We strongly suggest people to contribute to Matplotlib as it has an open community that warmly welcomes new contributors as we have experienced in proposing some contributions.

References

1. Amy Brown and Greg Wilson (editors). The Architecture of Open Source Applications.Volume 2. 2012
2. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition
3. Matplotlib Project. (n.d.). Retrieved February 26, 2017, from http://matplotlib.org/

4. Brader, Larry; Hilliker, Howie; Wills, Alan (March 2, 2013). "Chapter 2 Unit Testing: Testing the Inside". Testing for Continuous Delivery with Visual Studio 2012. Microsoft. p. 30

5. McCabe, Thomas J. "A complexity measure." IEEE Transactions on software Engineering 4 (1976): 308-320.

Mockito - the most popular mocking framework

By @Xin Liu, @Lu Liu, @Saiyi Wang, and @Xiang Teng
Delft University of Technology

Abstract

Mockito is the most popular mocking framework for unit test in Java [2]. It is famous for its compact and effective APIs and widely used by Java developers all over the world. Mockito is an open source project living in GitHub and it is being maintained and developed by the core developers as well as quite a few external contributors. In this chapter, we study the software architecture of Mockito in the following aspects: stakeholder analysis, context view, development view, functional view, evolution perspective, future release perspective and technical debts. The study goes from a high-level analysis to more technical views and perspectives. In this process, we find Mockito is a very well organized project with high maintainability, readability and extensibility.

Table of Contents

Introduction

In software development, unit tests are necessary to indicate whether small sections fit for work. In Java, these sections are usually a class or just a method. Mocking is a primary approach in unit test. In a unit test, developers only want to verify the correctness of a specific object. However, it usually has dependencies on other objects, which may be impractical to be called in a unit test. Mocking the dependencies and simulating their behaviors solve the problem.
The first release of Mockito (version 0.9) was in 2008. Mockito has released many versions since its continuous release model implemented. Till March 2017, the version of Mockito is 2.x, and the Mockito team is planning for releasing Mockito 3. Mockito is voted as the best mocking framework for java by massive StackOverflow community, which is due to its feature of letting users write beautiful tests with a clean and simple API [2].

Four TU Delft students from DESOSA (Delft Students on Software Architecture) [3] group have made a deep analysis on architecture of Mockito based on the theory of software architecture referring to the book *Software Architecture* of Ronzanski & Woods [1]. There are several viewpoints and perspectives mentioned in this book. The chapter starts with the analysis of stakeholders who are involved in the development of this project, followed by context view about the relationship and interactions between Mockito and its environment. After that development view shows the modular structure of the system, while functional view shows the functionalities of Mockito and how modules implement all the required features. The evolution perspective talks about the ability of the system to be flexible in the face of the inevitable change that all systems experience after deployment. And future release perspective shows the improvement of method and model in the future versions. At last, technical debt analyzes how well Mockito is implemented.

Stakeholder Analysis

In traditional software development process, the parties related to this are specified at the beginning. Different users are interested in different aspects of the software architecture. These users are defined as stakeholders. Some important stakeholders of Mockito are tabulated in Table 1. These stakeholders are categorized in 8 groups according to chapter 9 of the book Software architecture of Rozanki & Woods(2012) [1]. In addition, an analysis of their associated levels of power and interest is shown in Figure 1.

Identification

Mockito is an open-source testing framework started by @Szczepan Faber in early 2008. He is the main developer in the early stage. Currently Mockito is maintained by @Szczepan Faber, @Brice Dutheil, @Rafael Winterhalter, @Tim van der Lippe and other developers. Travis CI and Bintrary are used to facilitate continuous delivery.

Table 1. Stakeholders of Mockito

Stakeholder	Description
Developers & Testers	In the early stage, the main developers are @Szczepan Faber and his friends. Currently Mockito is maintained by the core development team consisting of @Szczepan Faber, @Brice Dutheil, @Rafael Winterhalter, @Tim van der Lippe and other 89 external contributors (as of 21th of February 2017) on Mockito.
Maintainers	As this is an open source and non-commercial software, therefore, most of the maintainers are the contributors of the GitHub and the core development team. For instance, @Christian Schwarz and @Pascal Schumacher are two main external contributors.
Assessors	According to the closed issue in Mockito, it could be asserted that the main assesors come from the core development team which consist @Szczepan Faber, @Tim van der Lippe, @Brice Dutheil and a few others.
Communicators	There are several ways that have been used to explain the system to the other stakeholders such as Dzone Reference Card, Javadoc(Mockito 2.7.10 API), Twitter and the Blog. For the blog, it is governed directly by @Szczepan Faber. Before 2017, the most information is written on LinkedIn. The documentation on DZone is written by @Marcin Zajączkowski.
Suppliers	Dexmaker is used by Mockito to support Android users. PowerMock is used to test code that is normally regarded as untestable by Mockito. Another tool that is used to manage project is called Maven. However, the actual building tool is Gradle. Moreover, IntelliJ IDEA is used by developers.
Users	Two main utilize places of Mockito are in creating a unit test and to simplify and enhance the integration tests respectively. Here are some projects that use Mockito in late 2010. Such as Sonar, EHcache, Dozer and so on. Some companies or commercial products also use it such as Atlassian or Grid Dynamics. The recent versions also support Android via Dexmaker.

Additional types of stakeholders

- Competitor: EasyMock, in fact, Mockito is a framework which is grown out from it.
- Cooperator: JUnit is an important collaborating framework of Mockito. It is a simple framework to write repeatable tests. TestNG is a testing framework inspired from JUnit and NUnit but introduced some new functionalities that make it more powerful and easier to use.
- Dependency: Spring, ByteBuddy, CGLib and AssertJ.

Power interest matrix

The power and interest on Mockito of each group is obvious by viewing the position in the Figure 1. The most important party of the stakeholders is the core developer team. It has high power as it makes the majority decisions in Mockito. The reason why they also have high interest is that they are the main contributors. The second group needing mention is the third-party dependencies. Mockito is an open source project which has a high-level dependency on third-party libraries. The details could be found in Table 1. So, they have high power in Mockito, however, they are not so interested in Mockito. The last group needed to be shown is users. It is the group having low power (only a few of them are contributors) and high interest (their products directly rely on the functionalities of Mockito).

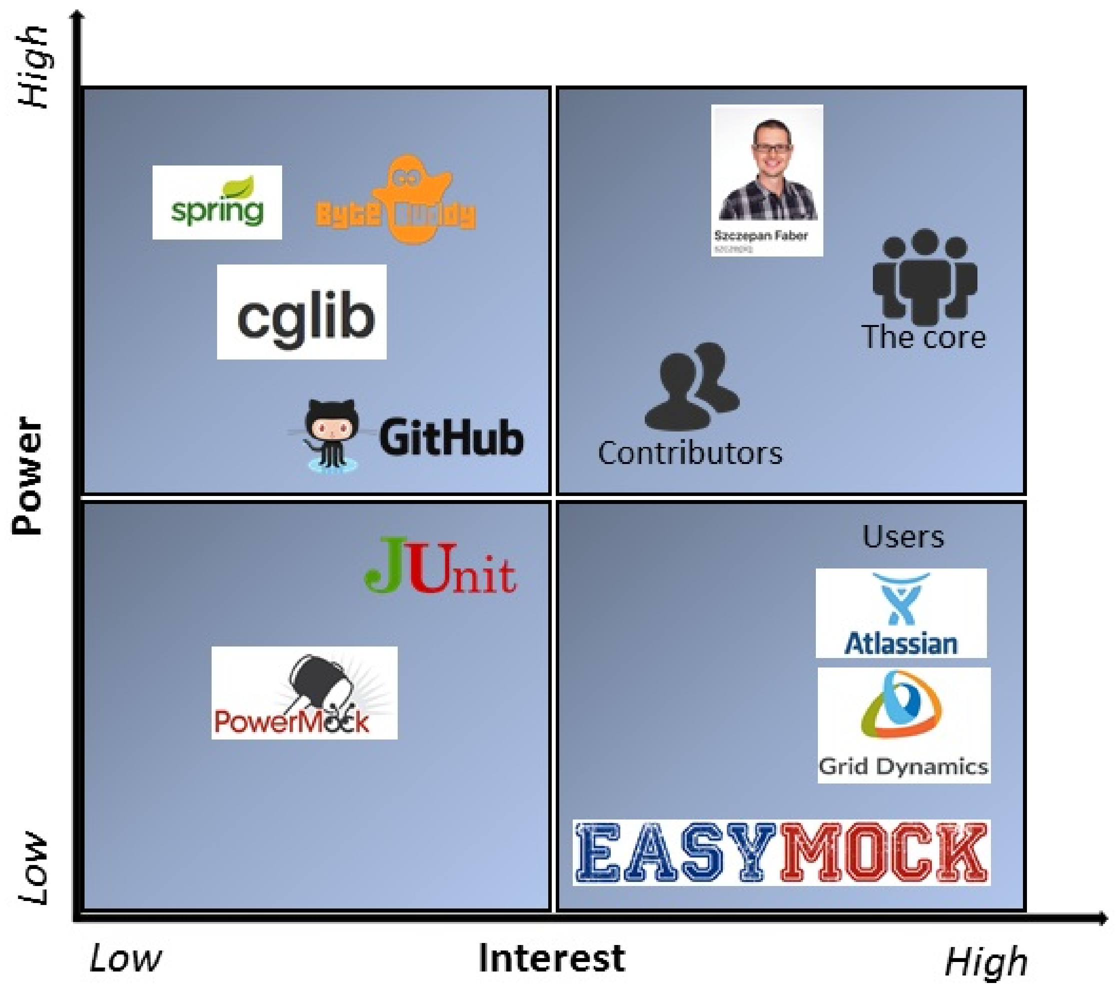


Figure 1. Power interest matrix

Integrators

In this section, the integrators who deal with the contributions and decide whether to merge them or not are identified. Also, how a decision is made about pull requests is discussed in details.

Identification

According to the activities regarding reviewing and merging recent pull requests, we identify @TimvdLippe as the main integrator, while @bric3 and @szczepiq also review the pull requests frequently.
@TimvdLippe takes part in reviewing most of the pull requests in recent time due to some reasons. He makes decision himself on small changes like minor optimizations and refactoring. When the pull requests have obvious influence on the code he would invite other core members to review together. He also friendly helps the external contributors to rebase their PRs to meet their format and style requirements. On the other hand, @bric3 and @szczepiq also show up frequently to help to review the pull requests and give suggestions.

Challenges

The integrators often hold different opinions towards a single pull request. For instance, in #932, @TimvdLippe and @szczepiq were in favor of the change while @bric3 thought the issue was more like a GitHub bug and he preferred not to fix it themselves. Another example is #942 where @TimvdLippe hoped to see more tests specifically for Java8 before merging it, while @szczepiq thought it was more desirable to forward the pull request even without the tests. We can then tell that integrators indeed have an issue with reaching consensus.

Merge Decision Strategy

When the integrators want to merge a pull request, they mainly care about the source code quality and code style of the incoming pull request. For example, pull request #955 is about modifying the code related to the release warning displaying, which has a satisfied code quality and therefore has been merged. Moreover, the integrators also consider the commit set and whether it adheres to the project conventions for submitting pull requests. Besides, the integrators will judge whether the pull request fits the project roadmap and the technical design of the project. Interestingly, we also found that there is no difference in treatment of pull requests from the core team or from the project's community.

Context View

With the service supplied by Mockito, users can write unit tests for Java projects. Users can search for how to use Mockito through Javadoc, release notes or Dzone. There are some related projects that are for the unit tests for other programming languages such as Python , Flex, Javascript, Scala, Objective C, Perl, PHP, MATLAB, TypeScript, and Dart. Also there are some extensions for Mockito such as Junit 5 mockito extension and Spock subjects-collaborators extension.

Figure 2 presents the context of Mockito including relationships, dependencies, and interactions between the framework and its environment. This environment includes the people, systems and external entities.

External entities

There are several entities in Mockito, and together they set up the environment of it. The external entities are listed below:

- Developing language: Java
- IDE for programming: IntelliJ IDEA
- Communication tools: Stack overflow, Mailing-List, GitHub, twitter
- Continuous delivery tools: Travis CI, Bintray
- Supported by: Maven(lastly), Gradle(primarily), PowerMock
- License support: MIT license integration tests that use Mockito as end-user
- Documentation: Javadoc.io, DZone
- Users: Sonar, Ehcache, Dozer, Apache Wicket, Spring Integration, Apache Felix, HBase, etc.
- Competitors: EasyMock
- Collaborating testing framework: JUnit
- Dependency: Bytebuddy, Spring, CGLib, AssertJ

Context view diagram

The diagram below shows the context view of Mockito. It contains the external entities and the most important stakeholders mentioned in Stakeholder Analysis section.

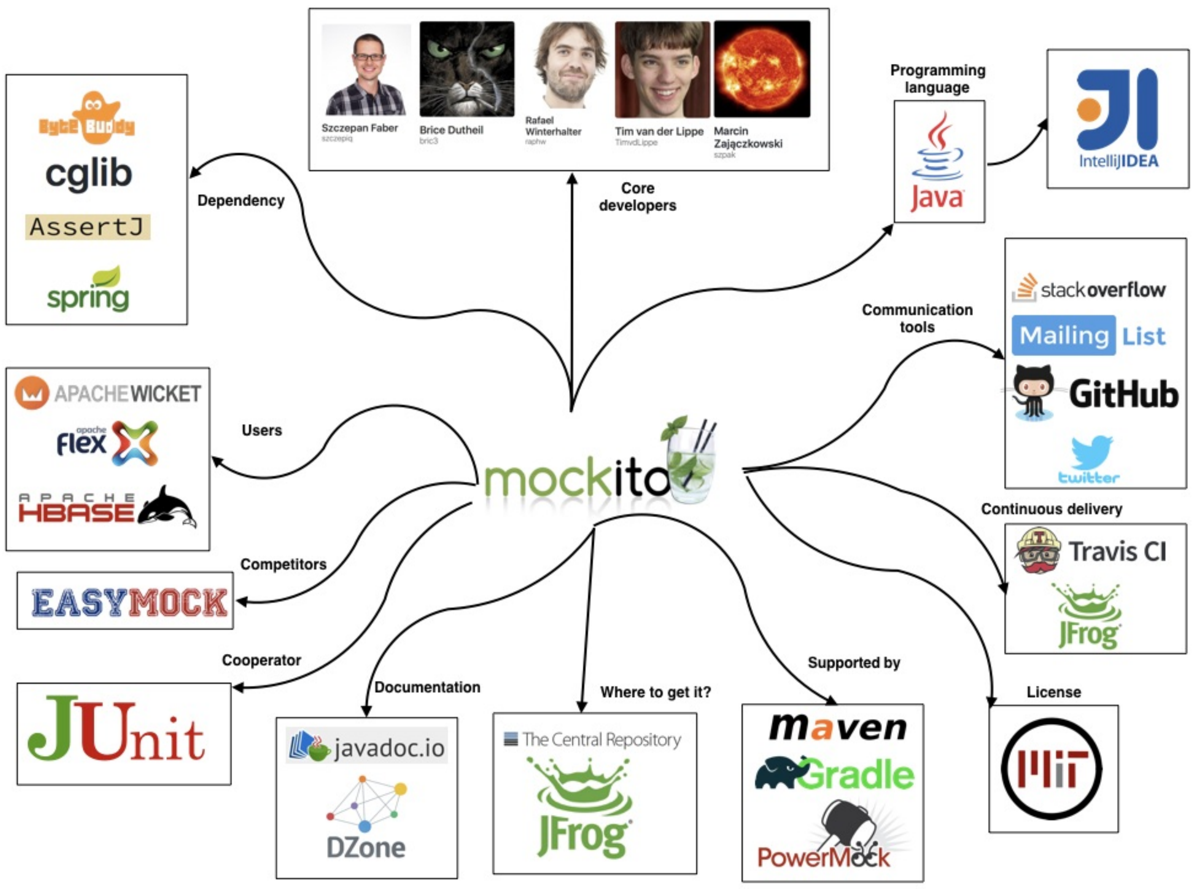


Figure 2. Context view diagram

From the context view diagram, both entities and important stakeholders are revealed. The core developers in the diagram consist the development team. They created Mockito, contribute to and maintain the whole project mostly. In the development process, the communication tools are used as communication platform of developers and platform to convey the information of the system to other stakeholders. The programming language is mainly Java and the IDE used is IntelliJ IDEA. When developing, the dependencies mentioned above are the base of the project, and used as library or framework of the system. Travis and Bintray serve for continuous delivery.

Apart from development, the access of Mockito is another crucial part in context view. To know detailed information of Mockito such as the examples of testing code, users are supposed to read Javadoc and DZone. To get the Mockito package, users can check the Maven Central Repository or Bintray. To use Mockito, Gradle is primarily required, but projects built with Maven are also supported.

The source code of Mockito and the associated documentation files are licensed under the MIT license.

Development View

As mentioned in the book written by Rozanki & Woods(2012)[1],

> The development view of a system describes the architecture that supports the software development process.

This section is about development of Mockito. Development view supports the design and build of software for complex systems. It includes module organization, common processing, and codeline organization.

Module Organization

Mockito consists of a large number of source files, which are organized into different modules logically. All the modules are described in the table below.

Table 2. Module organization

Module	Description
Documentation	Documentation of Mockito project including JavaScript and CSS code for the Javadoc website
Test	All the test-related files
Configuration	Mockito configuration utilities and configuration
Exceptions	Classes for exceptions and errors, stack trace filtering/removing logic, cleaning public APIs
Hamcrest	Mockito Hamcrest matcher integration
Invocation	Public API related to mock method invocations, invocation machinery and related classes, and implementations of real method calls
Junit	Mockito JUnit integration, rule and runners, and JUnit integration support classes
Listeners	Public classes related to the listener APIs
Runners	JUnit runners, internal classes and utils for runners implementations
Session	Mockito session builder and implementation
Stubbing	Answers for stubbed calls, implementations of default answers, stubbing logic and implementations
Verification	Verification checkers, implementations for dealing with matching arguments, verification logic and implementations
Creation	Classes for mock object creation including its setting, instance, ByteBuddy related stuff, and other
Debugging	Everything that helps debugging failed tests
Framework	Default Mockito framework and session
Handler	Classes calling all listeners wanted for the mock, before delegating it to the parameterized handler
Matcher	Argument matchers for verification and stubbing
Progress	Mocking progress stateful classes
Reporting	Classes for dealing with nicely printing verification errors
Util	All the static utils including reflection utilities, IO utils, etc.

All modules interact together to implement the function of Mockito. To show the organization of Mockito's source code, the module structure model is figured out. In Figure 3, we can see the modules into which the individual source files are collected and the dependency among these modules. Only the main part of the system is shown. Besides, there are a test part, some subprojects and other files related to the project.

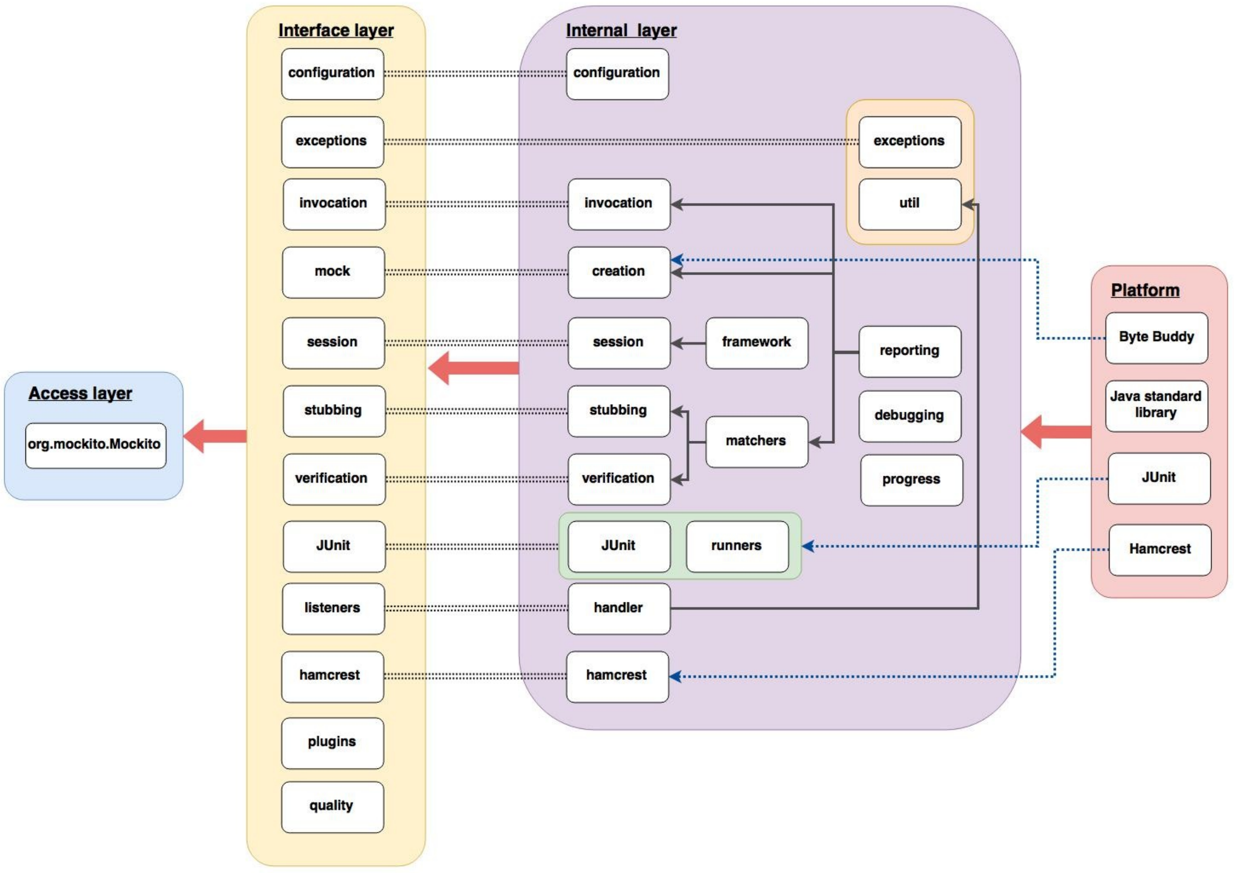


Figure 3. Module structure model

- **Access layer** - Provides external support of Mockito. Through Gradle or Maven, users can access Mockito as a library for unit tests.
- **Interface layer** - Interfaces for the internal layer, including configuration interface, exceptions interface, invocation interface, listeners, session interface, etc.
- **Internal layer** - All the core classes in Mockito are functioned, including configuration, exceptions, matchers, invocation, stubbing, session, etc.
- **Platform** - All the external and basic libraries for internal layer, including Java standard library, Hamcrest, JUnit, and ByteBuddy.

Common processing

Identifying and isolating the common processing into separate code units could help to reduce the level of duplication of the code and make the code easier to be understood by the users.
In this section, some of these common processes are discussed.

Logging message

A class named `SimpleMockitoLogger` is used to create logging messages. It is written by the team of Mockito. The first type of logging message is created while stubbing arguments are mismatched. The second type is generated while a stubbing is unused. And the third type is used to log method invocation.

Use of third-party libraries

The development of project Mockito is based on the third-party libraries. Here, three main libraries are discussed. The first one is Junit, by which many functions are achieved. For instance, Junit is used to detect unused stubs. ByteBuddy is to enable the developers to modify Java classes during the runtime without a compiler. As for Hamcrest, its existing matcher class is used to achieve the functions of Mockito. There are some other dependencies such as Spring, CGlib or AssertJ used. Although third-party dependencies are inevitable, the core developers want to reduce dependencies to lower the risk of version conflicts.

Codeline organization

The overall structure of codeline is defined as how the code is controlled, where different types of source code live in that structure, and how it should be maintained and extended over time.

There are five important folders at the root of the Mockito repository, which are **config/checkstyle**, **doc**, **gradle**, **src**, and **subprojects**.

Table 4. Codeline organization

File	Inside folders or files	Description
config/checkstyle	checkstyle.xml, checkstyle.xsl	
doc	design-docs, licenses, release-notes	All the documents of Mockito
gradle	mockito-core, root, wrapper	Everything related to Gradle
src	conf, javadoc, main/java/org/mockito, test/java/org	The core source code for the system, testing code for the system, code to implement Javadoc, and configuration
subprojects	android, extTest, inline, testing	Some subprojects based on main part

To show more details in the codeline organization, the codeline organization model is visualized in Figure 4.

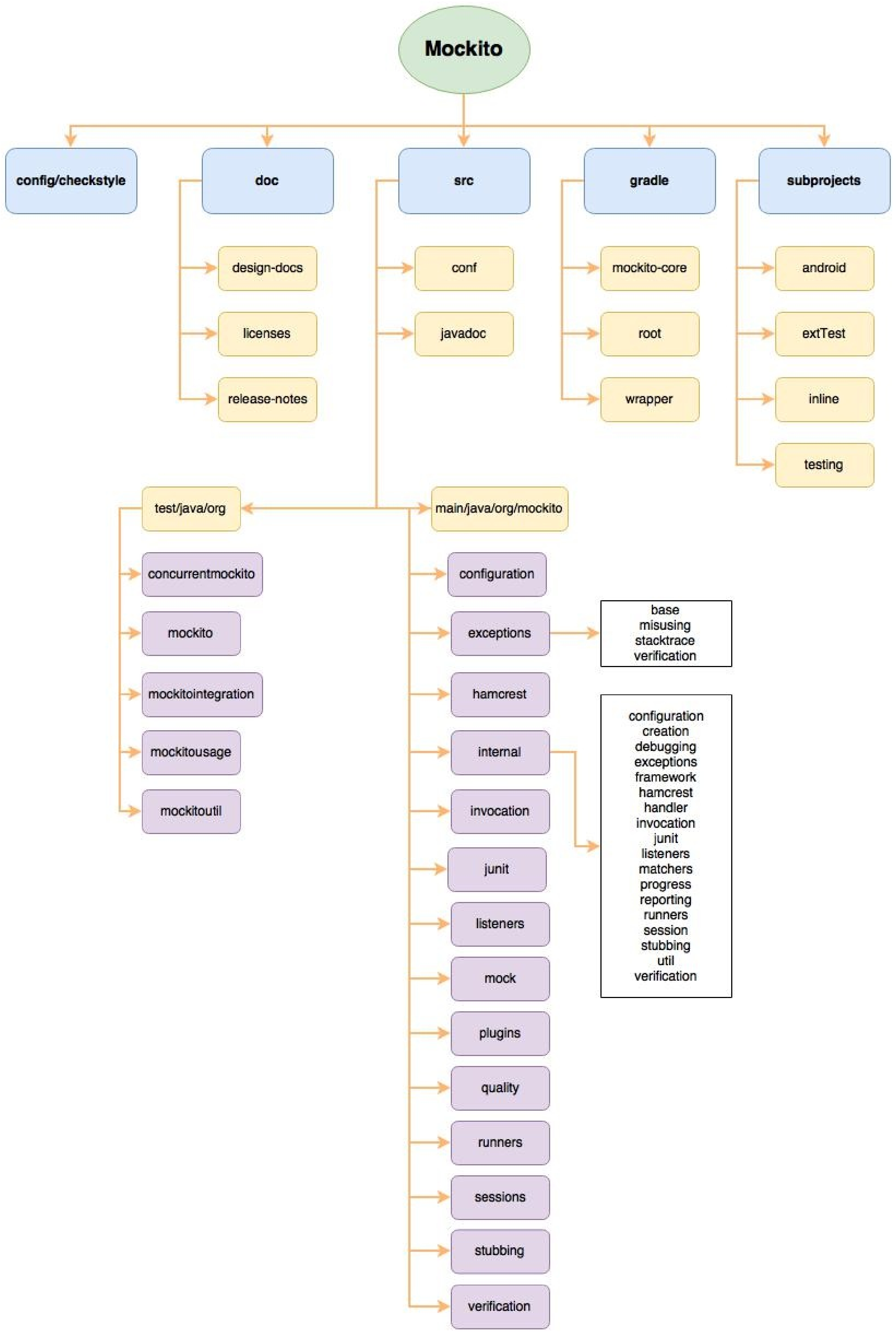


Figure 4. Codeline organization model

Release process

Since its first release in 2008, Mockito has released more than 266 versions till April 3rd, 2017, which is much more than many peer projects. It is due to the continuous delivery model on release management [4].

Releasing as soon as a change is made reduces management overhead and it shortens the maintaining duration. It also forces the team to work more efficiently and regularly. What's more, this mechanism motivates the external contributors as their contribution can be published once adopted. Lastly, it is a great opportunity for the contributors to develop techniques, tools and accumulate knowledge.

Mockito welcomes discussion on continuous delivery model and there is an open issue intending on it.

Functional View

According to Rozanski and Woods`s book [1],

> The functional view of a system defines the architectural elements that deliver the system`s functionality.
>
> The view documents the system's functional structure-including the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them.

In this part, the functional capabilities, external interfaces, internal structure and design philosophy are concerned.

Functionalities

Functional capabilities

Functional capabilities are what the system is required to do. The core functional capability of Mockito is to let users write a good unit test with a clean and simple API. Comparing to expect-run-verify library, EasyMock, it offers simpler and more intuitive approach. Users can ask questions about interactions after execution. There are several functionalities of Mockito, which are tabulated in Table 5.

Table 5. Functionalities of Mockito

Functionality	Description
Main: Provide library for unit test	Letting users declare Mockito dependency with Gradle and then users can use all the subpackage inside org.mockito to write unit test
No expect-run-verify	Mocks are often ready without expensive setup upfront, so users do not have to look after irrelevant interactions
Sub: Mock creation	The API is very slim, and there is only one kind of mock and only one way to create mocks. It can be serialized/ deserialized across classloaders
Sub: Mock Stubbing	Stubbing mock objects, void methods with exceptions, consecutive calls, etc.
Sub: Mock verification	BBD style verifications of exact number of invocations, ingnoring stubs, etc. in order with timeout
Sub: Spying testing process	Spying on real objects and abstract classes

Apart from these functionalities, Mockito have more details features which can be found in `Features and Motivatoins` in Wiki and there are updating features referring to different versions of Mockito mentioned in Java.doc.

External interfaces

The main package of Mockito is org.mockito, which is the external interface providing for users. The core subpackages of org.mockito are listed below [5].

Table 6. Core subpackage of org.mockito

Package	Description
org.mockito.configuration	Mockito configuration utilities.
org.mockito.exceptions.base	Base classes for exceptions and errors, stack trace filtering/removing logic.
org.mockito.exceptions.misusing	Exceptions throen when Mockito is misused.
org.mockito.exceptions.stacktrace	Stack trace filtering/cleaning public APIs.
org.mockito.exceptions.verification	Verification errors
org.mockito.exceptions.verification.junit	JUnit integration to provide better support for JUnit runners in IDEs.
org.mockito.hamcrest	Mockito Hamcrest matcher integration
org.mockito.invocation	Public API related to mock method invocations
org.mockito.junit	Mockito JUnit integration; rule and runners
org.mockito.listeners	Public classes relative to the listener APIs
org.mockito.mock	Mock setting related classes
org.mockito.plugins	Mockito plugins allow customized of behavior
org.mockito.quality	Mocking quality related classes
org.mockito.runners	JUnit runners
org.mockito.session	
org.mockito.stubbing	Stubbing related classes
org.mockito.verification	Verification related classes

Functional structure model

The functional structure model shows how the interfaces connected together to provide all the functionalities to users. The solid pink links show the steps for users to implement a unit test. Users mock an object through org.mockito.Mockito, and then stub it or apply other actions to the object, at last verify behaviors.

The bidirectional green dash line and monodirectional blue dash line show the interaction between interfaces inside org.mockito. It means the interfaces at the origin can call the methods from the interfaces at the direction of the arrow.

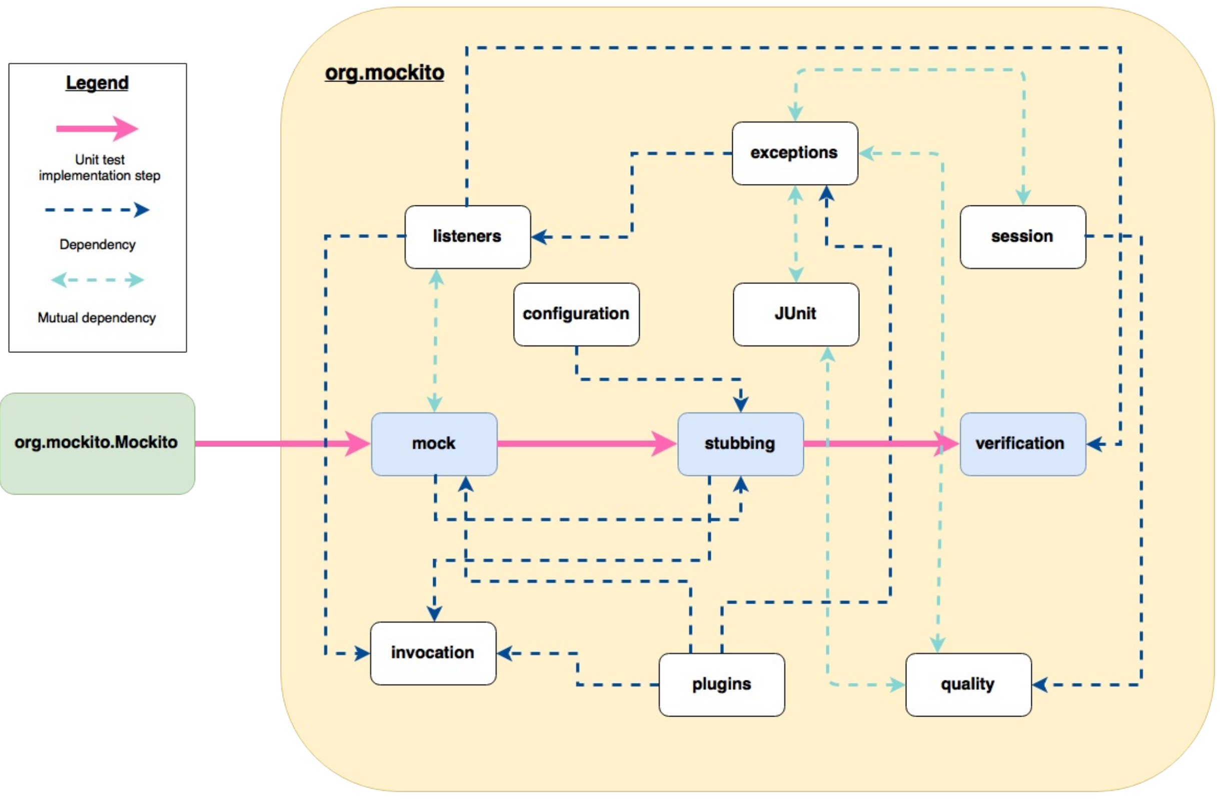


Figure 5. Functional structure model

Following clarifies how the three main functionalities interact with other modules shown in Figure 5. The module `mock` calls from `listeners` to register a listener for method invocations on this mock. In reverse, the listener is notified every time a method on this mock is called. `mock` also calls `Answer` from `stubbing` to create a mock with default answer. The module `plugins` utilizes `mock` to provide Android developers with mock creation options thus they can avoid the default byte-buddy/asm/objenesis implementation [5].

The module `stubbing` calls from `invocation` to get the stubbed method returned. Module `configuration` depends on `stubbing` to offer users options to custom the default answer. Module `verification` is depended by `listeners` for a listener to be notified by verification invocations on a mock.

Evolution Perspective [1]

Mockito has already been developed for years and open-sourced from the very beginning. At this moment, Mockito has many versions since it bases on a continuous release model. Since Mockito moved to Github circa 2012, the team wanted to automate the release process. At that time, the build script was based on Ant, after the repository migration the build script was progressively migrated to Gradle. At 2014 one core developer started to experiment continuous delivery of Mockito.

From the first version 0.9 released in 2008 to the latest version 2.7.18 released on 2017-03-18, Mockito has many releases and has been added lots of features. In Figure 6, we just highlight some important versions as example of the evolution of Mockito. The versions chosen are either some versions (or release candidates) with significant new features or the turning point bewtween two main versions, for instance, version 2.1.0 is a turning point from Mockito 1 to Mockito 2. Version 1.0 is the very beginning of Mockito 1, and after the improvements of every latter version till version 1.9.5 rc-1, Mockito 1 growed with a large number of features and functions. At this point, a brand new version of Mockito was needed, and indeed version 2.1.0 (Mockito 2) was released after that. During the period of Mockito 2.x, Mockito is using a continuous delivery model. That's why sometimes only small fixes are found at every version in Mockito 2.x period. The evolution also indicates that the framework is well developed and has a lot of features. However, bug fixes and new features are still added, and the Mockito team is preparing for Mockito 3.

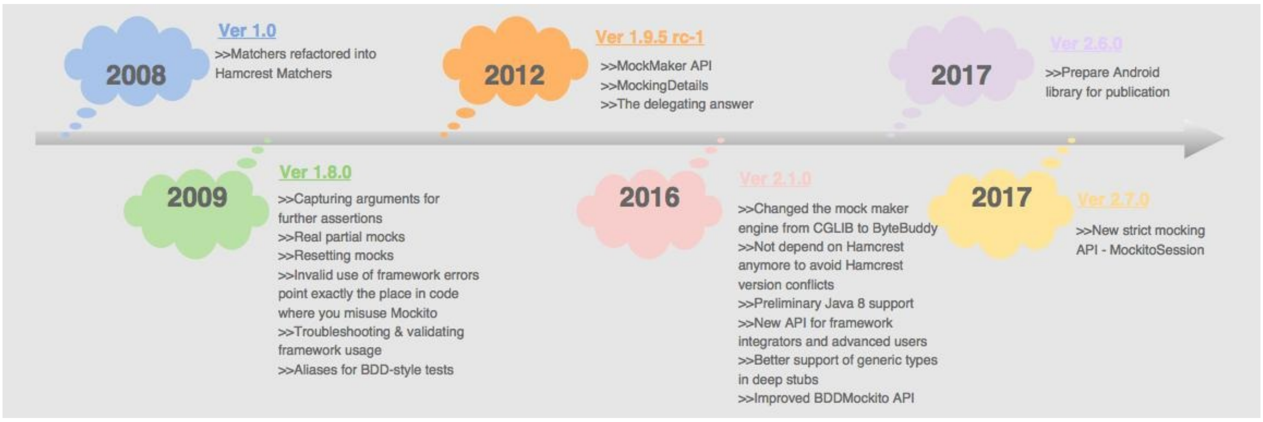

2008
Ver 1.0
>>Matchers refactored into Hamcrest Matchers
2012
Ver 1.9.5 rc-1
>>MockMaker API
>>MockingDetails
>>The delegating answer
2017
Ver 2.6.0
>>Prepare Android library for publication
2009
Ver 1.8.0
>>Capturing arguments for further assertions
>>Real partial mocks
>>Resetting mocks
>>Invalid use of framework errors point exactly the place in code where you misuse Mockito
>>Troubleshooting & validating framework usage
>>Aliases for BDD-style tests
2016
Ver 2.1.0
>>Changed the mock maker engine from CGLIB to ByteBuddy
>>Not depend on Hamcrest anymore to avoid Hamcrest version conflicts
>>Preliminary Java 8 support
>>New API for framework integrators and advanced users
>>Better support of generic types in deep stubs
>>Improved BDDMockito API
2017
Ver 2.7.0
>>New strict mocking API - MockitoSession

Figure 6. Some versions with added features as example of the evolution of Mockito.

Future Release Perspective

The current release model of Mockito is that every code change results in a new version in Central repository. Given high rate of new versions, the community is not comfortable with taking new Mockito versions at a fast pace. So, the Mockito team proposed some changes which may be implemented for the release model of Mockito 3 in early 2017.

The first-step new release plan is that release every change but not all releases go to Central library. And only substantial releases are pushed to standard repository (JCenter/Maven Center) at monthly cadence. They plan to push remaining versions to less prominent but still public repository for early adopters. Some proposals are listed below:

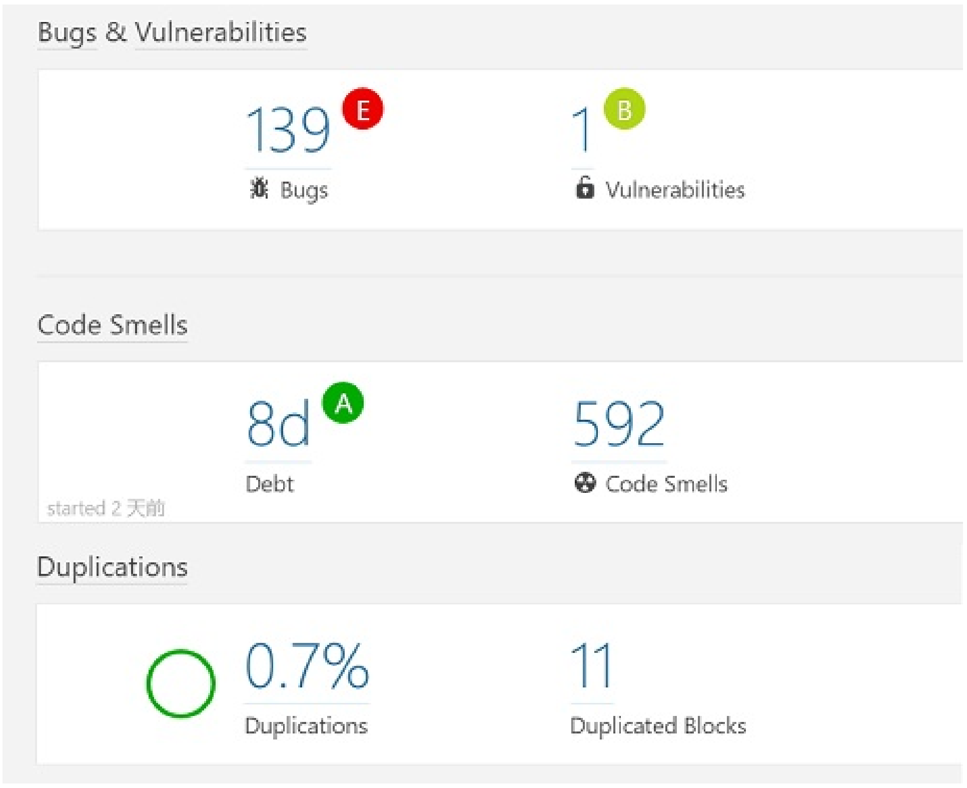
1. Every version lands in Bintray repository called "all-versions". That repository is not automatically linked to JCenter/Maven Central.
2. Every minor or major version change lands in both "all-versions" and "notable-versions".
3. For critical bugfixes, patch version change can be published to "all-versions" when commit message contains "[all-versions release]".
4. Every month, last patch version is automatically promoted from "all-versions" to "notable-versions".

Technical Debts

In this section, several tools are used to analyze the technical debts of Mockito. All the measurements are finished on 29/03/2017.

Automated inspecting code debts with tool

With assistance of SonarQube, we have an overview on technical debts of Mockito. Analysis is done with SonarQube v6.3 on the main source code `\src\main\java\org\mockito` of Mockito v2.7.21. In this section, we discuss bugs and vulnerabilities, code smells, duplications and cyclomatic complexity. Figure 7 is a screenshot of SonarQube analysis illustrating a general statistics of the technical debts.


Bugs & Vulnerabilities

139 E Bugs

1 B Vulnerabilities

Code Smells

8d A Debt

592 Code Smells

started 2 天前

Duplications

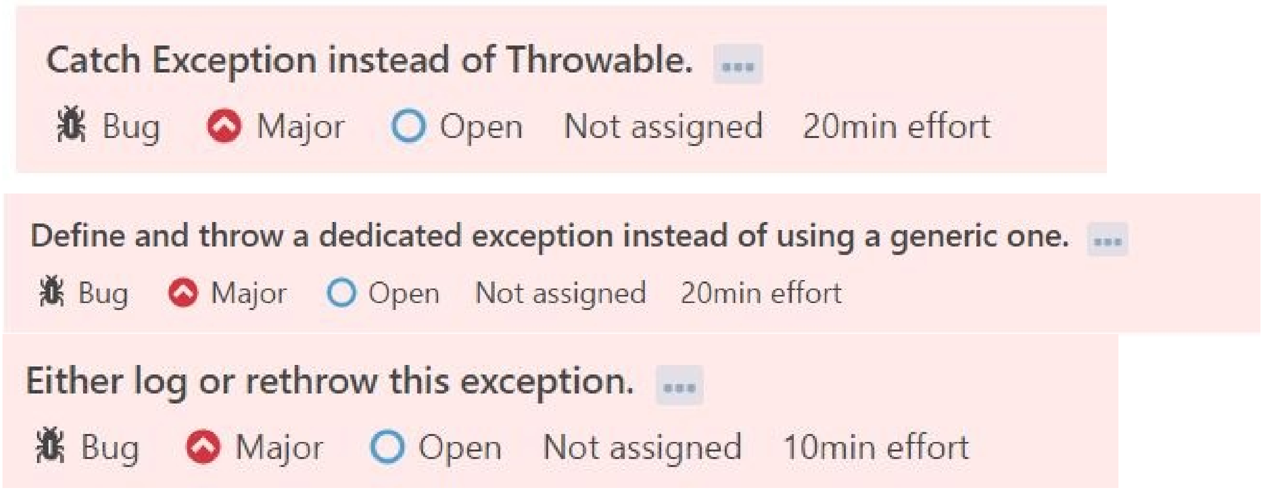
0.7% Duplications

11 Duplicated Blocks

Figure 7. An general technical debts statistics by SonarQube

Bugs & vulnerabilities

139 bugs have been found in the source code, as well as only 1 vulnerability. The total time needed to fix all the bugs and vulnerabilities is about four days and two hours, which is acceptable. Meanwhile, most of the bugs are repeated in different places of source code. Here we list the three types of bugs that are repeated most times.


Catch Exception instead of Throwable. ...
Bug Major Open Not assigned 20min effort

Define and throw a dedicated exception instead of using a generic one. ...
Bug Major Open Not assigned 20min effort

Either log or rethrow this exception. ...
Bug Major Open Not assigned 10min effort

Figure 8. Three main bugs

All these bugs shown in Figure 8 are related to the `Exception` in the source code. Thus the catch of `Exception` may lead to some problems. Besides, there are several bugs related to serialization and introduction of variables. However, the estimated effort should be taken to fix these bugs is much lower than the bugs mentioned above. The only one vulnerability is:

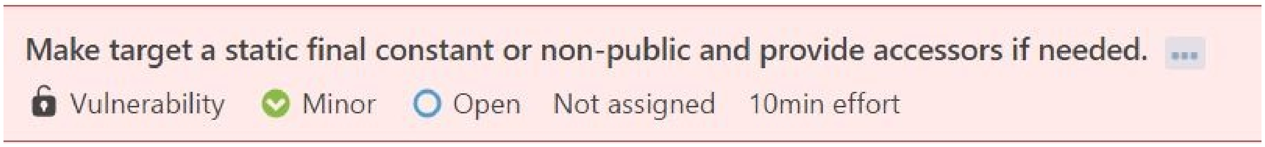
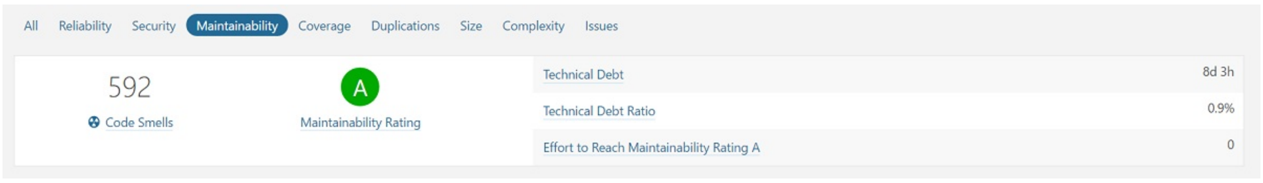

Make target a static final constant or non-public and provide accessors if needed. ...
Vulnerability Minor Open Not assigned 10min effort

Figure 9. One vulerability

It is a very small one that can be fixed using private attributes and accessor methods instead of public ones to prevent unauthorized modifications.

Code smells

SonarQube provides analysis on code smells, which are mainly related to maintainability issues in the code [6]. In Figure 10, it is shown that the main project of Mockito is highly maintainable. There are 592 code smells and the technical debts can be fixed in 8 days and 3 hours [7]. The maintainability is rated as "A".



592 Code Smells	A Maintainability Rating	Technical Debt	8d 3h
		Technical Debt Ratio	0.9%
		Effort to Reach Maintainability Rating A	0

Figure 10. An overview of maintainability-related code smells

The distribution of code smells can be viewed in Figure 11. The circles represent classes with code smells, while the size of the circle is proportional to the amount of code smells in the corresponding class. The vertical axis shows the time needed to fix the issues and the horizonal axis is the number of lines of the classes. It can be observed that all the issues can be fixed within a few hours.

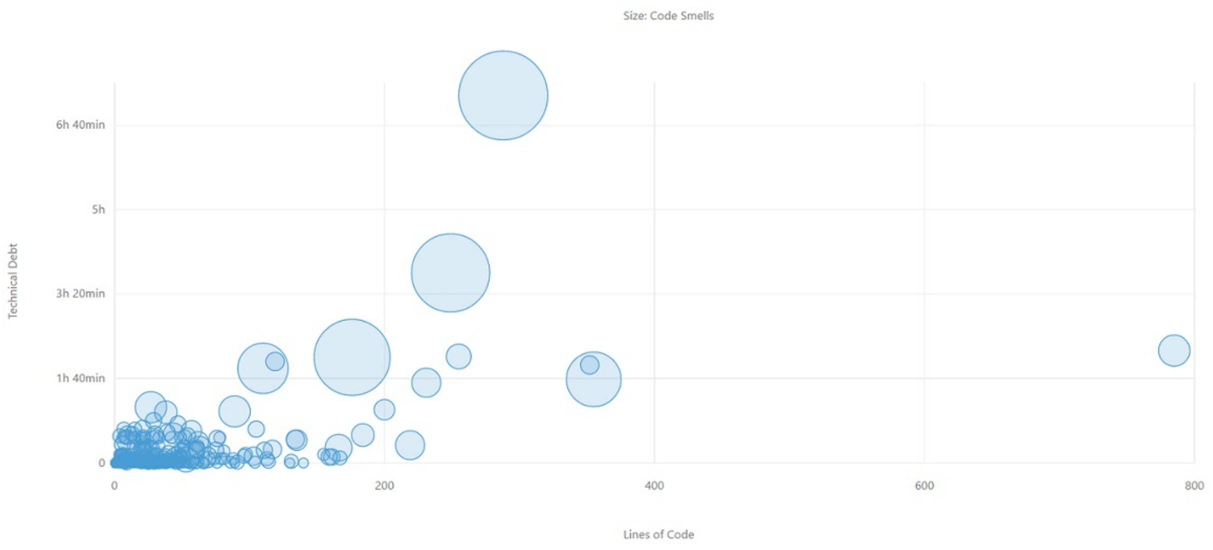


Figure 11. Distribution of the code smells

After a closer look at `/src/main/java/org/mockito/AdditionalMatchers.java`, to which the biggest circle is corresponding, it can be inferred that the situation is even better than it is reported. There are 37 code smells reported in `AdditionalMatchers`, 27 of which are of the same type and are suggested, "Remove this unused method parameter". Code section of one of those methods is given below. SonarQube argues that the parameters are not used, but actually they are got by utilizing the java method `pop()` in the argument stack in `reportAnd()`.

```java
public static boolean and(boolean first, boolean second) {
    mockingProgress().getArgumentMatcherStorage().reportAnd();
    return false;
}
```

Duplications

Duplication in the code happens when developers reuse existing code fragment by copying and adapting them. It saves time for a short term but brings difficulty in maintaining in a long run [8].
According to SonarQube, there are merely 3 files with a total of 214 duplicated lines, which means the duplication ratio of the source code of the main project is merely 0.7%.

Cyclomatic complexity

SonarQube measures the metric complexity. Its documentation says that in Java, keywords incrementing the complexity are `if, for, while, case, catch, throw, return` (that is not the last statement of a method), `&&, ||, ?` [9]. It can be inferred that this metric is about cyclomatic complexity, which indicates if there are too many branches or loops in the code. In the 14408 lines of code, the total complexity is 2793. Complexity per function is 1.4 and the distribution can be viewed in Figure 12. McCabe suggests that the limit of complexity in one module is 10 [10]. In Figure 12 there are six methods with complexity exceeding 10 and the maximum complexity is around 12, which is acceptable compared with the total amount of methods.

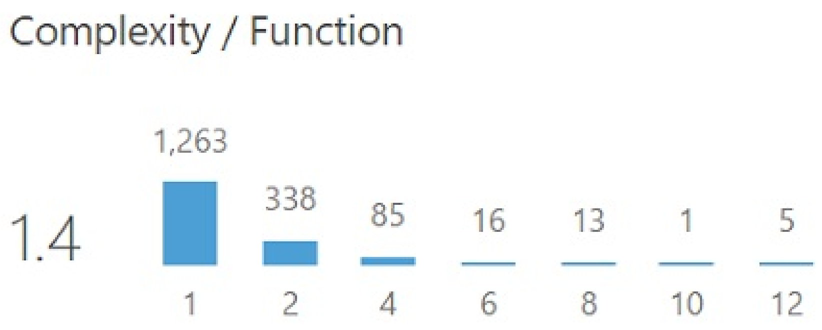
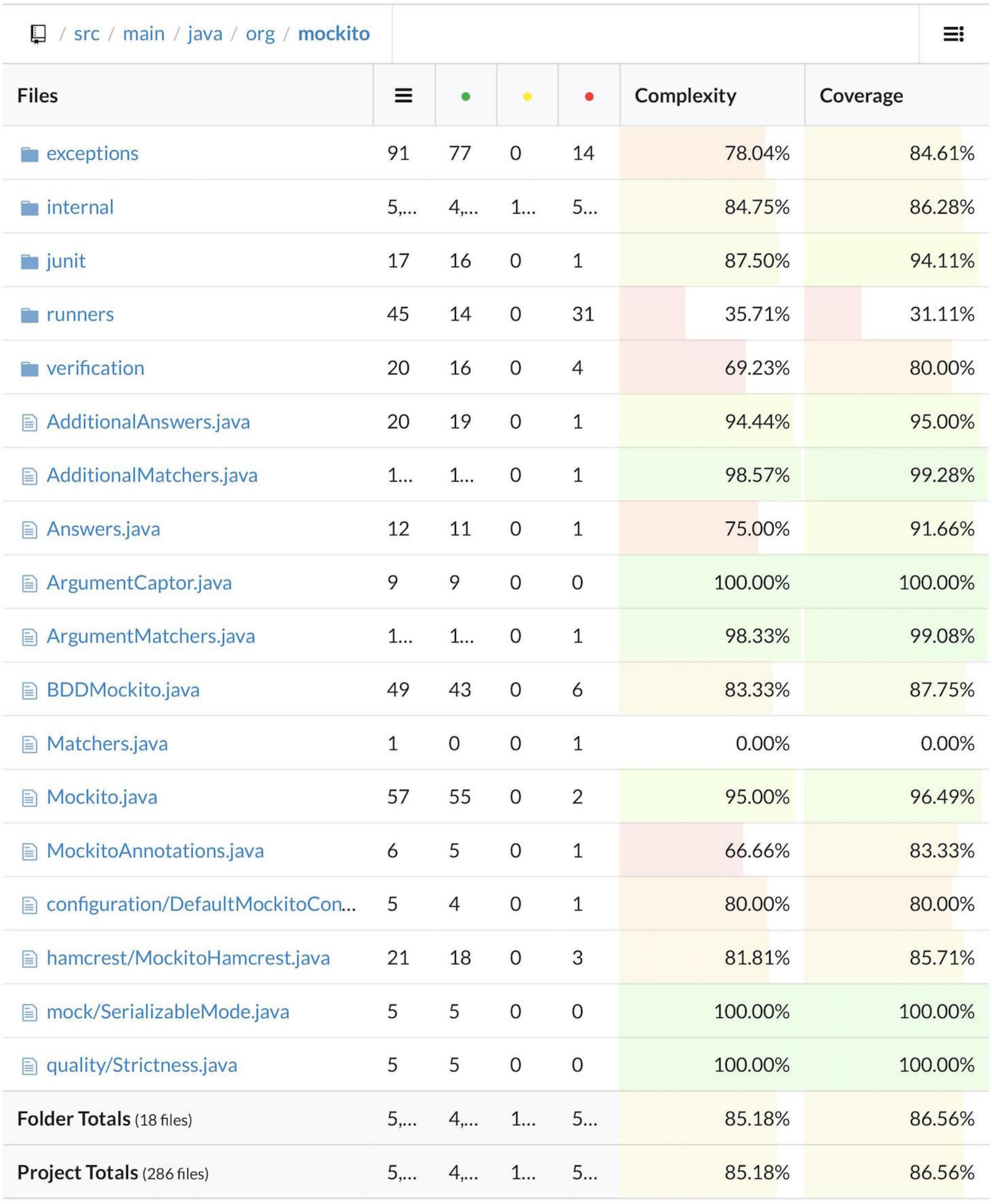


Figure 12. A screenshot of SonarQube on complexity/function distribution

Testing debts

In this section, we will look at the other type of technical debt called testing debt. This is caused by the lack of testing or by poor testing quality. In order to find the testing debt, we can either use tools or perform manual analysis on the testing. Mockito uses CodeCov [11] to generate reports about the amount of code covered by Java unit tests.

The report generated by CodeCov for Mockito shows a visualization figure in the form of a 'sunburst' shown in Figure 14. The coverage sunburst is an interactive graph that enables one to navigate into project folders in order to discover files that lack code coverage. The size of each slice is the total number of tracked coverage lines, and the color indicates the coverage (from red to green, the coverage percentage is from 70% to 100%). The center ring of the sunburst diagram shows the top level package, and each consecutive ring shows a deeper level of the package. The report (Figure 13) shows that the code coverage of Mockito is around 86.56%. This high percentage of code coverage is apparent in the sunburst with the dominating amount of the green circles.



/ src / main / java / org / mockito

Files	≡	●	●	●	Complexity	Coverage
exceptions	91	77	0	14	78.04%	84.61%
internal	5,...	4,...	1...	5...	84.75%	86.28%
junit	17	16	0	1	87.50%	94.11%
runners	45	14	0	31	35.71%	31.11%
verification	20	16	0	4	69.23%	80.00%
AdditionalAnswers.java	20	19	0	1	94.44%	95.00%
AdditionalMatchers.java	1...	1...	0	1	98.57%	99.28%
Answers.java	12	11	0	1	75.00%	91.66%
ArgumentCaptor.java	9	9	0	0	100.00%	100.00%
ArgumentMatchers.java	1...	1...	0	1	98.33%	99.08%
BDDMockito.java	49	43	0	6	83.33%	87.75%
Matchers.java	1	0	0	1	0.00%	0.00%
Mockito.java	57	55	0	2	95.00%	96.49%
MockitoAnnotations.java	6	5	0	1	66.66%	83.33%
configuration/DefaultMockitoCon...	5	4	0	1	80.00%	80.00%
hamcrest/MockitoHamcrest.java	21	18	0	3	81.81%	85.71%
mock/SerializableMode.java	5	5	0	0	100.00%	100.00%
quality/Strictness.java	5	5	0	0	100.00%	100.00%
Folder Totals (18 files)	5,...	4,...	1...	5...	85.18%	86.56%
Project Totals (286 files)	5,...	4,...	1...	5...	85.18%	86.56%

Figure 13. Coverage report of Mockito

Even though the overall coverage of Mockito is relatively high, there are still some packages that are marked in red in the sunburst. In Figure 14, we can see that in the creation package of bytebuddy, there exist several parts of inner files that are not fully covered. Figure 14 also indicates that in the debugging package, the not fully covered files exist. These problems are mentioned in issue (#904). From code coverage report (Figure 15), we can see that some classes in Mockito are completely untested. The `debugging/WarningCollector.java`, `debugging/WarningsPrinterImpl.java` and `bytebuddy/MockMethodDispatcher.java` files are in the list of totally not covered files, which explains the red-marked parts in Figure 14. This is caused by some unused classes in these files. The improvement of this may be removing those unused classes or testing them to ensure that they are working properly.

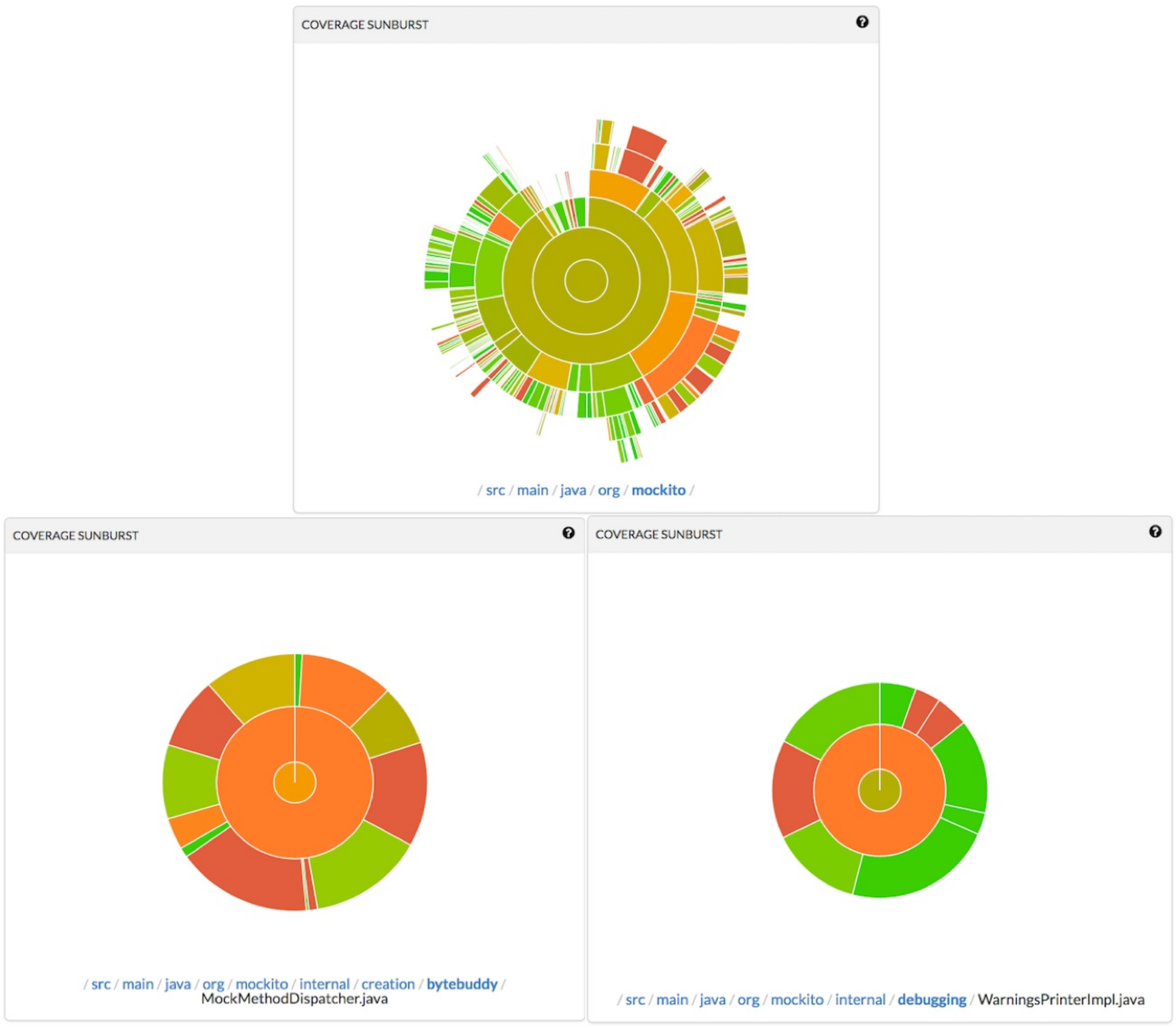
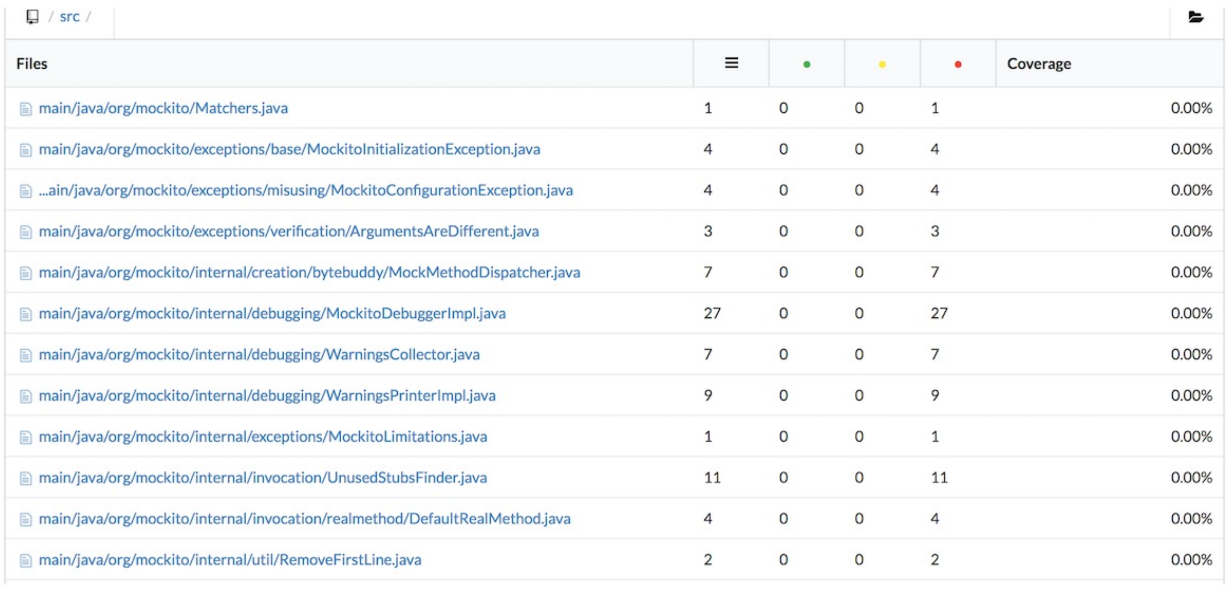


Figure 14. Coverage sunbursts of Mockito, bytebuddy and debugging files



/ src /

Files	≡	●	●	●	Coverage
main/java/org/mockito/Matchers.java	1	0	0	1	0.00%
main/java/org/mockito/exceptions/base/MockitoInitializationException.java	4	0	0	4	0.00%
...ain/java/org/mockito/exceptions/misusing/MockitoConfigurationException.java	4	0	0	4	0.00%
main/java/org/mockito/exceptions/verification/ArgumentsAreDifferent.java	3	0	0	3	0.00%
main/java/org/mockito/internal/creation/bytebuddy/MockMethodDispatcher.java	7	0	0	7	0.00%
main/java/org/mockito/internal/debugging/MockitoDebuggerImpl.java	27	0	0	27	0.00%
main/java/org/mockito/internal/debugging/WarningsCollector.java	7	0	0	7	0.00%
main/java/org/mockito/internal/debugging/WarningsPrinterImpl.java	9	0	0	9	0.00%
main/java/org/mockito/internal/exceptions/MockitoLimitations.java	1	0	0	1	0.00%
main/java/org/mockito/internal/invocation/UnusedStubsFinder.java	11	0	0	11	0.00%
main/java/org/mockito/internal/invocation/realmethod/DefaultRealMethod.java	4	0	0	4	0.00%
main/java/org/mockito/internal/util/RemoveFirstLine.java	2	0	0	2	0.00%

Figure 15. Coverage report of some uncovered files

Overall, the high coverage rate of Mockito benefits from the use of Travis CI and Codecov as part of their continuous integration process. This allows contributors and core developers to receive extended feedback on their testing performance and to keep track of their progress. In this way, they can find and improve the low coverage files in time.

Evolution of technical debts

Despite the extensive testing of Mockito, the technical debts have grown in the past few years because the scale of Mockito is getting bigger. These include the increment on the number of new features, the number of users, and the involved developers. Besides, the increased reliability on third-party dependencies is also one of the important reasons. In this section, how these factors evolved from the beginning (2008) till now (2017) is shown. The first part of this section is about the code related analysis, and the second part notes the changes on the number of TODOs in the code over this period. Most of the evidences or examples used in this section come from the release note of Mockito.

Code base analysis

The code base is the whole project of Mockito, this is much larger than what we discussed in automated inspection part. Figure 16 shows the total number of code lines and files in each version of Mockito. Both code lines and files have increased dramatically from 23660 lines and 334 files in its first release in 2008 to 81920 lines and 947 files in its latest version. From the figure, it could be noticed the number for both code lines and files were not increased stably until version 2.2. There are some fluctuations with two big modification happened in version 1.3 and 1.8. This also includes the version 1.7 that has a significant decrease on both numbers. After v2.1, numbers of both code of lines and files remain stable. It may benefit from the fixed bugs and improvements before the release of Mockito 2, and since then the technical debt decreased, which leads to relatively mature state of Mockito with minor bugs. For example, the switch of mock maker engine from CGLIB to ByteBuddy allows Mockito to fix some long-standing bugs that they had with CGLIB. More details could be found in the next part of this section *TODOs in code*.

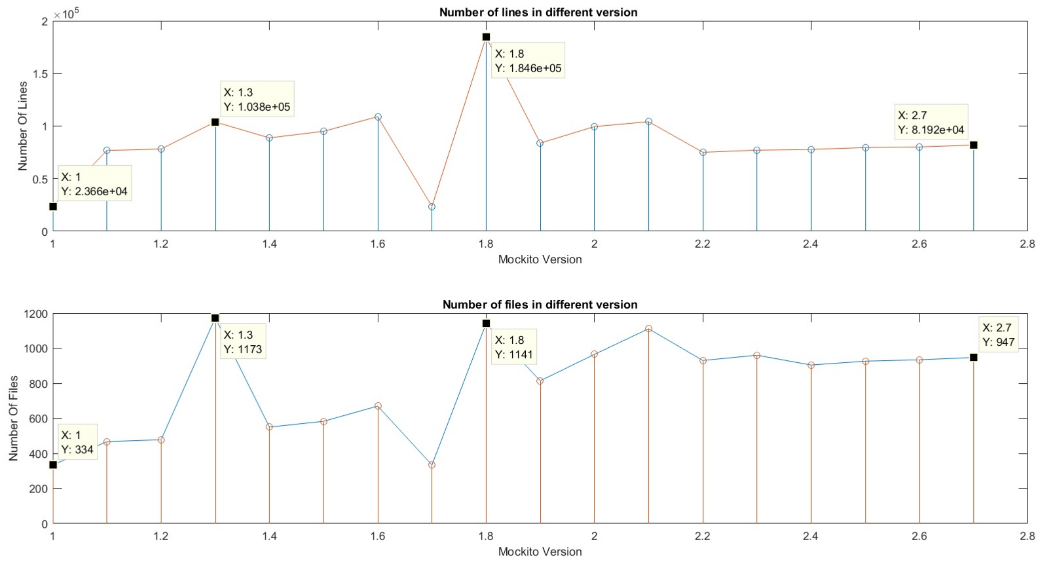


Figure 16. The scale analysis on the number of code lines and files in different versions

TODOs in code

Another important factor that can reflect the level of technical debt is the number of TODOs in code. The more TODOs in the current stage means the higher possibility of paying technical debt of the past. Figure 17 shows the tendency of the change of TODOs in each version over last 9 years. The most obvious feature in this figure is the big jump while Mockito was moving from version 1 to version 2. In that period, the works that need to be done increased dramatically. This may not directly mean the increasing of technical debts. However, the big changes of the framework layout or structure will certainly increase the frequence of occurence of technical debt. In order to find out what exactly happended in that period, The release notes is studied. Indeed, the biggest change happened in the release of v2.1. There were in total 239 improvements, 726 commits by 49 different authors and 160 remaining changes which still need to be done in the following releases (releases after v2.1).

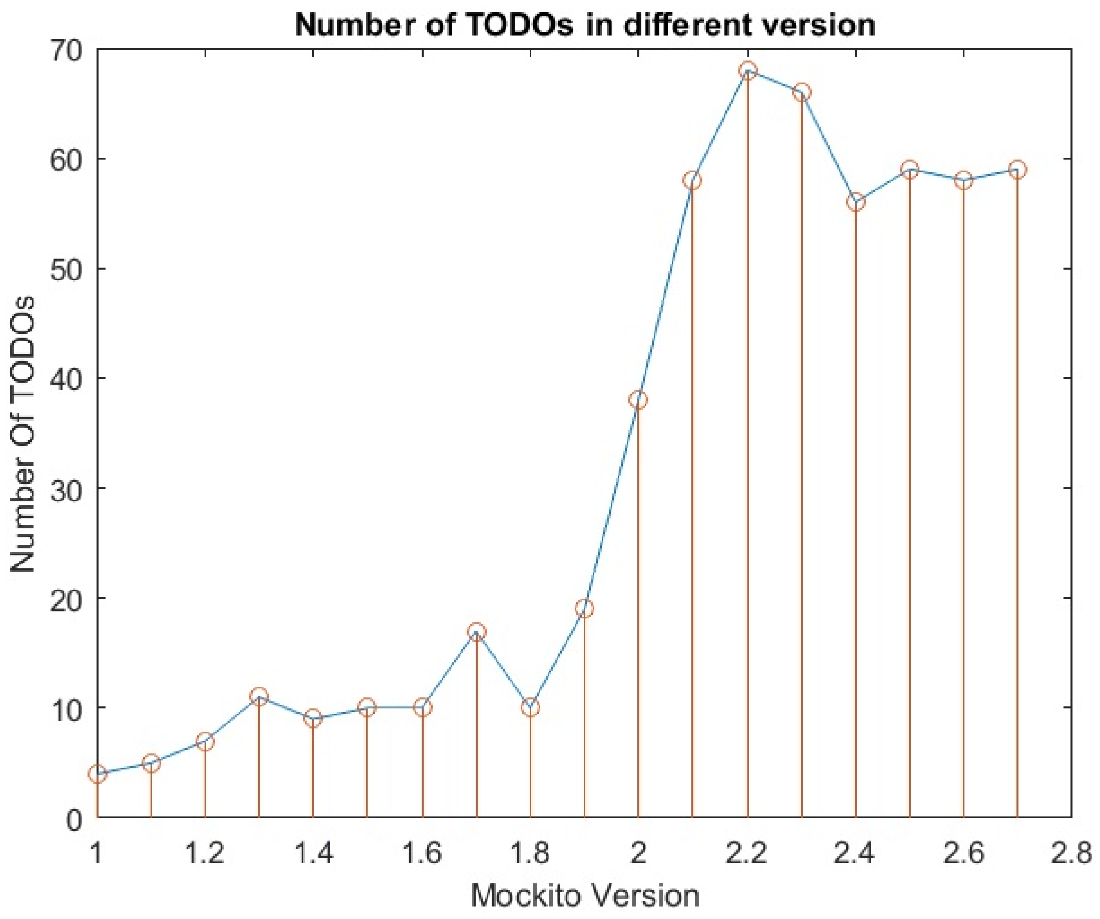


Figure 17. An overview about the changes of the TODOs number in each Mockito version

Conclusion

This chapter provides the readers with an overview of Mockito from multiple software architectural views and perspectives as defined in Rozanski & Woods's book. Conclusion can be drawn that Mockito is of a highly maintainable architecture as well as low load of technical debts.

In module structure model, Mockito is divided into four layers. Such a hierarchy is not only easy for users to utilize the functionality without knowing internal implementations, but also beneficial from a maintainability perspective. The functionalities of Mockito are around the main process of mocking, that is, mocking, stubbing and verification. Based upon this, many more user-friendly APIs are derived.

Mockito keeps a low load of technical debts although it has been existing for nine years (till April, 2017). The size of its code base has been kept stable around 80 thousand lines ever since Mockito 2. Another notable observation is that its test coverage is high as 86.56% when analyzed.

To conclude, Mockito is a well-organized open source project that is under high quality of maintaining and development. Given the healthy status with issues fixed and new features introduced at a proper pace, we believe that Mockito will become an even more popular mocking framework in the future.

References

[1] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012. Available: http://www.viewpoints-and-perspectives.info/home/book/. Accessed on: 3rd April 2017.
[2] Mockito, version 2. [Online]. Available: http://site.mockito.org/. Accessed on: 3rd April 2017.
[3] DESOSA, Delft Students on Software Architecture. [Online]. Available: https://www.gitbook.com/book/delftswa/desosa2016/details. Accessed on: 3rd April 2017.
[4] Mockito WIKI, Continuous Delivery Overview. [Online]. Available: https://github.com/mockito/mockito/wiki/Continuous-Delivery-Overview. Accessed on: 03 April 2017.
[5] Javadoc, Mockito 2.7.17 API package information. [Online]. Available: https://www.javadoc.io/doc/org.mockito/mockito-core/2.7.17. Accessed on: 3 April 2017.
[6] SonarQube Documentation, Concepts. [Online]. Available: https://docs.sonarqube.org/display/SONAR/Concepts. Accessed on: 03 April 2017.
[7] SonarQube Documentation, Computation of technical debt. [Online]. Available: https://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt. Accessed on: 03 April 2017.
[8] SolidSourceIT, "Does source code duplication matter?", 03 August 2012. [Online]. Available: https://solidsourceit.wordpress.com/2012/08/03/does-source-code-duplication-matter/. Accessed on: 03 April 2017.
[9] SonarQube Documentation, Metrics. [Online]. Available: https://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity. Accessed on: 03 April 2017.
[10] WIKI, Cyclomatic complexity. [Online]. Available: https://en.wikipedia.org/wiki/Cyclomatic_complexity. Accessed on: 03 April 2017.
[11] CodeCov, Mockito coverage. [Online]. Available: https://codecov.io/github/mockito/mockito. Accessed on: 30 March 2017.

Neovim

In order of appearance: Ioannis Petros Samiotis, Thomas Millross, Sander Bosma and Jente Hidskes.

Abstract

This chapter describes the software architecture of Neovim: an open source code editor based on Vim. This analytical essay will provide interested readers with objective and relevant insights into the challenges and architectural decisions of the Neovim development effort. Neovim's software architecture is assessed within the Rozanski and Woods [1] framework. The system *stakeholders* are detailed and categorised. Then the context and development *viewpoints* are described, followed by an analysis from the *perspectives* of variability and evolution. Finally the technical debt of the system is assessed and quantified.

Table of Contents

Introduction

Neovim is an extensible code editor, written mainly in C. The project is a fork of the famous code editor Vim. It is a modal editor, which means the keypresses are interpreted differently depending on the current mode. The main modes are *normal*, *insert* and *visual* mode. The keyboard shortcuts and commands allow users to edit and navigate through files faster than is possible with regular editors.

Early on the project established its goals of making Vim development more open to the community, refactoring the Vim codebase and removing misfeatures. The vision page mentions that Neovim is intended *"for users who want the good parts of Vim, and more"*. The community is active and welcoming to new users and contributors alike.

This chapter presents an analysis of Neovim's software architecture based on the book Software Systems Architecture by Rozanski and Woods [1]. We begin by considering the groups and individuals influencing the project through a stakeholder analysis. We then present a context viewpoint, which defines the relationships between Neovim and its environment. An in-depth study of Neovim's codebase is presented in the development viewpoint section. Next, two perspectives are explored: the variability and evolutionary perspectives. Finally, we discuss the technical debt in Neovim's codebase, before concluding.

Stakeholder Analysis

We present here a stakeholder analysis of the Neovim project. To compile information, we began by documenting our existing knowledge of the stakeholders involved and supplemented this knowledge with data sources including: GitHub issues and pull requests, Gitter conversations and the online documentation. For categorisation, we took the Rozanski and Woods stakeholder list [1] as a basis, then removed the categories irrelevant to Neovim. Finally we estimated the power and interest of each group through structured discussion within our team.

Users

Neovim **users** are ultimately the reason the project exists. Professional data scientists, sysadmins and devops experts are known to favour Vim. Users could be divided based on their skill-level into *beginners* and *advanced* users. Beginners may struggle initially with Neovim's minimalist interface, and frequently consult the help and documentation, striving to improve. The advanced group have traversed the steep learning curve, developed muscle-memory in their fingers, and memorised countless keyboard shortcuts. Design decisions for Neovim primarily emphasise the maximisation of editing efficiency for the advanced group. Beginners are encouraged to improve their skills, for instance using vim-tutor-mode.

Developers

Many people have contributed to the Neovim codebase as **developers**. The ~14 core developers include @justinmk the Benevolent Dictator For a Limited time, the lead developer @ZyX-I and @jamessan, with @tarruda recently becoming much less active. These core developers also fulfil additional roles, such as **maintainers** and **assessors**, keeping the project running whilst ensuring contributions adhere to coding standards and follow the style guide. There is not a dedicated team of **testers**; new tests are added by developers whenever significant changes are submitted.

Developers can further be divided into subcategories, in recognition of their differing concerns and motivations. **Related project & plugin developers** are essential for the growth of a healthy software ecosystem in the Neovim environment. Communication and overlap between these development teams helps to ensure that requirements are met and clean interfaces are maintained without regression. **Vim developers** may have submitted code still in use by Neovim, or recent bug-fixes that have been ported. Neovim developers also actively assist Vim developers by porting relevant patches to their codebase.

Other Stakeholders

There are no *official* **support staff** or dedicated **communicators**. Volunteers provide support through a wiki which contains the documentation, and a user manual which can be accessed through the Neovim `:help` command. Individual support is also offered through StackExchange, GitHub Issues, Twitter, Gitter, IRC, Google Groups and Reddit. Everyone who replies on these channels could be considered as support staff.

Suppliers are the stakeholders that develop distributions and packages. They take decisions on whether to distribute Neovim for their platforms and hence can potentially influence the future popularity of the software. Neovim accepts donations through Bountysource, so the donators could be regarded as **acquirers**. However, they have no formal decision making power. Finally, the **competitors** are an important stakeholder, described in the context viewpoint section.

Stakeholder Management

To visualise the relative power and interest of the stakeholders, a power-interest grid is provided in Figure 1. The power axis indicates influence in the decision making process and project direction. The interest axis represents the ongoing attention paid by a stakeholder to the project development. Stakeholders in the upper right corner should be closely managed, while those in the lower left require minimal engagement. The stakeholders in the upper left corner should be kept satisfied and those in the lower right should be kept informed about developments, to maintain their interest.

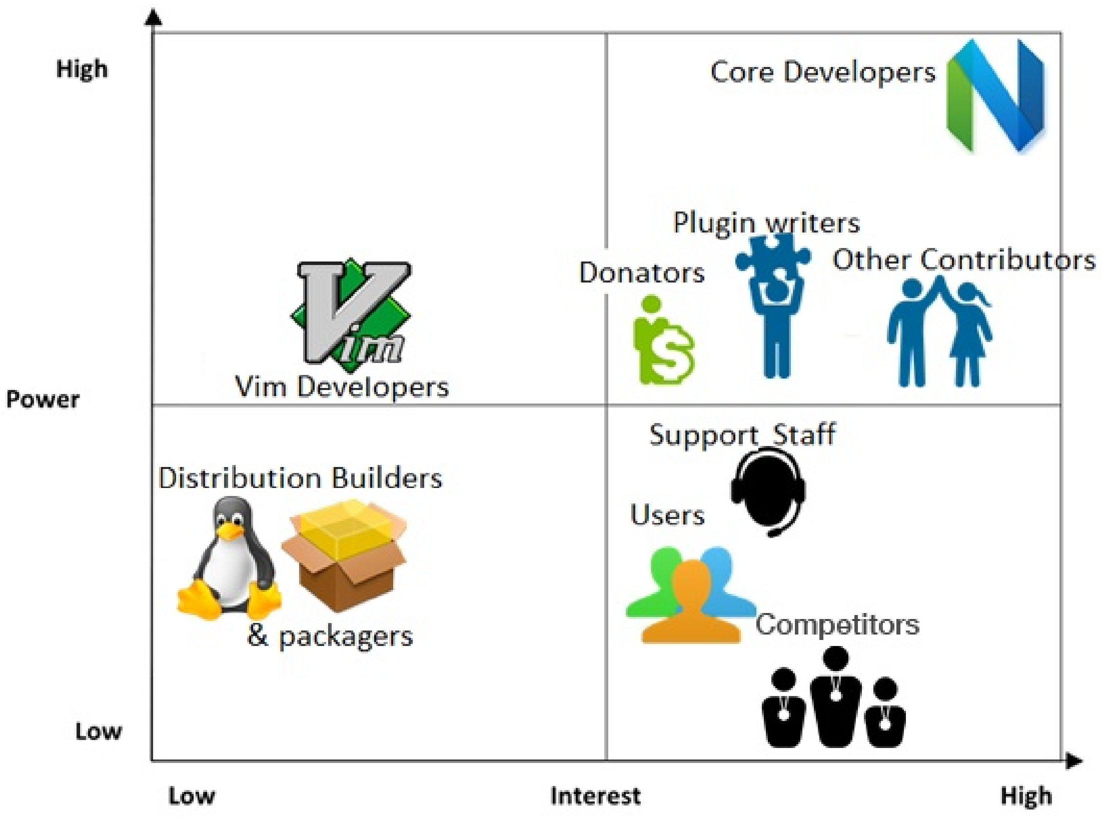


Figure 1: the power-interest grid

Context Viewpoint

The context viewpoint describes *"the relationships, dependencies, and interactions between the system and its environment"* [1]. We highlight Neovim's project scope, then use a *context model* to elaborate on the relationships with external entities.

System Scope and Responsibilities

Neovim is a code editor for those users who prefer to have a simple but powerful environment. Its responsibilities include opening text files and allowing the user to edit these files. Neovim assists the user in navigating these files and understanding the code contained within them by offering syntax highlighting and other features. An embedded scripting language called Vimscript is included, to enable extension and configuration by the user (as described in the variability perspective).

Note that Neovim is *not* striving to be an integrated development environment (IDE). While it offers the user a vast array of extensibility that could provide IDE-like features, the intention is to remain a simple but powerful code editor.

Context Model

Neovim's context model is illustrated in Figure 2. This model depicts Neovim in the centre, surrounded by the external entities it interacts with. These entities are described below, beginning at the competitors and rotating clockwise.

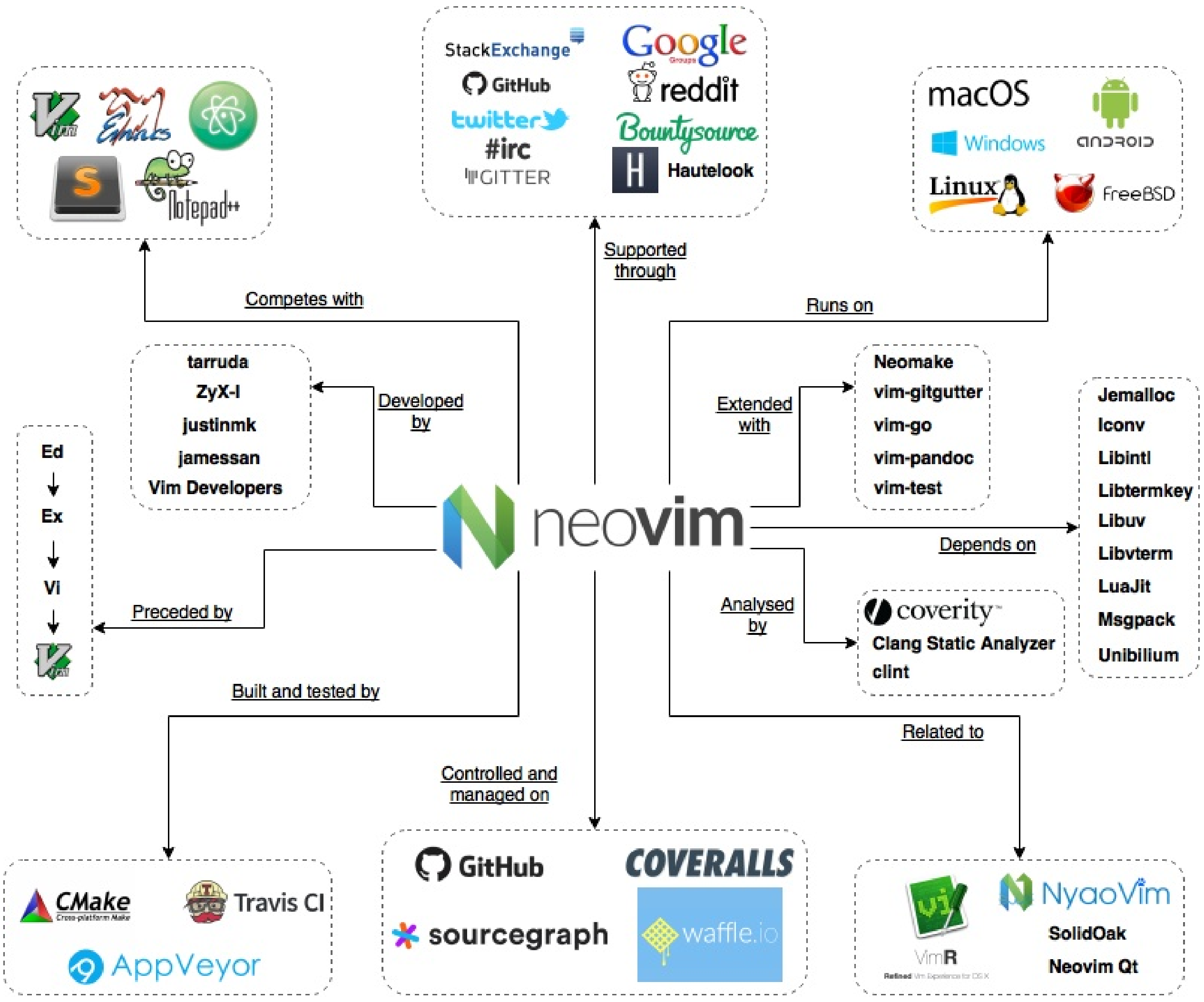


Figure 2: the context model of Neovim

Competitors

Neovim has competition from mainstream code editors such as **Atom**, **Sublime Text** and **Notepad++**. Vim and GNU Emacs are the main competitors from the same niche and require a small elaboration:

- **Vim** and Neovim are of course quite similar. Neovim introduced new features, some of which are now also found in Vim. Many differences are also found *under the hood* and in the way the communities operate.
- **GNU Emacs** is an extensible, customisable text editor -- and much more. At its core, Emacs is a Lisp interpreter that just so happens to support text editing. Plugins can

change almost everything; there is an entire ecosystem providing functionality such as calendars, project planners, PDF readers and even internet browsers.

Community

Neovim has an extensive community, known for being open and friendly to newcomers. See the stakeholder analysis for an overview.

Operating Systems

Neovim runs on a variety of operating systems: macOS, Linux, FreeBSD, OpenBSD and even Android. Neovim chose not to inherit Vim's support for obsolete operating systems. Windows support is the focus of the current release target.

Plugins

Plugins can extend the functionality of the editor or overwrite its default behaviour; detail is available in the variability perspective.

Software Dependencies

Neovim is built on top of existing C libraries that handle low level operations. These dependencies are enumerated in Table 1.

Library	Description
jemalloc	an efficient replacement for the system provided `malloc`, used to allocate memory
gettext	used to localise the interface to the user's locale
unibilium	Neovim's built-in UI is a terminal user interface. Unibilium provides information about the supported features of the terminal emulator within which it runs
libtermkey	to decode keypresses, enabling the functionality described above
libvterm	provides the built-in terminal emulator
libuv	abstracts the operating system layer
MessagePack	exposes Neovim's API, allowing clients to be written in any programming language
LuaJIT	provides support for Lua scripting

Table 1: dependencies of Neovim

Static Analysis

Static analysis tools help developers to identify bugs, style deviations and technical debt. Three different static analysis tools are run on the Neovim codebase. Coverity is the most advanced static analysis tool used by Neovim and integrates with both GitHub and Travis. Clang Static Analyzer is the static analysis tool offered by the Clang compiler. Finally, clint is used to check for coding style violations and suspicious patterns.

Related Projects

Neovim's architecture (as described in the development viewpoint) enables the creation of graphical UIs and other clients that cannot be implemented in Vim. For instance, VimR replaces Neovim's built-in terminal UI with a graphical UI native to macOS. NyaoVim also provides a modern graphical UI for Neovim using Electron, but registers itself as a client over MessagePack instead of replacing the built-in terminal UI. SolidOak is an aspiring IDE for Rust that embeds Neovim as its editor component. Finally, Neovim-Qt is a client graphical UI like Nyaovim, using Qt5 instead of Electron. Neovim-Qt will be the default graphical UI on Windows.

Source Code Control and Issue Management

Neovim's entire development infrastructure is hosted on GitHub. Coveralls is used to keep track of code coverage. SourceGraph is offered as an alternative way of browsing the code online and Waffle is offered as an alternative project management tool.

Build and Continuous Integration

Neovim replaces the autotools build system with one implemented in CMake, which is arguably easier to work with, particularly on Windows. Neovim uses AppVeyor for Continuous Integration builds on Windows and TravisCI for macOS and Linux builds.

History

Neovim comes from a long line of editors. **ed** was the line editor provided by the Unix operating system. It inspired the creation of **ex**, a line editor for Unix systems that took advantage of video terminals. **vi** was born as the visual mode for ex. Over the years, vi became the standard Unix text editor. Vim (Vi IMproved) is a clone of vi providing more features, such as syntax highlighting and an extended range of ex commands. Finally, Neovim is a fork of Vim.

Development Viewpoint

A development viewpoint describes *"the architecture that supports the software development process"* [1]. We describe a number of *codeline models*, then present the *module structure model*.

Codeline Models

This section is mainly structured after the four activities from the "Codeline Models" chapter in Rozanski & Woods [1]. Additionally, it contains a section relating to the standardisation of testing.

Folder Structure

A description of the top-level folders of the Neovim repository is shown in Table 2 below.

Folder	Description
build	generated folder containing all object files and binaries
cmake	CMake recipes to generate the Makefiles required during the building process
config	CMake and versioning configuration
contrib	useful files for contributors: autocompletion configurations, Doxygen settings, etc.
man	source for displaying manual pages (through the `man nvim` command)
runtime	all run-time data: syntax highlighting, indentation scripts, icons, in-program documentation, plugins, etc.
scripts	scripts for building, releasing, etc.
src/nvim	C source code including some related files organised into subdirectories and some unorganised loose files. See the module structure model for more information
test	testing related files; functional tests are placed in test/functional, unit tests are placed in test/unit. The structure of test/unit is identical to src/nvim
third-party	CMake recipes to build third party dependencies
unicode	files containing Unicode definitions

Table 2: description of the top-level folders

Build Approach

Neovim uses a combination of GNU Make and CMake to automate the building process. Make ensures that source code is compiled in the correct order; it uses a Makefile which lists dependencies between targets and specifies the steps required to build a target. Since it is difficult to write Makefiles that are independent of the system configuration (e.g. installed compiler and libraries), CMake is used to dynamically create Makefiles specific to the system on which it runs. Once a user has cloned the Neovim repository from GitHub, they simply execute the `make` command. This will automatically call both CMake and Make as required.

Release Process

Each time a commit is made to the master branch, Travis-CI automatically builds the Doxygen and user documentation. It also generates the Clang Static Analyzer, translation, clint, vim-patch and Coverity reports. Finally, a nightly release is made which contains the Linux and MacOS binaries. Windows binaries are generated by AppVeyor. The nightly releases are mostly *for testing*. At major milestones, an official release is tagged which contains only the source code. Users can either compile Neovim themselves or wait until distribution maintainers update their packages to provide the new version.

Configuration Management

Neovim uses the standard GitHub pull-based development model for collaboration. The project is split into multiple repositories. All files mentioned in the subsections above are located inside the Neovim repository. Related projects, such as plugin clients, continuous integration support and forks of dependencies are in GitHub repositories under the same organisation.

Standardisation of Testing

Neovim categorises test into unit and functional tests. Unit tests are compiled into a shared library and executed through the LuaJIT FFI library, which enables Lua code to use the C functions and data structures of the Neovim code. Functional tests, on the other hand, do not interface with the C code directly, but instead use the API via remote procedure calls (RPCs). While the unit tests usually only check the return values of functions, the functional tests can also check for example whether the resulting screen displays the correct state. These tests can be run locally. They are also automatically executed when changes are pushed to GitHub, together with clint to detect coding style violations and Coveralls for test coverage (as in coverage and testing debt).

Module Structure Model

A module structure model shows the organisation of the source files into modules that contain related code [1]. Such a structure provides an overview of the source code which guides developers to understand and navigate the codebase.

The module structure model for Neovim in Figure 3 displays a simplified conceptual overview for two reasons:

1. the codebase is complex and exhibits a structure which is difficult to discern due to inheriting 25 years worth of updates and changes without a significant refactoring (see technical debt). Neovim was created to address this issue, and this work is ongoing.
2. C is a low-level language, but low-level analysis does not provide the ideal overview for documenting the module organisation.

Within the model, each *box* represents a module containing the names of some of the relevant source files of that module. The arrows show inter-module relationships, and the relative height differences graphically represent the layering.

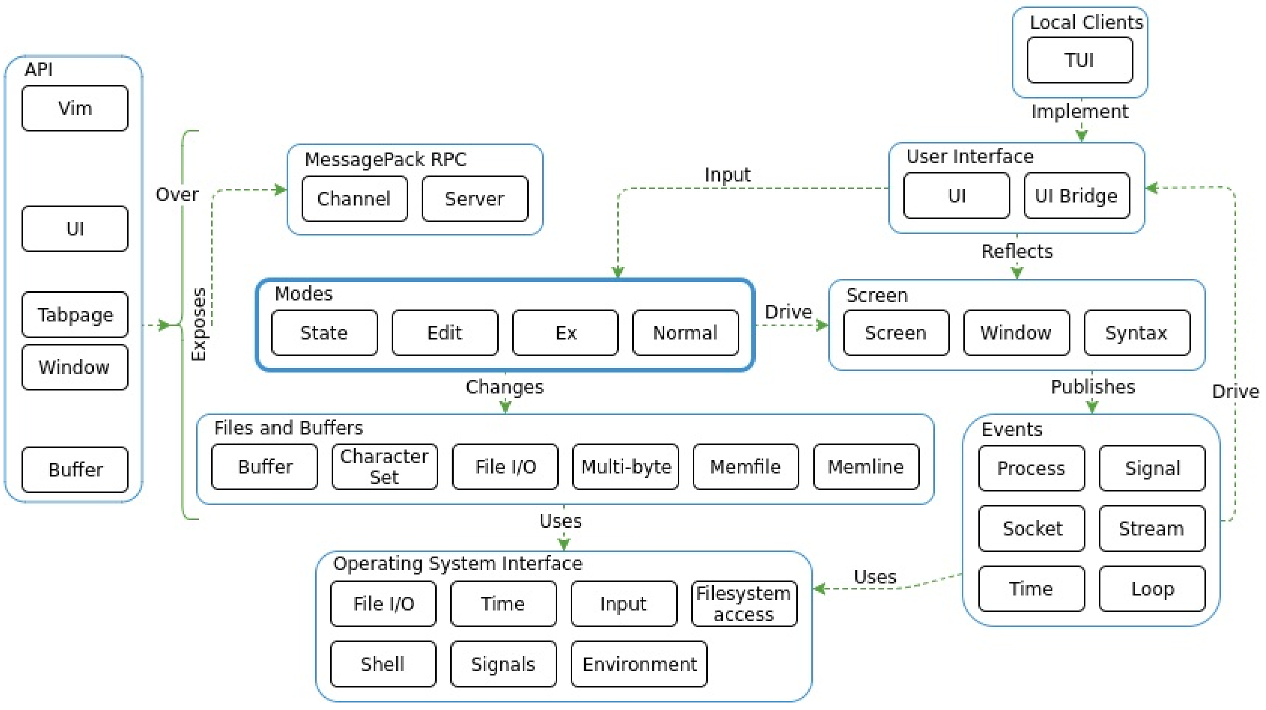


Figure 3: the module structure model

We begin with the *Modes* module, which has a thick border. Neovim models a pushdown automaton that changes state on receipt of input from the operating system. A state in this case refers to a mode (such as insert mode) or normal mode) that implements an interface. The modes guide the pushdown automaton's state changes and Neovim's behaviour, and hence the modes drive all other modules.

A mode operates on a buffer (Vim jargon for a loaded file) to edit its contents. The *Files and Buffers* module contains the code required to load files into memory, for which it needs to handle I/O, character encoding and multi-byte characters. Because I/O is inherently operating system specific, this module interacts with the *Operating System Interface* module. This module provides abstractions over filesystem access, signals, input handling and other low-level operations that vary across operating systems.

When a buffer is changed from any mode, the mode notifies the *Screen* module. This module is responsible for maintaining an internal representation of what should be visible on the UI. For each running Neovim instance, there is one such screen that displays one or more windows (a view into a buffer) in tab pages (a layout of windows). An architectural decision taken by Neovim is that all code related to GUIs is removed from the core. Instead, there is a *User Interface* module that reflects the Screen module. Synchronisation between the User Interface and the Screen module is handled through the *Events* module: the Screen module publishes events that in turn drive the User Interface module.

The *API* module exposes the User Interface and other modules over the *MessagePack RPC* module. GUIs or other clients can subscribe to this RPC channel to for example interact with the running Neovim instance, or control it over a (local) network connection.

Finally, the *Local Clients* module contains clients that bypass the RPC interface and directly communicate with the User Interface module. The built-in Terminal UI (TUI) is an example of such a local client.

Variability Perspective

This section highlights three types of software variability: compile-time, load-time and run-time variability [3]. The choice of which type of variability to utilise is a design decision which balances disk-space requirements, run-time performance and usability. Neovim uses all three types to some degree.

Compile-time

Compile-time options influence the binary created by the build process; the chosen options are compiled into the binary, the others removed. This results in a reduction in binary size, but also an increase in performance as these options now do not have to be evaluated at run-time. A classic example of compile-time variability is enabling or disabling certain feature sets, an option heavily used by Vim. Neovim has removed these feature sets and instead compiles most features unconditionally. The remaining compile-time variability options include inherently compile-time dependent OS-specific code that cannot be abstracted and the built-in TUI that can be disabled.

Settings

Neovim is highly customisable, with over 300 configurable settings such as syntax highlighting, colour schemes, indentation width, and default case sensitivity of searches. Key bindings can also be personalised for the user. These settings can all be configured at run-time or added to the initialisation file for load-time variability. As this file is interpreted as Vimscript, it can also be used for implementation of plugins.

Plugins

Neovim plugins can be broadly divided into two categories: *remote plugins* and *Vimscript plugins*. VimAwesome lists over 15000 Vimscript plugins, with the majority designed originally for Vim. Most are compatible with Neovim. After Neovim implemented asynchronicity, Vim developed an incompatible alternative implementation. Asynchronous plugins are thus an exception: they are not compatible unless support is manually added. However, most plugins are purely synchronous and can thus be used by both systems.

Remote plugins can communicate with the Neovim API via MessagePack. To simplify remote plugin development, Neovim-specific *API clients* are currently available in 17 languages, although in theory any language implementing MessagePack (of which there are over 50) can be used to develop new remote plugins. Since remote plugins are introduced by Neovim, these plugins do not work on Vim.

Inter-plugin dependence is uncommon, although some such as vim-airline are designed to integrate with others. Plugin conflicts are also rare, with the possible exception of overlapping key bindings.

External User Interface

Neovim can also be embedded into other programs through the RPC API. The external GUI starts Neovim in headless mode using a command line parameter (another form of load-time variability). Instead of drawing the screen, Neovim sends the screen state over the RPC API to the external UI. Conversely, key strokes are received by the external GUI and sent across the API to Neovim. The RPC layer can also by bypassed directly by replacing the built-in TUI with another UI (see VimR in related projects).

Evolutionary Perspective

The evolutionary perspective takes a view on the ability of the system's architecture to be flexible [1]. We take a slightly different approach and consider the changes required of Vim's architecture from the perspective of @tarruda when he decided to fork Vim. We use the following process, adapted from Rozanski & Woods [1]: characterise the evolution needs, assess the current ease of evolution and rework the architecture. We conclude by evaluating the extent to which @tarruda's architectural goals have been met.

After two decades of evolution, Vim had accumulated a complex codebase that few people understood. Bram Molenaar was possibly the only person capable of maintaining Vim's codebase, leaving him reluctant to merge new features due to the risk of regression. Vim was thus unable to keep up with the improvement and evolution demands of its users.

@tarruda realised this and decided it was time to rework Vim's architecture to increase its flexibility and reduce the time taken to implement new features. He characterised Vim's evolutionary requirements using his prior experience contributing to Vim, as in Table 3.

Change	Type	Magnitude	Timescale
Simplify maintenance to support new features and make it easier to provide bug fixes	Functional	Large-scale, high risk	Not immediately required, rather a long-term goal
Enable the implementation of modern user interfaces separate from the core of the editor	Functional, environment	Large-scale, high risk	Required sooner rather than later
Increase extensibility (see the variability perspective)	Functional, environment	Large-scale, high risk	Required almost immediately

Table 3: Neovim's change requirements when forked from Vim

For each change, @tarruda specified a plan to help in estimating the ease of evolution:

1. To simplify maintenance, Neovim would migrate to a CMake-based build system, remove legacy support and compile-time features and remove platform-specific code in favour of libuv. This is a difficult and high-risk requirement that would change the codebase as a whole. Luckily, much of it could be automated using existing tools.
2. To enable modern user interfaces and increase extensibility, Neovim would implement a job control mechanism over MessagePack. This is again a difficult requirement that would require substantial refactoring of the codebase. However, there was already prior work and experience from attempting to bring asynchronous plugins to Vim.

Since then, Neovim's developers have refactored Vim's architecture in a step-wise fashion, limiting changes to distinct sub-systems. They strived to encapsulate functionality within well-defined modules (see the module structure model) with separate concerns. Where possible, these modules were abstracted behind interfaces. A focus on unit and functional tests within a continuous integration framework helped towards ensuring the changes were reliable.

Evaluation of the Initial Project Goals

Neovim is now three years old; how have these initial goals been met? In November 2016, Neovim developers stated that at least 20,000 new lines of C code had been written, with 2200 new tests in addition to Vim's own test suite. By that time, 273 contributors had committed more changes than Vim had received in twelve years. For a complete and up-to-date overview of differences between Vim and Neovim, see the vim-differences manual page.

The asynchronous plugins and built-in terminal emulator are prime examples of new features that have been implemented. All the work on the foundation is just now starting to bear its fruit with the appearance of many external GUI clients and several clients making use of the asynchronicity to implement features previously impossible in Vim. As new features have been implemented without the addition of a significant support burden, it is clear that the changes have led to simplified maintenance requirements. All of this is not without technical debt, however, as the following section will show.

Additionally, Vim 8.0 was announced in December 2016 as the first release in ten years, sporting many of the features initially found only in Neovim. This suggests that the new competition was the inspiration and motivation for this vital renewal in development effort.

Technical Debt

This section explores technical debt in the Neovim codebase. Before entering into the details, let us first consider the perspective that the Neovim project exists partly as an answer to the technical debt accumulated by the Vim project. This blog describes examples of technical debt found in Vim's codebase, before highlighting Neovim as the solution.

It is clear that Neovim developers understand Neovim within this context. The lead maintainer @justinmk demonstrates his familiarity with the technical debt metaphor on numerous occasions (1, 2, 3, 4). Furthermore, on the progress page in the Wiki a lot of items are listed that are directly or indirectly related to paying technical debt. Finally, during a period of limited new features, in response to the question of "is Neovim in maintenance mode?", he responds "Yes. We're paying down technical debt".

Coverity Scan Static Analysis

Coverity Scan is an advanced static analysis tool that helps developers find and fix defects. It is executed periodically to scan Neovim's source code. At the time of writing, there are 89 outstanding issues out of a total of 458 found since the project's inception. 40 of these are dismissed as false positives and 329 have been fixed. The last scan (March 5, 2017) concluded a defect density (defined as the number of defects per 1,000 lines of code) of 0.33.

Some of the 89 outstanding issues were first detected in 2014. All open Coverity issues can be categorised as technical debt, since all of these defects are known but nobody has managed to fix them (or determined if they are false positives).

In Figure 4 we present the defect density of Neovim over a period of time as generated by Coverity. As can be seen, Neovim's complexity fluctuates around the average defect density of open source software projects of similar size (between 100,000 and 499,999 lines of code). The spike visible around January 2017 is likely a misconfiguration or something going wrong on Coverity; a defect density of 14 would mean there are total of 3500 defects, which does not correlate to the amount of changes made in this period.

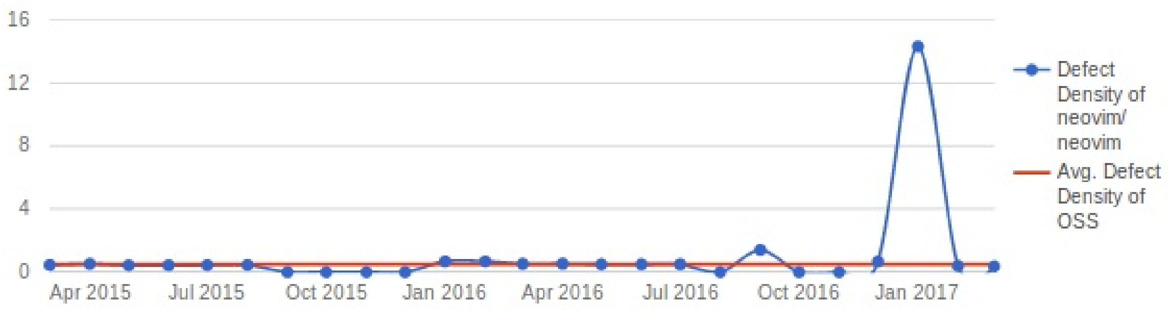


Figure 4: Neovim's defect density over time, courtesy of Coverity Scan

Cyclomatic Complexity

A popular and well-studied [3] metric for measuring code complexity is the Cyclomatic Complexity metric (CC). This quantifies the number of independent paths through the source code of a particular function. The explanatory power of CC is frequently challenged [4, 5]. However this discussion is beyond the scope of this essay, and we will assume the metric has scientific validity and informational value with regards to *technical debt* for the remainder of this section: a high CC is an indicator for technical debt.

Coverity marks all functions with a CC of greater than 15 as being too complex. The Coverity scan of March 5 2017 marked 728 functions as being too complex, out of 5128. The average CC of all the functions is 10, while the average of the functions marked too complex is 47. The highest CC found in Neovim is 728. While the threshold of 15 is debatable [6], it is clear that some of Neovim's functions are way too complex.

"ToDo" Style Placeholders

A complementary approach to measuring technical debt is counting the occurrences of "TODO", "FIXME", "XXX" et cetera within the codebase. This approach relies upon the commenting habits and conventions of the developers, and does not provide a comprehensive understanding of the technical debt of a project. However, because Neovim has had mostly the same group of maintainers over its lifetime, we assume that this metric provides some insight into the evolution of technical debt in Neovim.

The occurrences were counted by searching for the strings mentioned above, which were chosen from looking through comments and searching for common "ToDo" style variants. The results were inspected and the following files were excluded to remove many false positives: `build/*` , `runtime/syntax/*` , `*.txt` and all binary files. It is uncertain whether all categories contribute equally to technical debt, but quantifying their difference is not simple. Therefore, Figure 5 only displays the sum of all these occurrences for various versions of the project.

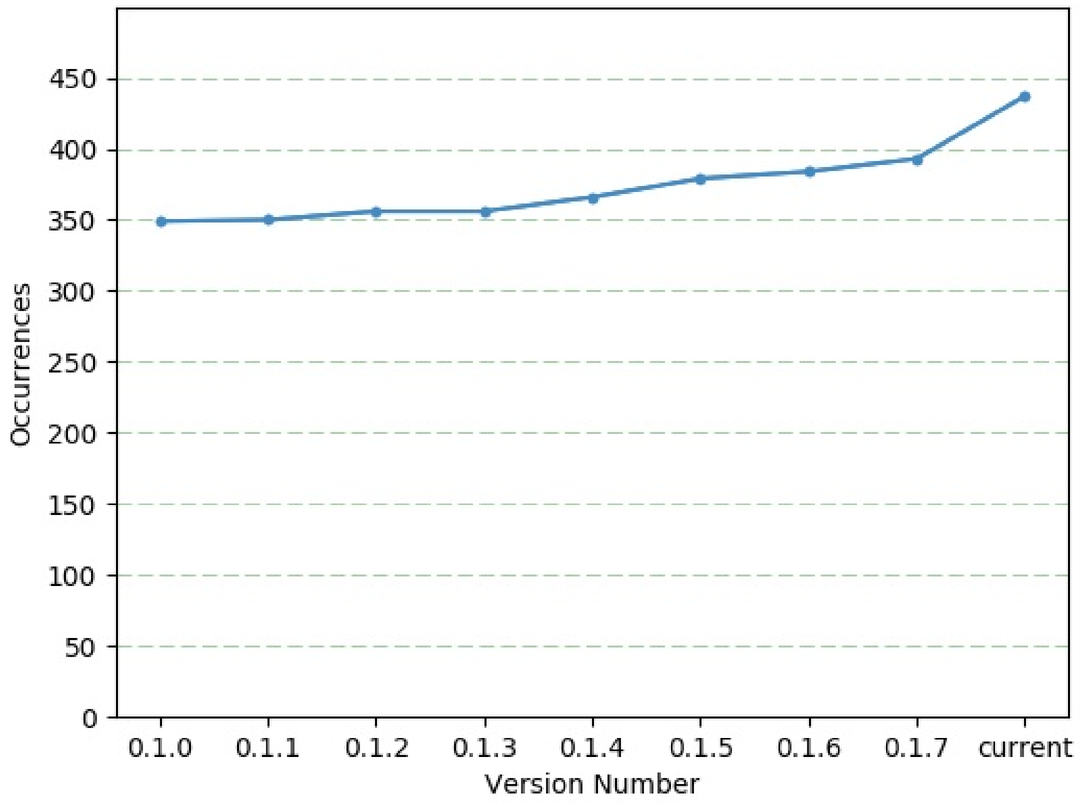


Figure 5: Number of occurrences of "TODO", "FIXME" and "XXX" over several versions of the project

It can be seen that the technical debt gradually seems to increase as the project evolves. A sharp increase can be observed in the current (unreleased) version, although this number may decrease before an official release is made.

Coverage and Testing Debt

Neovim uses Coveralls to keep track of the total test coverage. From Coveralls we can observe that coverage increased gradually from 57% in 2014 at Neovim's inception to 76% in March 2017. This can be explained by Neovim's policy of adding tests whenever significant changes are made.

Instead of identifying testing debt in the actual tests' code, we can identify a rather large case of possible testing debt on a higher level. "New style" tests imported from Vim are written in Vimscript, whereas tests made for Neovim are written in Lua. The gain from rewriting these tests is small in comparison to the effort required and hence they are imported as-is. The result is having tests written in two different programming languages, which can be considered testing debt.

There are also "old style" legacy tests that are left over from the fork of Vim. These are written using yet another framework and are actively encouraged to be rewritten using Lua, as can be seen from the many pull requests named "Migrate legacy test", e.g. #2988. In this sense, Neovim is working to pay off some of its testing debt.

Conclusion

The goal of this chapter was to present Neovim and examine its software architecture. We did so by documenting the stakeholders, giving the context and development viewpoints, considering the variability and evolutionary perspectives and discussing Neovim's technical debt. We have shown that despite the clean simplicity of the UI, there is significant complexity hiding under the surface. Challenging architectural decisions have been taken in order to satisfy a variety of stakeholders.

Through our research, we have discovered a conscientious community of hardworking volunteers, dedicated to improving the text editor they love. The Neovim codebase, like any other, is not perfect. There are still issues and ongoing improvements. However, we may conclude that the Neovim team has improved the Vim experience and brought Vim into the twenty-first century.

We would like to thank:

- @justinmk and the community for actively welcoming new contributors, assisting with our contributions and clarifying some technical details for the *module structure model*

- the dedicated teaching assistants @sandervdo and @valmai for their invaluable feedback and encouragement
- our peers that reviewed the chapter prior to completion, helpfully highlighting areas of improvement
- professors @avandeursen and @azaidman for designing and managing such an original and valuable course and assignment.

References

1. Rozanski, N., & Woods, E. (2011). Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.
2. Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). Feature-Oriented Software Product Lines: Concepts and Implementations. Springer.
3. McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.
4. Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. Software Engineering Journal, 3(2), 30-36.
5. Graylin, J., Hale, J. E., Smith, R. K., David, H., Kraft, N. A., & Charles, W. A. R. D. (2009). Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. Journal of Software Engineering and Applications, 2(03), 137.
6. Watson, A. H., Wallace, D. R., & McCabe, T. J. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric (Vol. 500, No. 235). US Department of Commerce, Technology Administration, National Institute of Standards and Technology.

Netty: Asynchronous Event-Driven Network Application Framework

Ade Setyawan Sajim, Muhammad Ridho Rosa, Priadi Teguh Wibowo, Mohamat Ulin Nuha

Delft University of Technology, 2017

Abstract

Netty is an asynchronous event-driven network application framework in Java which has been used by many large companies, such as Facebook, Apple, and RedHat. Netty Project has been actively developed since 2004. It is developed by a team of core developers and hundreds of external contributors. This chapter illustrates the software architecture of Netty in multiple parts. It starts with an explanation about Netty's stakeholder and followed by the description of four viewpoints (context view, development view, deployment view and functional view) in Netty. The chapter is then continued by identification of technical and testing debt and presentation of two perspectives (evolution perspective and performance-scalability perspective). Lastly, a conclusion from the process of analyzing Netty is presented.

1 Introduction

When developing networking applications, developers usually dreams of having a high performance and high throughput in their software [25]. Unfortunately, having control over networking code means lots of hard work on networking primitives. The problem gets even worse if security and scalability issues are also considered. A networking framework that tries to solve this issue is Netty. Netty is a non-blocking I/O (NIO) client-server framework which eases network programming and enables fast construction of network applications [2]. Netty has been designed thoughtfully with the experiences gained from the implementation of multiple protocols to ensure ease of development, high performance, stability, and flexibility without a compromise [2].

In the following section, the result of analysis on Netty is shown. The analysis was done using the guidelines from Nick Rozanski and Eoin Woods [1]. The analysis starts by explaining about stakeholders and context view (section 2 and 3). To have a deeper insight on Netty's technical side, the analysis continued by describing on Netty's development view, deployment view, and functional view (section 4, 5, and 6). Some drawbacks inside Netty's system is presented on technical and testing debt sections (section 7 and 8). To understand Netty even further, two extra perspectives, evolution perspective, and performance-scalability perspective, are presented on section 9 and 10. Finally, section 11 concludes our analysis.

2 Stakeholders

Stakeholders are an individual, team, organization, or classes who have interests and concerns about a project [1]. List of Netty stakeholders can be seen on table 1.

Type	Representation	Description
Acquirers	Founder	Acquirers supervise the acquirement of the system or product. Trustin Lee is considered to be the acquirer for Netty Project as he is the Netty founder.
Assessors	Core developers	The core developers in Netty since one of their jobs is to ensure that all contributors sign Individual Contributor License Agreement or Corporate Contributor License Agreement
Communicators	Developers and the community	In Netty, the communicators are core developers, especially Trustin Lee, Scott Mitch and Norman Maurer. Besides developers, the community can also be communicators since they can contribute by spreading information about Netty through creating articles and Netty tutorial documentations.
	Core developers and	One way to search for Netty's developers is to look at contributor section in graphs tab on Netty's GitHub. Such section shows the commits and the contributions that could be used to detect the developers. To see Netty's core

Developers	developers and external contributors	developers, a look up on The Netty Project members page. From the mentioned page, it can be found that there are fifteen core developers in Netty. Between those developers, top four contributors are Trustin Lee, Norman Maurer, Scott Mitchell, and Frederic Bregier.
Maintainers	Core developers and external contributors	The maintainers are the core developers and external contributors because they control the development of the system since it starts working which can be seen from pull request and issue analysis.
Production engineers	Core developers	In Netty, production engineers are the core developers. Their role is to plan, deploy, and maintain the software environments in which Netty will be developed and examined.
Suppliers	Libraries which Netty depends on, Java, IntelliJ	An example of a supplier is libraries because they help how Netty constructed. Java is also a supplier since Netty is developed in Java. IntelliJ is also an instance of Netty's supplier because IntelliJ is an IDE on which Netty is currently developed and officially supported.
Support Staffs	Community, core developers, external contributors	They communicate and solve problems via Github, Stack Overflow, Twitter, IRC, RSS, or Google Group. GitHub serves as a mean to discuss issues, primarily bugs and performs code reviews. Twitter is used as media social interaction. StackOverflow is a place of discussion between developers. RSS is used to publish updated information regarding the project frequently.
Testers	Core developers and external contributors	By looking up Netty's pull request section at its GitHub repository, testers of Netty project can be identified.
Users	Any organizations that use Netty and personal users	Beside personal users, companies that use Netty can be identified by looking at related article that featuring Netty. Some of the identified companies are Apple, Facebook, and RedHat.

Table 1. Netty Stakeholders

Besides several types of stakeholders above, there is a category that is not included in the table. This category is not found in [1] but related to the Netty project. Below is the extra stakeholder which is identified:

- Funders/Donators

monetary fund or giving donations. Netty project gets the donation from several companies such as Clinker, Spigot, Twitter, JetBrains, and Yourkit [26].

2.1 Power/Interest Grid

After identifying several types of stakeholders, an analysis of the power/interest of each stakeholder can be performed. The power/interest prioritization can be split into four categories:

- **High power, high interest**: These people need to be managed closely since they have the power to halt or change the development of the project or add new functionalities and have a high interest in the development of the project. In Netty, the founder who also acts as the acquirer and core developers are included in this category.
- **High power, low interest**: This category includes the people who required to be kept satisfied in the development of the project. They have high power in the development of the project but low interest in it. Suppliers are included in this group because any changes on their system will affect Netty development but any Netty development has a minor impact on their system.
- **Low power, high interest**: Any persons or organizations who need to be informed about the development of the project are included in this category, for example, any companies that use Netty, personal users, and funders/donators. External contributors, who also act as communicators, also falls between this category and "high power, high interest" category since some of them contribute a lot to Netty which makes their influence significant in Netty development.
- **Low power, low interest**: This category consists of people or organizations who required to be monitored but with minimum effort. Competitors fall between this category and "low power, high interest" category because some of them care about how to be a better solution than Netty.

The following is the power/interest grid figure for the selected stakeholders

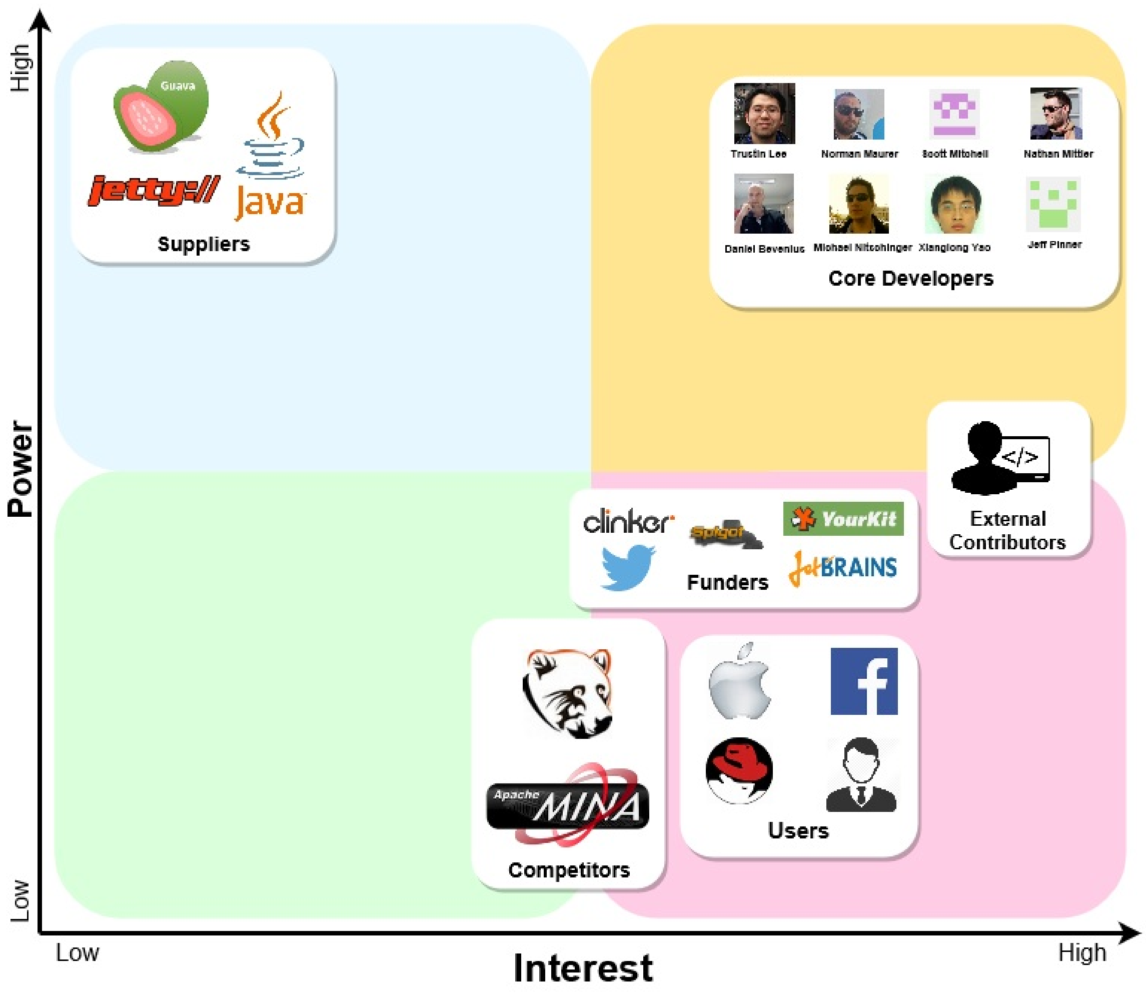


Figure 1. Netty's Power/Interest Grid

3 Context View

Context view explains the behavior of the system and illustrates the relations, dependencies, and interactions between the system and its environment [1].

3.1 System Scope

Netty as the asynchronous event-driven framework which supports several protocols has the advantage of the development of network application between server and clients. Netty purpose is to become an easy and quick NIO client server framework network application which support several protocols such as FTP, SMTP, HTTP, XML, and HTTP2.

Design Philosophy

Trustin Lee mentioned in his presentation that Netty design philosophy is to make a very simple network application and highly performing application which is also maintainable [13].

3.2 External Entities

External entities in Netty project can be divided into three parts, competitors, software platform & dependencies, and development & community.

Competitors

Competitors are the organizations who are engaged in competitions with each other. The competitors of Netty project are Apache Mina and Grizzly. Both of them are focusing on developing NIO client server framework network application.

Software Platform & Dependencies

In this section, the platform and the dependencies of Netty will be discussed.

Platform

Netty is compatible with any platform as long as it could run JDK because Netty is designed on top of Java platform.

Dependencies

Netty depends on many libraries. Between those projects, some of them are used in testing environment, i.e., Mockito, JUnit, and Java Hamcrest.

Beside libraries, Netty also depends on several entities, for example IntelliJ as its officially supported IDE. Netty also rely upon GitHub as its version control and issue tracker. To ease build activity, Netty also utilizes Maven as build automation tool and Travis CI as continuous integration tool. Another instance of Netty dependency is shown by its license usage since Netty is distributed under Apache License v2.0.

3.3 Context View Diagram

In this section, the context model of Netty is presented by combining every information provided in stakeholder section and previous subsections. The relationship between external entities and its interfaces or context view graph of Netty is visualized by figure 2.

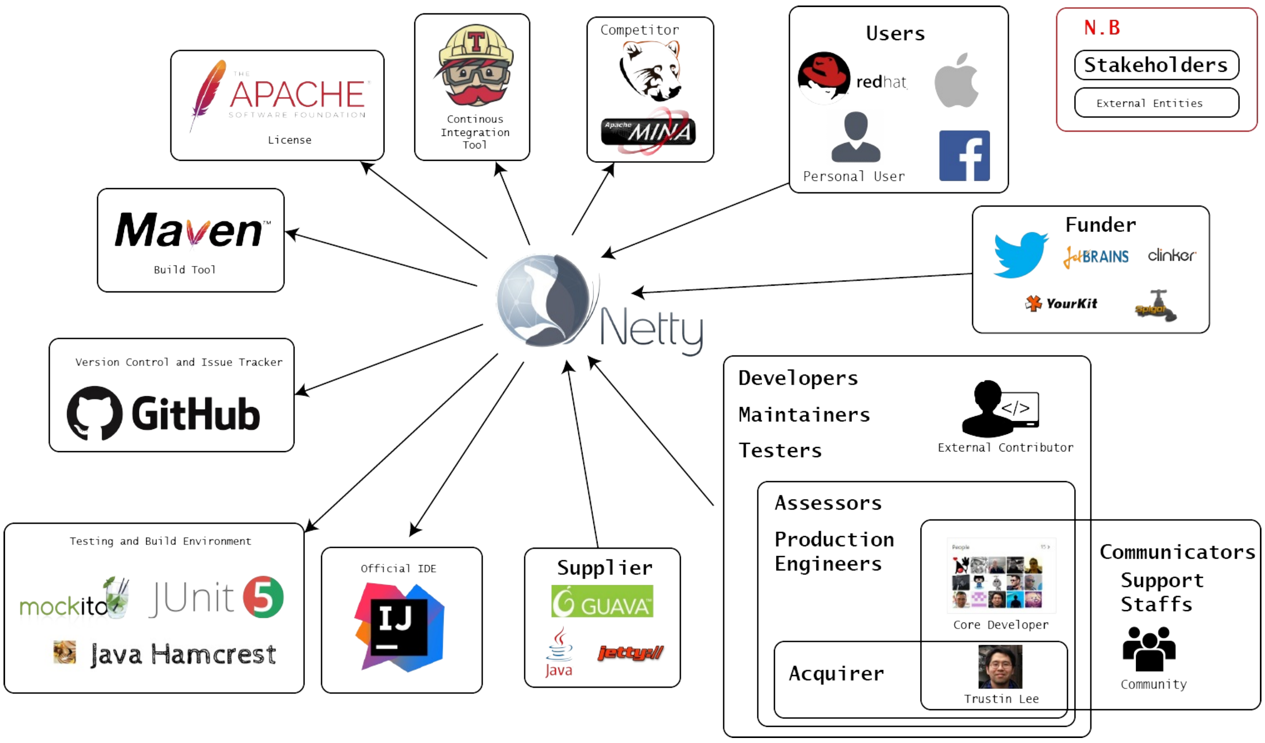


Figure 2. Context view graph of Netty

4 Development View

In this section, the architecture that addresses the aspects of system development process in Netty is explained. This section consists of three subsection; module structure models, common design models, and codeline models.

4.1 Module Structure Models

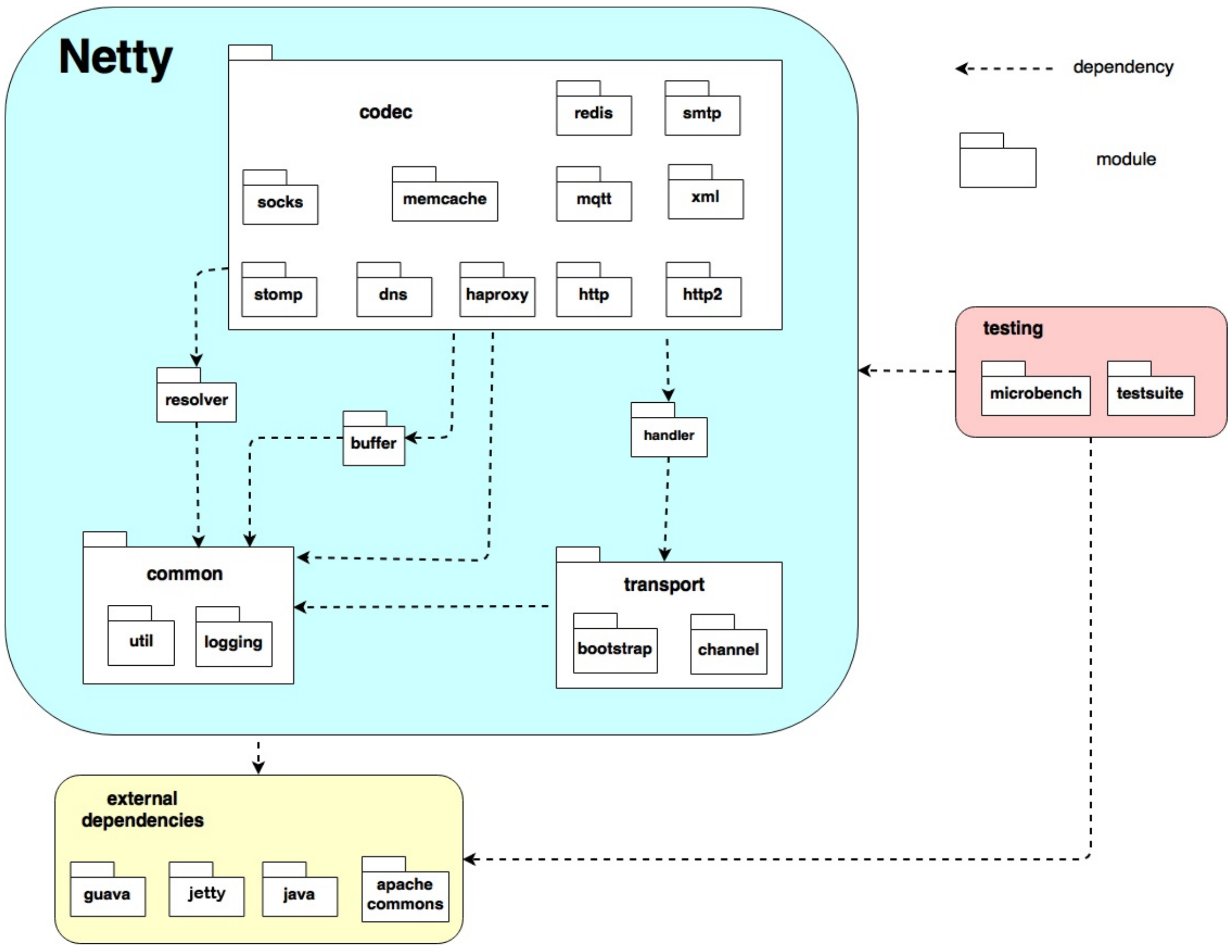


Figure 3. Netty's Modules Structure Model

As a framework, the source code of Netty could be organized as model structure, as shown on the figure above. In figure 3, the ecosystem of Netty is divided into three major parts: internal modules, testing system, and external dependencies.

Internal Modules

The followings are the explanation of each module inside internal modules with its description and dependencies:

- `netty-common`
  This module contains utility classes and logging facade. There are several sub-modules in Netty Common, such as `io.netty.util` and `io.netty.util.internal.logging`. This module does not have any dependencies to other modules in Netty. However, it has a dependency to `java.util` package.
- `netty-resolver`
  This module deals with resolving an arbitrary string that represents the name of an endpoint into an address and resolving a domain name asynchronously, which supports the queries of an arbitrary DNS record type as well. This module has a dependency to sub-module `io.netty.util`.

- `netty-buffer`
  Module `netty-buffer` handles fundamental data structure to represent a low-level binary and text message. This module has a dependency to `io.netty.util` and `java.*` package.
- `netty-transport`
  This module deals with channel API and core transports, native socket transport for Linux using JNI, Rxtx transport, SCTP transport, and UDT transport. It has several sub-modules such as `io.netty.channel` and `io.netty.bootstrap`. This package has dependencies to `io.netty.util`, `io.netty.buffer`, and `io.netty.resolver`.
- `netty-handler`
  Module `netty-handler` relates to flow control handler, flush control, ipfilter (filter IP address), logging (Logs the I/O events for debugging purpose.), SSL (Secure Socket Layer), stream, timeout and traffic. It has a dependency to `io.netty.channel`.
- `netty-codec`
  Module `netty-codec` handles codec-related functionalities, for example, dealing with packet fragmentation and reassembly issue found in a stream-based transport.

Testing System

Testing system also contains multiple modules, which will be explained below:

- `netty-microbench`
  This module deals with performing a series of micro-benchmark tests. It is built on top of OpenJDK JMH. This module has dependencies of all above modules.
- `netty-testsuite`
  This module contains packages for integration tests and common test suite. This module also has dependencies of all above modules.

External Dependencies

Besides having dependencies among its modules, Netty also has a dependency to external entities, such as Guava, JBoss, and Apache Commons.

4.2 Common Design Model

Any commonality across element implementations inside Netty is explained in this subsection.

Common Processing

By analysing Netty repository, it was acknowledged that there are many common processes that happen between its components inside its system. To deal with this issue, Netty has a module called `netty-common`. The practice of isolating common processing into separate code modules helps the development of Netty since the usage itself is spread across all modules, and the development of common processes can be done independently of other modules.

Below are some identified common processing elements inside this module:

1. Message logging
   There are three logging classes that Netty recommends to use, e.g., `Log4J2Logger`, `Log4JLogger`, and `Slf4JLogger`. In each class, there are five levels of logging; `trace`, `debug`, `info`, `warn`, and `error`.
2. Resource leak detector
   Resource leak detector helps Netty deals with any resource leak. There are four levels on how resource leak detector works; `disabled`, `simple`, `advanced`, and `paranoid`.
3. Net util
   This element holds a number of network-related constants which commonly used on many Netty's components.
4. Constant pool
   Constant pool helps Netty's elements to store any constant which will be used during its execution.

Design Patterns

Another important thing discussed in this section is Netty's design patterns as they are solutions to general problems that software developers faced during software development [20]. Presenting design pattern implemented in Netty will help contributors to understand how object-oriented structure resides in Netty source code. Below are listed several design patterns used by Netty:

1. Reactor Pattern: Reactor pattern is implemented by Netty as the framework is related to servers request handling with single-threaded event loop [19].

2. Intercepting Filter Pattern: Package `ChannelPipeline` implements an advanced form of the Intercepting Filter pattern to give a user full control over how an event is handled and how the handlers in the pipeline interact with each other [6].

4.3 Codeline Models

This section will explain how the directory of Netty project is managed, with a goal to ensure the build, test, and release processes could lead to a reliable Netty software.

Source Code Structure

The Netty project directories are organized based on their functionalities. Every folder mostly consists of Java classes and `pom.xml` file. Almost all the source code folders contain the main codes and the testing codes. There is a file called `package-info.java` which is used to explain the package where the file `package-info.java` stored. The module organization of Netty ensures every main code has a testing file located in the same directory. Figure 4 shows the source code organization of Netty 4.1 directory (Artifact ID).

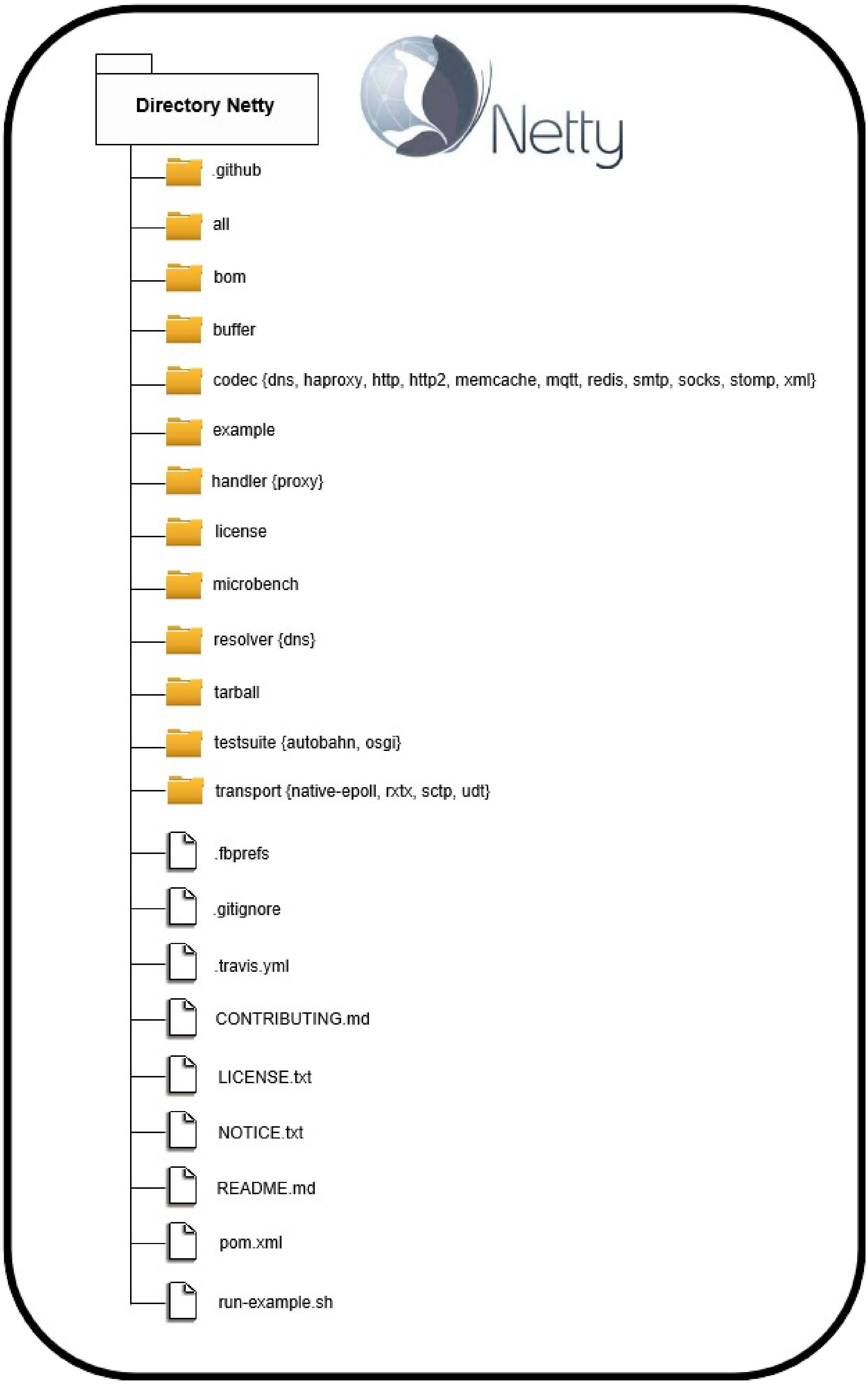

Directory Netty
Netty
.github
all
bom
buffer
codec {dns, haproxy, http, http2, memcache, mqtt, redis, smtp, socks, stomp, xml}
example
handler {proxy}
license
microbench
resolver {dns}
tarball
testsuite {autobahn, osgi}
transport {native-epoll, rxtx, sctp, udt}
.fbprefs
.gitignore
.travis.yml
CONTRIBUTING.md
LICENSE.txt
NOTICE.txt
README.md
pom.xml
run-example.sh

Figure 4. Netty's Code Structure Organizations

The Netty source code file always starts with copyright and license. After the declaration of Java package, usually, there is a short comment about the code.

Built and Testing Management

By looking at Netty repository, Netty's developers implement several ways of testing to verify that it works as it supposed to be. For example, after a developer finishes creating a class or modifies the source code, he/she will need to do a unit testing, usually by using added tools, such as JUnit and Mockito.

Regarding the build management, the developers of Netty also set up the build standards. For example, after the contributors have successfully set up the development environment, pushed a commit and submitted a pull request, a continuous integration platform called Travis CI is used. Continuous integration platform is vital to Netty for ensuring Netty's source code quality and to implement quick and easier error detection, since each introduced change is typically small. Another example is when developing Netty in a local repository, a developer can also utilize Maven to check whether his code is built successfully or not.

Release Management

In Netty Project, if the core developers intend to release a new version of Netty, they need to follow the standard Maven release procedure, which uses maven-release-plugin :

1. Stage the new release into the staging repository.
2. Verify the staged files are all good. If not, drop the staging repository and try again.
3. Close the staging repository so that no more modifications are made into the staging repository.
4. Release the staging repository so that the new release is synchronized into the Maven central repository.

Configuration Management

To configure Netty, a developer should follow the Developer Guide. First, a developer should use 64-bit operation system. Then, he/she should install the necessary build tools (e.g. Oracle JDK 8 or above, Apache Maven 3.1.1 or above, and Git). Last, a developer has to use Java programming language and also utilize IntelliJ IDEA since it is the officially supported IDE. A developer can also use other development environments as long as the contributor adheres to the Netty's coding style.

5 Deployment View

Deployment Viewpoint describes the environment into which the system will be deployed and the dependencies of the system [1]. Netty is supported by any operating system that can run Java 5+ (Netty 3.x) or Java 6+ (Netty 4.x). Furthermore, Netty does not require any specific hardware requirement to deploy.

As for the third-party software dependencies, the base functionality of each sub-module of Netty only requires JDK 5+ (Netty 3.x) or JDK 6+ (Netty 4.x) to run. For the development of Netty, it requires JDK 7+. However, some sub-modules require additional dependencies, for example, doing Transport Security (TLS) with Netty would require either OpenSSL or JDK (Jetty ALPN/NPN).

The binary files of Netty can be downloaded in Netty's Download Page as jar files. The jar files are available for all modules (`netty-all-x.x.x.Final.jar`) or for each sub-module (for example `netty-transport-x.x.x.Final.jar`). Netty does not differentiate the binary files for different operating systems or different instruction set architecture.

6 Functional View

In this section, the functional view graph of Netty will be presented and the explanation of the elements will be discussed briefly. Figure 5 displays the functional graph of Netty. Explanation on each component is available below the figure.

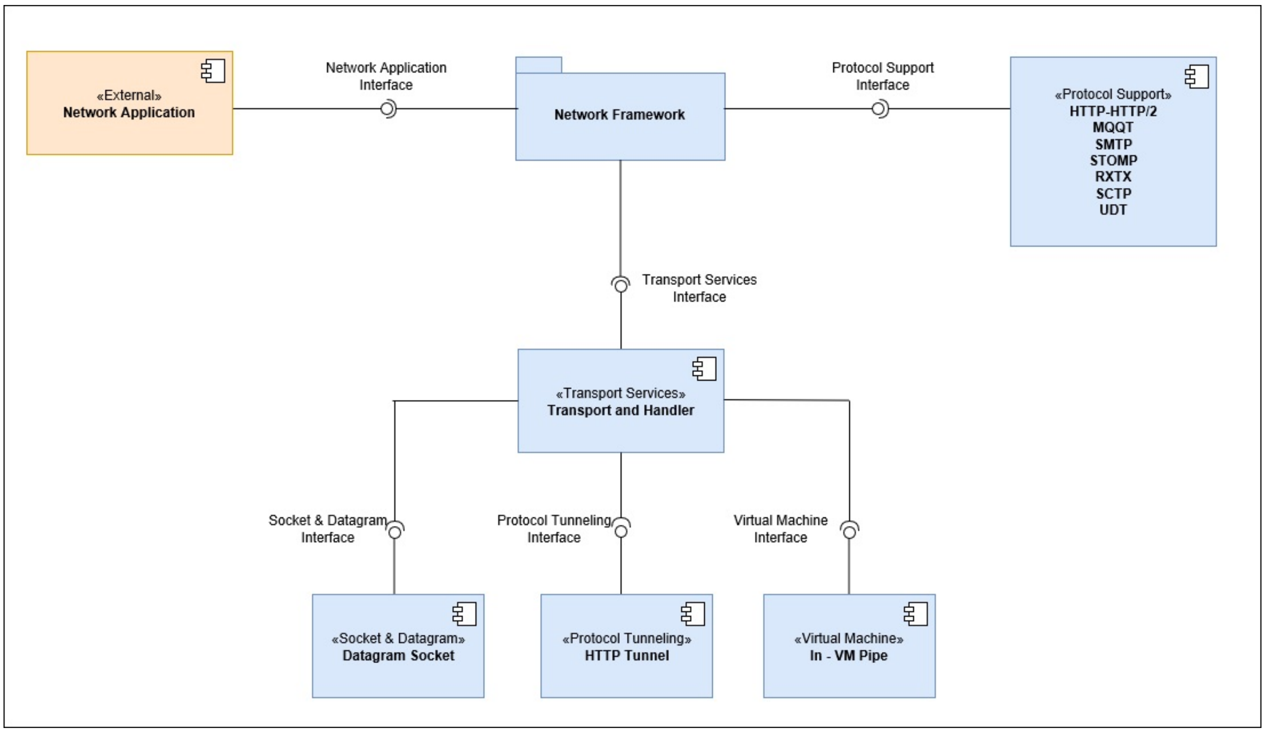


Figure 5. Netty's Functional View Diagram

6.1 Network Application

A network application is an application that is running on the different host. The communication between host needs to be handled by software. In this case, Netty plays a crucial role to ensure the communication effectiveness, security, and maintainability between that hosts.

6.2 Network framework

Netty as a network framework has a purpose to built and to ensure all the aspects go well for communication between clients and server. Netty was proposed as a unified API for various transport types, such as blocking and non-blocking socket, and designed based on flexible and extensible event model [2]. Netty has interfaces with the application, the supported protocols, and the transport services.

6.3 Protocol Supports

Netty supports many protocols, such as UDT (User Datagram Protocol), SCTP (Stream Control Transmission Protocol), HTTP/HTTP2 (Hypertext Transfer Protocol).

6.4 Transport Services

Netty's transport services can be divided into three parts, datagram socket, transport tunnel, and virtual machine.

Datagram Socket

The datagram socket is a package that has responsibilities for sending and receiving datagram packets [6].

Transport Tunnel

In Netty, HTTP tunneling is used as a component to connect a restrictive HTTP Proxy [6]. HTTP Tunnel consists of two components, client-side and server-side. The client-side will act as a SOCKS proxy, while the server-side translate and forward the HTTP request.

Virtual Machine

Netty offers a functionality to replace Java Virtual Machine's garbage collector. Instead of waiting for the garbage collector to work, Netty has the capability to flush the memory directly after reading or writing to a socket [6].

7 Technical Debt

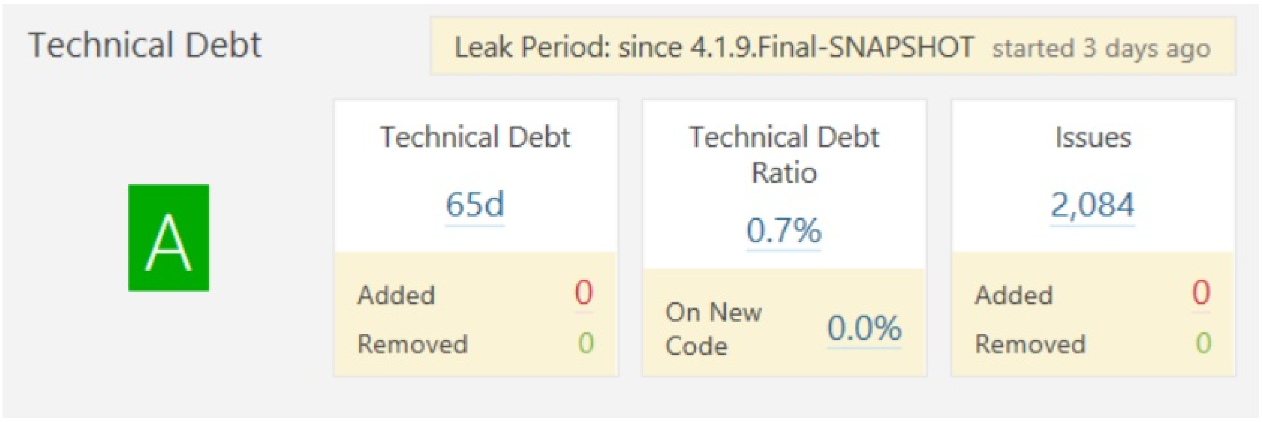
Technical debt refers to any extra development work that arises when applying an easy to implement code over the best overall solution [1]. In this section, an analysis of technical debt existence inside Netty repository is presented.

7.1 Identifying Technical Debt

There are two methodologies chosen during technical debt analysis; static code analysis and manual code analysis. Details on each methodology are explained further in the following subsections.

Static Code Analysis

The chosen static code analysis tool for identifying technical debt is SonarQube [8]. SonarQube works by analyzing the folder that contains the source code and shows the overall technical debt data. The following figure displays the details of technical debt given by SonarQube. In figure 6, Netty version 4.1 has technical debts amount of 65d (65 days). The parameter that is used in SonarQube, 'day', is based on SQALE (Software Quality Assesment based on Lifecycle Expectations) Methodology [24].



Technical Debt

Leak Period: since 4.1.9.Final-SNAPSHOT started 3 days ago

A

Technical Debt	Technical Debt Ratio	Issues
65d	0.7%	2,084
Added 0 / Removed 0	On New Code 0.0%	Added 0 / Removed 0

Figure 6. Technical debt of Netty 4.1 generated using SonarQube

The following figure shows the technical debt in size of circle radius in term of number lines of code (x-axis) and number of issues (y-axis).

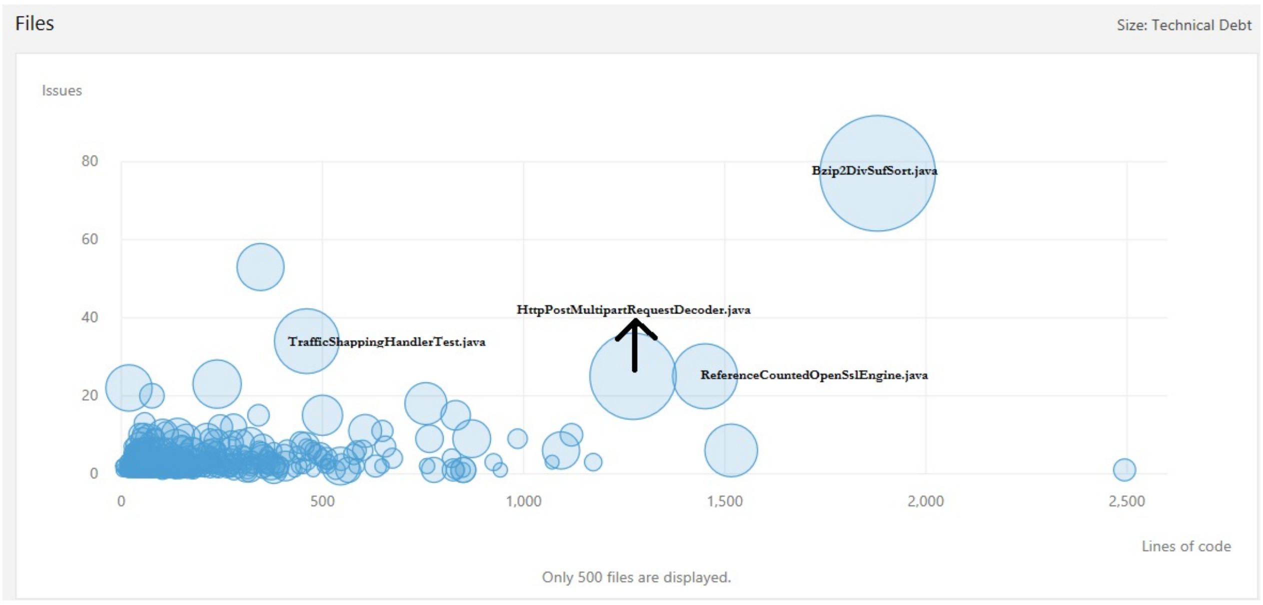


Figure 7. Technical debt in terms of lines of code (x-axis) and issues (y-axis) of Netty 4.1 generated using SonarQube

In Netty version 4.1 the biggest technical debt is located in `Bzip2DivSufSort.java` . -The other files that have the potential problems are `TrafficShapingHandlerTest.java` and `ReferenceCountedOpenSslEngine.java` .

Code Smells

By utilizing static code analysis tool, Netty's code smells can also be detected. According to [11], there are twenty-one types of code smells. From those twenty-one types, five different smells were identified inside Netty repository. A further explanation of those identified code smells is provided below.

Long Method

Class `Bzip2DivSufSort.java` is an example of a class inside Netty that has this characteristic. This class has a large method, i.e., `ssMultiKeyIntroSort` method. The cyclomatic complexity of this method is 48 which is greater than 25 (the authorized complexity recommended). Cyclomatic complexity itself is a quantitative measure of the number of linearly independent paths through a program's source code [12].

Large Class

`Bzip2DivSufSort.java` is an example of a large class inside Netty since it contains 2116 lines of code. A large class is not recommended because it exaggerates its complexity, which can lead to a poor performance and difficult to understand and to maintain.

Long Parameter List

`ssMergeBackward` method is an example of a method that has long parameter list. This method requires seven parameters, i.e., `final int pa` , `int[] buf` , `final int bufoffset` , `final int first` , `final int middle` , `final int last` , and `final int dept` .

Switch Statements

In object-oriented programming, a situation where switch statements or type codes are needed should be handled by creating subclasses [11]. Unfortunately, inside Netty source code, there are 678 matches of switch statements.

Duplicate Code

Based on the SonarQube analysis result, there are 14,425 lines of duplicated code. The file that has the most duplication code is `HttpPostMultipartRequestDecoder.java` that has 610 duplicated lines.

Manual Code Analysis

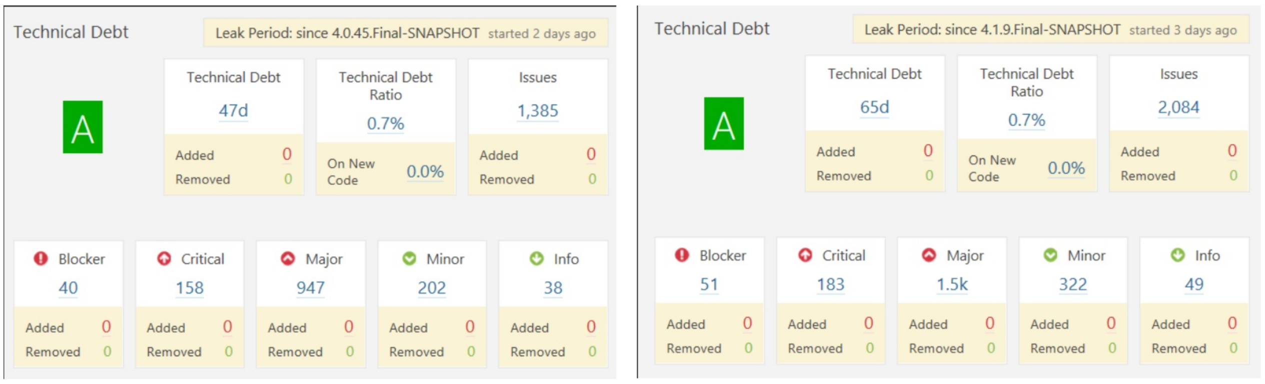
Besides static code analysis, another way to find out any technical debts inside a system is by doing manual code analysis. This methodology requires a lot of time and energy, but if done correctly, it will present a better understanding on how such a system deals with its technical debt and the quality of its code. There are several manual code analysis that can be done in the code, one of which is SOLID principle analysis [10] One of the modules in Netty, `netty-common` , was analyzed. In the analysis, there are no violations of SOLID principles that can be found. Each class in `netty-common` has its own responsibility, the module is open to extension but closed to modification, and base classes can be substituted using their derived classes.

Evolution of Technical Debt

SonarQube, a static code analysis tool mentioned in the previous subsection, was utilized again to see how the technical debts evolved in Netty.

Technical Debt Evolution Between Different Major Versions

For this analysis, two latest Netty major versions were analyzed, version 4.0 and version 4.1 [19]. Figure 8 displays the details of technical debt given by SonarQube. This figure shows that even though the Netty version is increased, the technical debt is also raised. Therefore, a newer version doesn't always lead to a better technical debts management.



Technical Debt — Leak Period: since 4.0.45.Final-SNAPSHOT started 2 days ago

A

Technical Debt	Technical Debt Ratio	Issues
47d	0.7%	1,385
Added 0 / Removed 0	On New Code 0.0%	Added 0 / Removed 0

Blocker	Critical	Major	Minor	Info
40	158	947	202	38
Added 0 / Removed 0	Added 0 / Removed 0	Added 0 / Removed 0	Added 0 / Removed 0	Added 0 / Removed 0

Technical Debt — Leak Period: since 4.1.9.Final-SNAPSHOT started 3 days ago

A

Technical Debt	Technical Debt Ratio	Issues
65d	0.7%	2,084
Added 0 / Removed 0	On New Code 0.0%	Added 0 / Removed 0

Blocker	Critical	Major	Minor	Info
51	183	1.5k	322	49
Added 0 / Removed 0	Added 0 / Removed 0	Added 0 / Removed 0	Added 0 / Removed 0	Added 0 / Removed 0

Figure 8. Technical debt of Netty 4.0 and Netty 4.1 generated using SonarQube

Figure 9 and 10 show the technical debt in size of circle radius in term of number lines of code (x-axis) and number of issues (y-axis). At this point, it can be seen from the mentioned figures that the biggest technical debt is changing. In Netty version 4.0 the most significant technical debt is located in file `HttpPostMultipartRequestDecoder.java`, but in Netty version 4.1 the biggest technical debt is located in `Bzip2DivSufSort.java`. The others files that have the potential problem both in version 4.0 and 4.1 are `TrafficShappingHandlerTest.java` and `ReferenceCountedOpenSslEngine.java.`

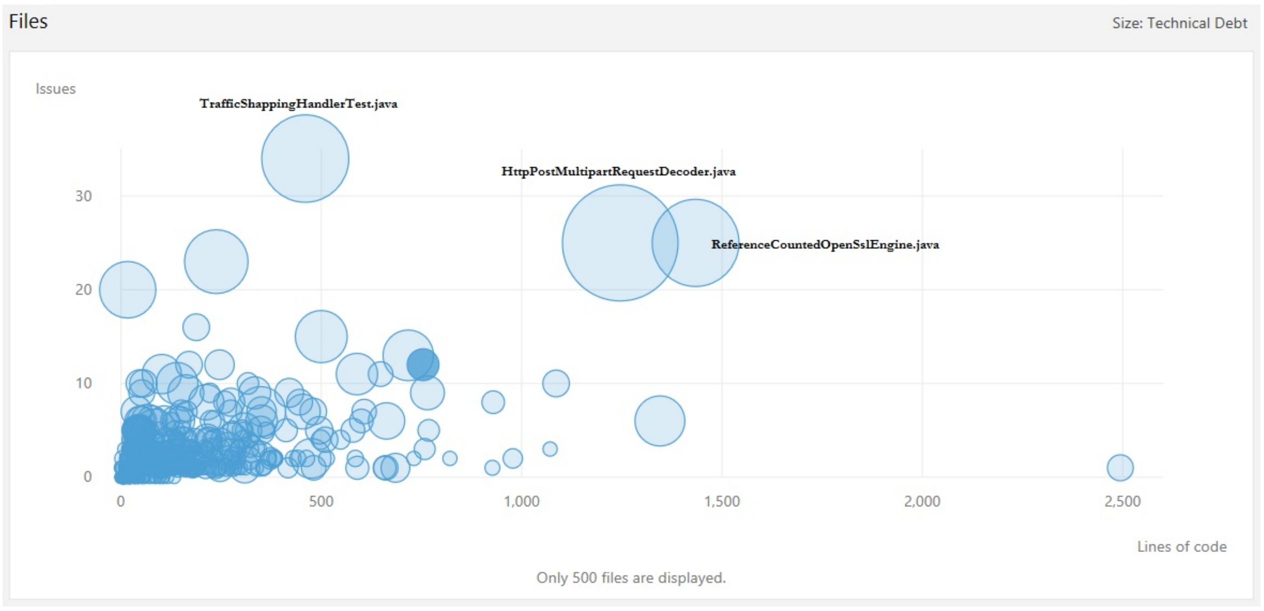


Figure 9. Technical debt in terms of lines of code and issues of Netty 4.0 generated using SonarQube

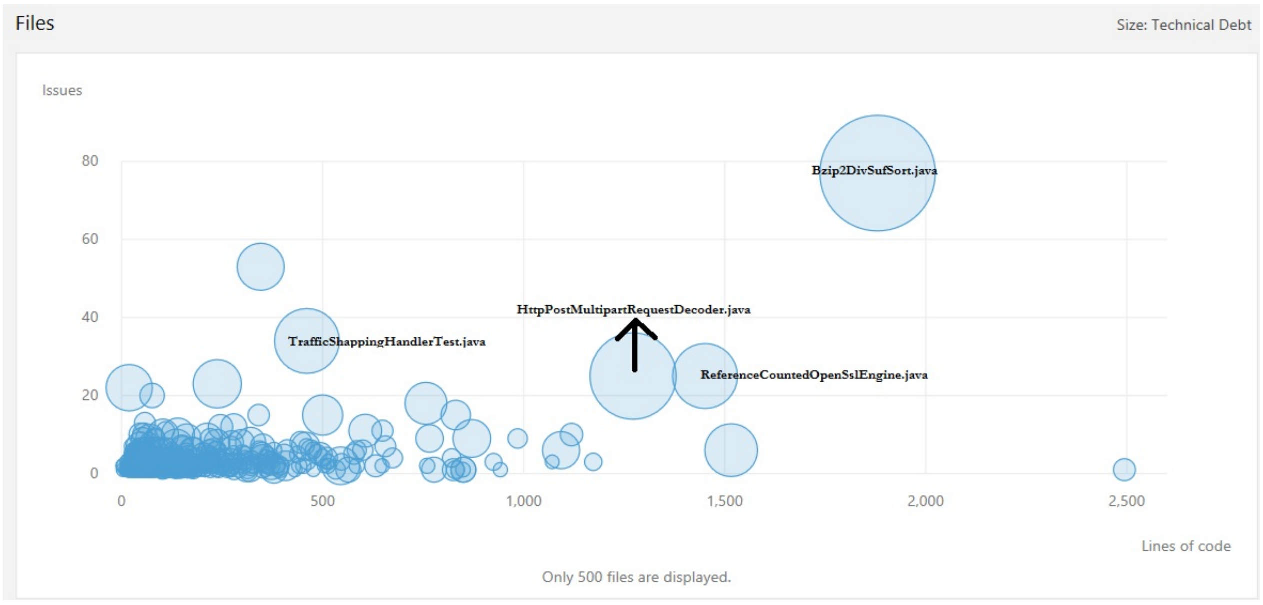


Figure 10. Technical debt in terms of lines of code and issues of Netty 4.1 generated using SonarQube

Technical Debt Evolution In Similar Major Version

Another interesting fact that was found is that Netty developers are always trying to reduce the number of technical debts inside Netty throughout its development time. This might seems untrue if the comparison is only done between different major versions, but if compared to similar major releases, this argument becomes valid. Figure 11 and figure 12 support this claim.

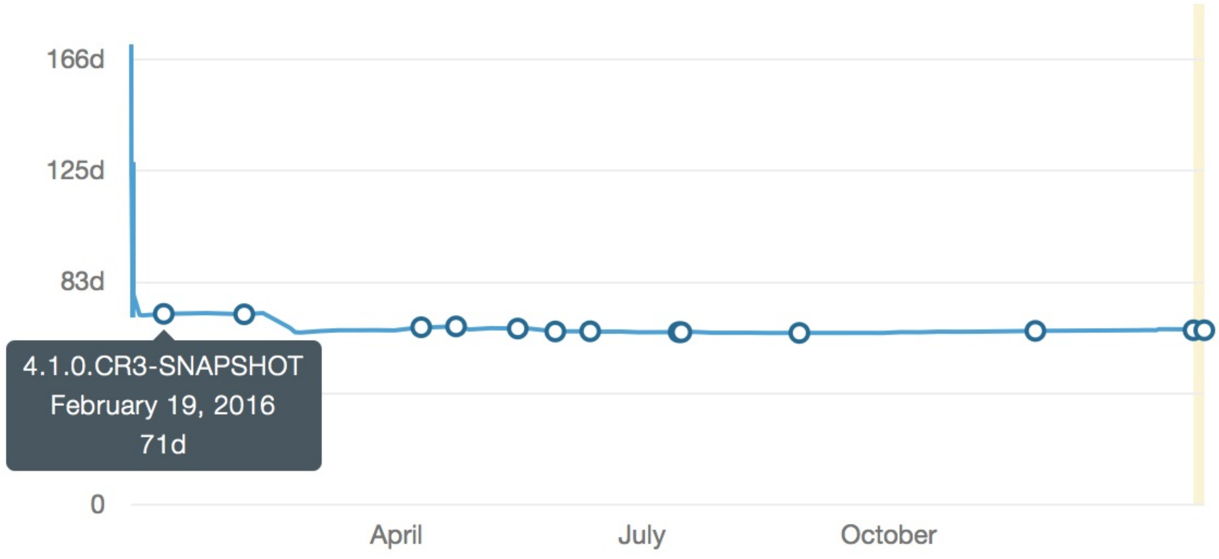


Figure 11. Technical debt of Netty 4.1 recorded at February 19, 2016 analyzed using SonarQube

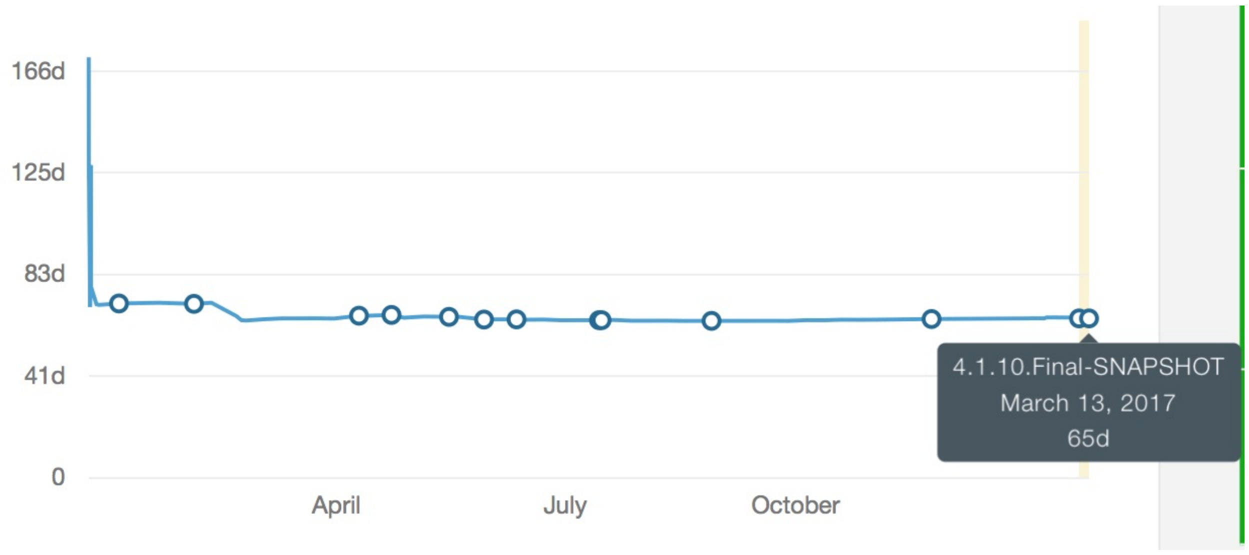


Figure 12. Technical debt of Netty 4.1 recorded at March 13, 2017 analyzed using SonarQube

From both figures, it can be seen that the number of total technical debts is decreased over the last single year. On February 19, 2016, it had 71 days of technical debt, while on March 13, 2016, it only 65 days. This is a good understanding of Netty developers' effort and also displays a nice progress since Netty only start utilizing SonarQube since the February 2016. This result looks even better if total lines of code is included in the analysis since there are 18453 of newly added codes during this period.

8 Testing Debt

Similar to technical debt, testing debt is also a poor aspect that may arise when developing software. In the following subsection, details related to testing debt inside Netty will be explained.

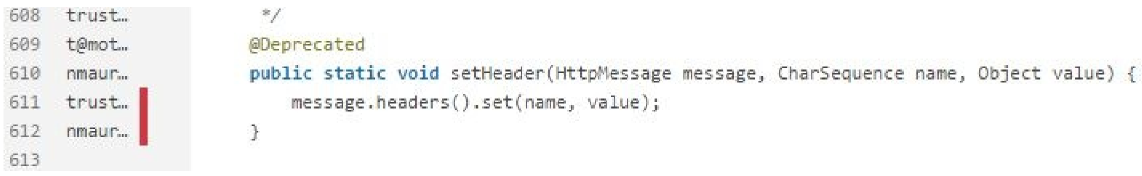
8.1 Code Coverage

Code coverage determines how many lines of code is being tested. Using SonarQube [8], it is found that Netty 4.1 has overall code coverage around 61.4 %. Though Netty code coverage is relatively low, it is not necessarily a good metric to know whether the system is well-tested. Even when it has a high number, it does not guarantee anything. So, it is necessary to look for sections of code where coverage is missing and analyze it whether the uncovered line is critical or does not matter if it is skipped.

- SonarQube show that several modules which have a relative big percentage of total line of code are standalone testing modules, such as Microbench and Testsuite. Thus, these modules will not be covered by unit testing.
- `Transport/RXTX` module also has 0% coverage. After checking at Netty source code, it

is found that this module does not contain a testing folder.

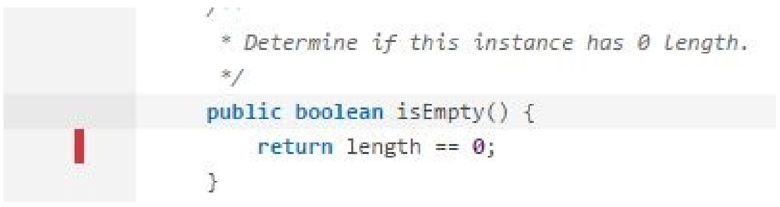
- Deprecated methods found in Netty source code, as shown in figure 13, are not tested as well, as usually the method will be removed later in next version or has a purpose to preserves the "backward compatibility".



```
608  trust…        */
609  t@mot…       @Deprecated
610  nmaur…       public static void setHeader(HttpMessage message, CharSequence name, Object value) {
611  trust…           message.headers().set(name, value);
612  nmaur…       }
613
```

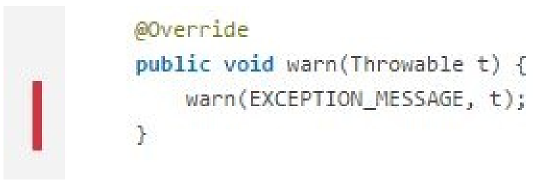
Figure 13. Deprecated method example

- Simple and logging methods found in Netty source code, as shown in figure 14 and 15, are not tested as they do not have a significant impact on Netty system or damage its functionalities if it has a bug.



```
/**
 * Determine if this instance has 0 length.
 */
public boolean isEmpty() {
    return length == 0;
}
```

Figure 14. Simple method example



```
@Override
public void warn(Throwable t) {
    warn(EXCEPTION_MESSAGE, t);
}
```

Figure 15. Logging method example

If some part line of code explained before are not covered, it is fine if unit testing does not include them. However, because code coverage of Netty 4.1 is still relatively small, there is still room for improvement as explained later in Testing improvement subsection.

8.2 Testing Procedures

Unit Testing

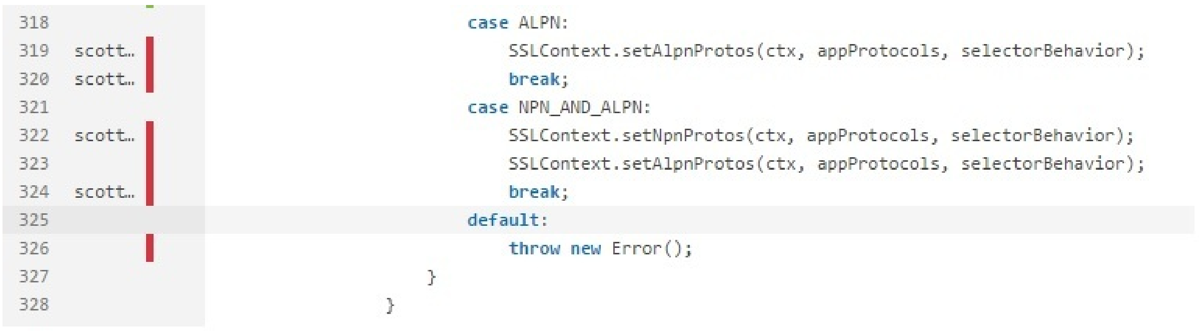
Unit testing is a testing technique test individual modules in the smallest possible chunks, isolated as much as possible from other code modules and runtime dependencies. Then it will be easier to identify, analyze and fix the defects. The concern with this method is functional correctness of the standalone modules.

Netty 4.1 is a Java-based project, so it utilizes JUnit as its unit testing framework. In Netty project, each module has its unit testing in its folder, for example, module `netty-common` has unit testing placed at `Netty/Common 4.1/src/test/java/io/netty/` separated from its main source code.

8.3 Testing Improvement

One obvious improvement that can be done in Netty project is increasing its code coverage. By manual checking on, it was found that some line of codes is safer or better to be tested, which are:

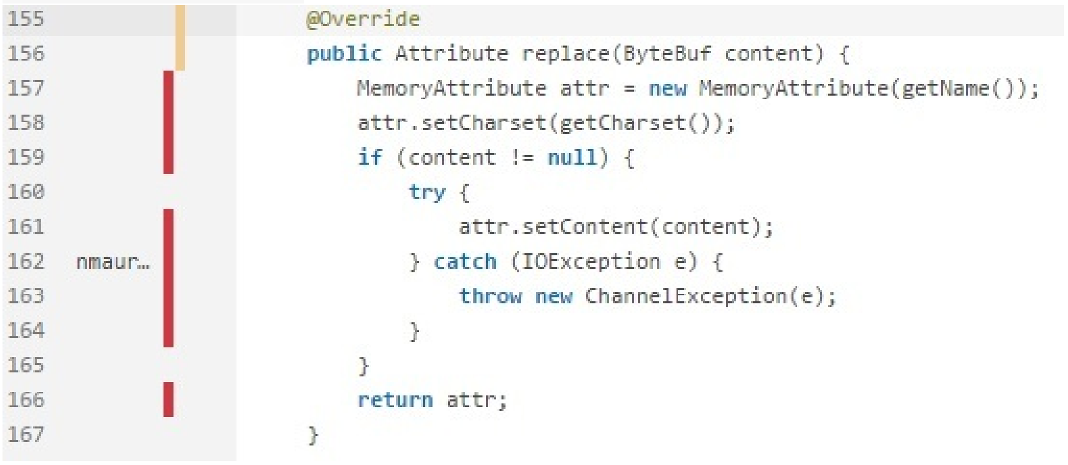
- Branching: The example for this kind uncovered line of codes is shown in figure 16. It can be seen that it is not a trivial code, so there should be a test case that covers all important branching line.



```
318                           case ALPN:
319  scott…                       SSLContext.setAlpnProtos(ctx, appProtocols, selectorBehavior);
320  scott…                       break;
321                           case NPN_AND_ALPN:
322  scott…                       SSLContext.setNpnProtos(ctx, appProtocols, selectorBehavior);
323                               SSLContext.setAlpnProtos(ctx, appProtocols, selectorBehavior);
324  scott…                       break;
325                           default:
326                               throw new Error();
327                       }
328                   }
```

Figure 16. Uncovered Branch

- Override Methods: This type of uncovered line of code experience the same thing as uncovered overloading methods. They are not covered because the test case only tests one kind Override method. So it would be better if there is a test case that covers all important Override methods. The example of this type is shown in figure 17, found in `MemoryAttribute.java` from HTTP module.



```
@Override
public Attribute replace(ByteBuf content) {
    MemoryAttribute attr = new MemoryAttribute(getName());
    attr.setCharset(getCharset());
    if (content != null) {
        try {
            attr.setContent(content);
        } catch (IOException e) {
            throw new ChannelException(e);
        }
    }
    return attr;
}
```

Figure 17. Overriding Methods

One of the factors that impede addition of tests is the documentation of Netty source code. There is no clear or detail explanation regarding the class or methods. For example, there should be a statement about input or output arguments (its type, its purpose, etc.). This would help developers or contributors to understand the code faster, especially if they want to test a method or create a class or a method that utilize a method from other class.

9 Evolution Perspective

Since the start of its development in 2001, Netty has been released in hundreds of version. In the age of more than fifteen years, Netty still keeps evolving and adding new capabilities into its system. Figure 18 displays the number of commits into Netty repository to help illustrates how evolution happens inside Netty.

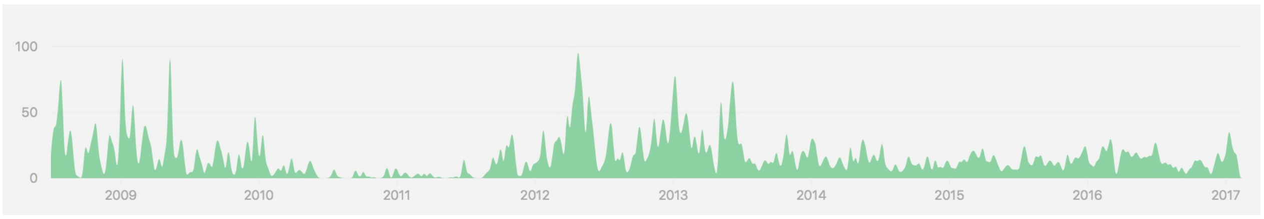


Figure 18. Netty number of commits

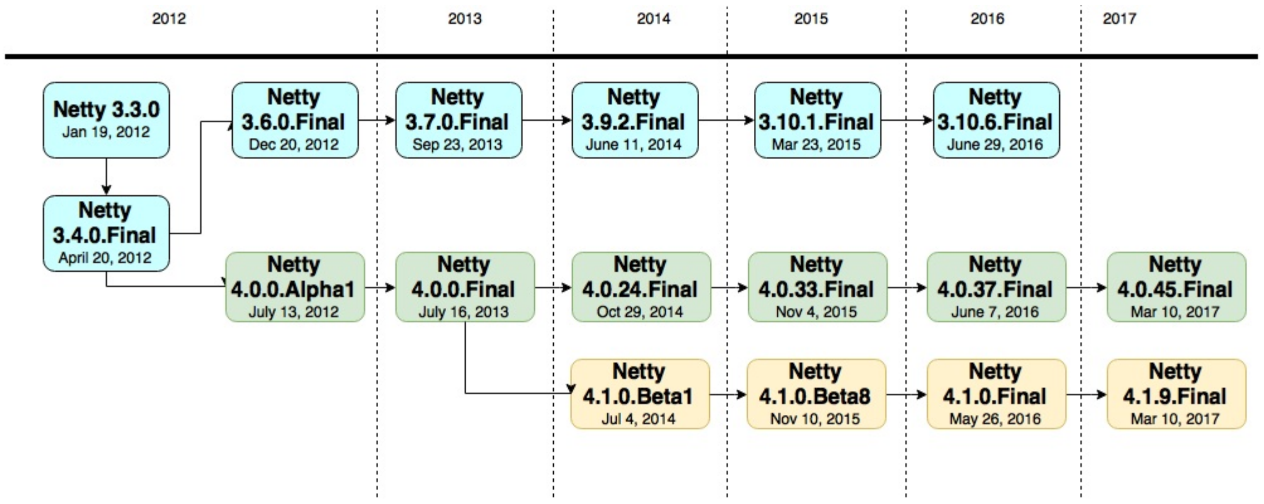

2012
2013
2014
2015
2016
2017
Netty 3.3.0 Jan 19, 2012
Netty 3.4.0.Final April 20, 2012
Netty 3.6.0.Final Dec 20, 2012
Netty 3.7.0.Final Sep 23, 2013
Netty 3.9.2.Final June 11, 2014
Netty 3.10.1.Final Mar 23, 2015
Netty 3.10.6.Final June 29, 2016
Netty 4.0.0.Alpha1 July 13, 2012
Netty 4.0.0.Final July 16, 2013
Netty 4.0.24.Final Oct 29, 2014
Netty 4.0.33.Final Nov 4, 2015
Netty 4.0.37.Final June 7, 2016
Netty 4.0.45.Final Mar 10, 2017
Netty 4.1.0.Beta1 Jul 4, 2014
Netty 4.1.0.Beta8 Nov 10, 2015
Netty 4.1.0.Final May 26, 2016
Netty 4.1.9.Final Mar 10, 2017

Figure 19. Netty version history

The progress of Netty version can be seen on figure 19, that shows how Netty evolved into different major version changes while still maintaining the older version. For example, after Netty released version 4.0.0.Final, they still maintain version 3.x.x until the next two years. Even after version 4.1.0.Final was released, the version 3.10 still got updated. During the development of a version, Netty usually releases several alpha, beta and/or CR (candidate release) versions first.

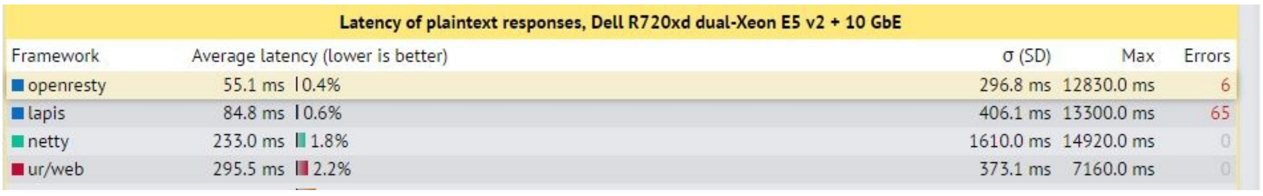
To see the reason why Netty keep upgrading to a newer version, one can look at the issues and pull requests in Netty repository [14]. The core developers will choose which issues can be added to a milestone of a version. When every item in the milestone is completed, the version related to that milestone will be released. The issues included in the milestone usually comes from bug fixing and requests of a new feature. Most of the requested features are related to security improvement and extra protocol support.

10 Performance and Scalability Perspectives

This section discusses the two quality properties of Netty, which are performance and scalability. Performance concern of Netty is how fast Netty system performs its workload. Meantime, scalability focuses on the predictability of the Netty system's performance as the workload increases, for example, the increased number of users [1]. As non-blocking IO and asynchronous client-server framework, Netty offers various of advantages compare its competitors, which are synchronous or blocking IO framework or other NIO framework, in terms of performance or scalability [16]. These perspectives will be elaborated below.

Performance

One of performance indicator is latency which is the amount of time required to satisfy a request. In the test [17], to measure the latency, the framework is tested by responding with the simplest of responses: a "Hello, World" message rendered as plain text. The test shows that Netty Project has lower latency, i.e. 233.0 ms, compared to one of Netty competitors or Project Grizzly, i.e. 9680.0 ms.



Latency of plaintext responses, Dell R720xd dual-Xeon E5 v2 + 10 GbE

Framework	Average latency (lower is better)		σ (SD)	Max	Errors
🟦 openresty	55.1 ms	0.4%	296.8 ms	12830.0 ms	6
🟦 lapis	84.8 ms	0.6%	406.1 ms	13300.0 ms	65
🟩 netty	233.0 ms	1.8%	1610.0 ms	14920.0 ms	0
🟥 ur/web	295.5 ms	2.2%	373.1 ms	7160.0 ms	0

Figure 20. Latency of Plaintext Response Test by TechEmpower [17]

Scalability

Scalability is the ability of a system to handle this increased workload [1]. The workload that effects the scalability of Netty is the number of concurrent connections. The scalability test [18] shows that Netty 4 could reach 1,200,000 transactions per second with more than 1000 connection.

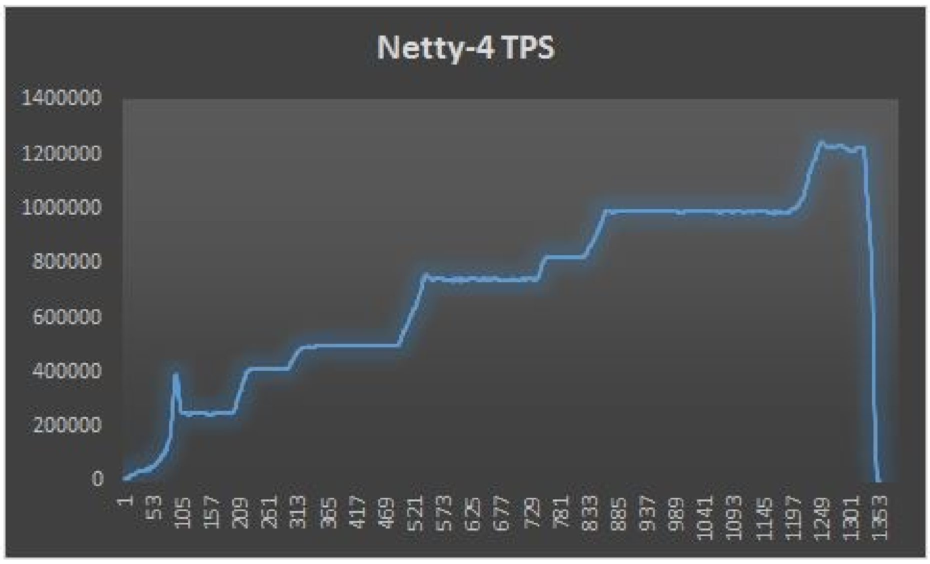


Figure 21. Result of Netty Scalability Test [18]

11 Conclusion

In this chapter, stakeholders, viewpoints, and perspectives of Netty are exploited in depth. Each section will lead to the understanding of Netty's software architecture. This section will summarize the chapter of Netty Project.

In stakeholders analysis, it is found that Trustin Lee is the founder of Netty and it is developed by a team of core developers and external contributors through GitHub. Besides personal developers, Netty is also used by well-known companies in their product, such as Apple, Facebook, and RedHat. Netty also has competitors as a client-server framework, e.g. Apache Mina and Grizzly. Then context view illustrates the relationship between external entities and Netty systems, for example, Java as a programming language, JUnit as testing frameworks, users, Apache License v2.0 as licenses in which Netty are distributed, IntelliJ as IDE and GitHub as version control and issue tracker.

Furthermore, the architecture of Netty Project is explained in depth by Development View, Deployment View, and Functional View. Development View section discusses all modules inside Netty, common processes, design constraints, design pattern, code structure, built and testing management of Netty Project. In Deployment View, the environment into which Netty is deployed is described, for example, the operating systems and third-party software dependencies. Netty's functional elements and their responsibilities are explained in Functional View.

After elaborating four viewpoints, technical and testing debt of Netty are explained. To identify technical debt, static code analysis and manual code analysis are performed. SonarQube is a tool to do static code analysis and it could detect that Netty has five types of code smells. It is also found that Netty Project has no SOLID violations during manual code analysis. Using SonarQube, it is also discovered that the code coverage of Netty is relatively low, i.e. 61.4 %. Thus, increasing Netty's code coverage is a must to improve the testing capabilities of Netty.

In addition, perspectives are included to deepen the analysis of Netty Project. Evolution perspective explains how Netty evolve since the development started in 2001. Meanwhile, the performance test [17] in performance and scalability perspectives section shows that Netty has lower latency, i.e. 233.0 ms, compared to its competitor, i.e. Project Grizzly. Furthermore, the scalability test [18] shows that Netty 4 could reach 1,200,000 transactions per second with more than 1000 connection.

After a profound and extensive analysis of Netty Project, it can be concluded that Netty is a reliable framework and maintainable project due to the involvement of many parties and proper methods of development. After reading this chapter, the reader is expected to understand the architectural complexity and potential shortage of Netty Project, then help them to contribute in Netty.

References

1. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. New Jersey: Addison-Wesley, 2014.
2. The Netty Project. Netty: Home. http://netty.io/index.html.
3. Georgios Gousios. How do project owners use pull requests on Github?. http://www.gousios.gr/blog/How-do-project-owners-use-pull-requests-on-Github.html, 2014.
4. The Netty Project. Writing a Commit Message. http://netty.io/wiki/writing-a-commit-message.html.
5. The Netty-Project. The Netty Project - Individual Contributor License Agreement v1.0. https://docs.google.com/forms/d/e/1FAIpQLSd7Bzje39G__THDJLRgKpQZ4gODNE26x_hZW3ofQOkgL6RGCA/viewform?formkey=dHBjc1YzdWhsZERUQnhlSklsbG1KT1E6MQ.
6. N. Maurer and M. A. Wolfthal. *Netty in Action*. 10th ed. Shelter Island: Manning, 2014.
7. Technopedia. What is Technical Debt. https://www.techopedia.com/definition/27913/technical-debt.
8. SonarQube. Features | SonarQube. https://www.sonarqube.org/features/.

9. StackOverflow. When using netty for a large file upload, how can I get uploaded size while upload is in progress?. http://stackoverflow.com/questions/29507687/when-using-netty-for-a-large-file-upload-how-can-i-get-uploaded-size-while-uplo, 2015.
10. Amir Khan. The Principles of OOD. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.
11. M. V. Mäntylä and C. Lassenius. "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study". Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431.
12. McCabe (December 1976). "A Complexity Measure". IEEE Transactions on Software Engineering: 308–320.
13. Trustin Lee. Trustin Lee Presentation in Twitter University. https://www.youtube.com/watch?v=0aoeSsKarc8, 2014.
14. The Netty Project. Netty project - an event-driven asynchronous network application framework. https://github.com/netty/netty.
15. Norman Maurer. Why Netty. http://normanmaurer.me/presentations/2014-netflix-netty/slides.html#1.0, 2014.
16. Norman Maurer. Network - Application Development The Easy Way. http://normanmaurer.me/presentations/2013-wjax-netty/#/10.
17. TechEmpower. Test types : Plaintext. http://www.techempower.com/benchmarks/#section=data-r9&hw=i7&test=plaintext
18. Ronen Nachmias. Netty 4 Throughput test. https://github.com/ronenhamias/netty-perf-testing/wiki/Netty-4-Throughput-test, 2014.
19. Alon Dolev. What is Netty ?. http://ayedo.github.io/netty/2013/06/19/what-is-netty.html, 2013.
20. TutorialsPoint. Design Pattern - Overview. https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
21. The Netty Project. Netty 4.1. https://garage.netty.io/sonarqube/overview?id=2477.
22. Daniel Bimschas. Zero-Copy Event-Driven Servers with Netty. https://www.slideshare.net/danbim/zerocopy-eventdriven-servers-with-netty, 2011.
23. Wang Wei. Importance of Logs and Log Management for IT Security. Retrieved March 29, 2017, from http://thehackernews.com/2013/10/importance-of-logs-and-log-management.html, 2013.

24. Jean-Louis Letouzey. The SQALE Method for Evaluating Technical Debt. In 3rd International Workshop on Managing Technical Debt. Zurich, Switzerland. 2012.

25. Alon. What is Netty?. http://ayedo.github.io/netty/2013/06/19/what-is-netty.html, 2013.

26. The Netty Project. Netty: Thank you for your donation. http://netty.io/sponsor/thanks.html.

Abstract

In this chapter, we will analyze and discuss the architecture of Node.js: an event-driven sever-side JavaScript environment. Several aspects of Node.js are analyzed, starting with the stakeholders. We give a context view to describe the different components involved and a development view to outline how Node.js is developed and the ways in which contributions can be made. We discuss the functional view, which provides the functionalities and evolution of Node.js. Lastly, we discuss Node.js from a scalability and performance perspective. We end with a short conclusion summarizing our most interesting findings regarding Node.js' architecture.

Table of Contents

Introduction

Node.js is an open-source tool for developing a wide array of server applications. Development started back in 2009, led by developer Ryan Dahl [1]. Initially Node.js only supported Mac OS X and Linux, but later in 2011 the project was expanded to also support Windows and other lesser known operating systems.

The reason why Node.js was originally started is because Ryan Dahl was fed up with the disconnect between the client and the web server. Each time the client wanted to be updated with new information it had to query the web server (for example, to keep track of progress of a file upload). This had been a long-standing problem in the field of web development, but most developers just decided to deal with it. Node.js was the first real attempt at solving this problem at the root by enabling real-time communication between the client and the server. It was immediately well received, as shown by the enthusiastic reaction of the audience during the original Node.js launch presentation [2].

Since its inception, Node.js has surpassed many of its early expectations [3]. The total amount of active contributors to the project itself increased each year to a record amount of about 480 contributors at the end of 2016. The number of downloads also continues to grow, with the total amount of downloads averaging at over 480,000 a day.

Because of the immense popularity of Node.js we wanted to study its architecture and how it is used to achieve some of the unique functionalities Node.js has to offer. We will first discuss the Stakeholders and the Context View to get an idea of who and what is involved in the development, usage and maintenance of Node.js. Afterwards we will analyze Node.js from a development viewpoint where we will talk more about the technical structure of Node.js and the design guidelines and patterns used by Node.js. Then we will highlight some of the most unique and distinctive functionalities that Node.js has to offer in the Functional View. Through this we hope to explain what made Node.js so unique and successful. Finally we will describe how the architecture of Node.js manages to serve such a wide userbase by considering the performance and scalability.

Stakeholder View

This section discusses various stakeholders involved in Node.js and provides a classification of the stakeholders as proposed by Rozanski and Woods [4]. Node.js is owned and governed by the Node.js Foundation which consists of the following stakeholders:

1. Board: Sets the business direction and oversees legal, financial and marketing domains.
2. Technical Steering Committee (TSC): Sets the technical direction and is responsible for technical governance of all Node.js projects including Node.js Core.
3. Foundation Members: They can be businesses or individuals. Both types of members have representation on the board.
4. Core Technical Committee: The Foundation sponsors the Node.js Core project and entrusts its governance to the Core Technical Committee (CTC).
5. Collaborators: The Collaborators along with the CTC maintain the nodejs/node GitHub repository. The collaborators are primarily involved in development, maintenance and testing of Node.js. Collaborators are organized into various Working Groups (WG) which have specific areas of responsibility such as testing, build, documentation and so on.

Figure 1 provides an overview of Node.js stakeholders.

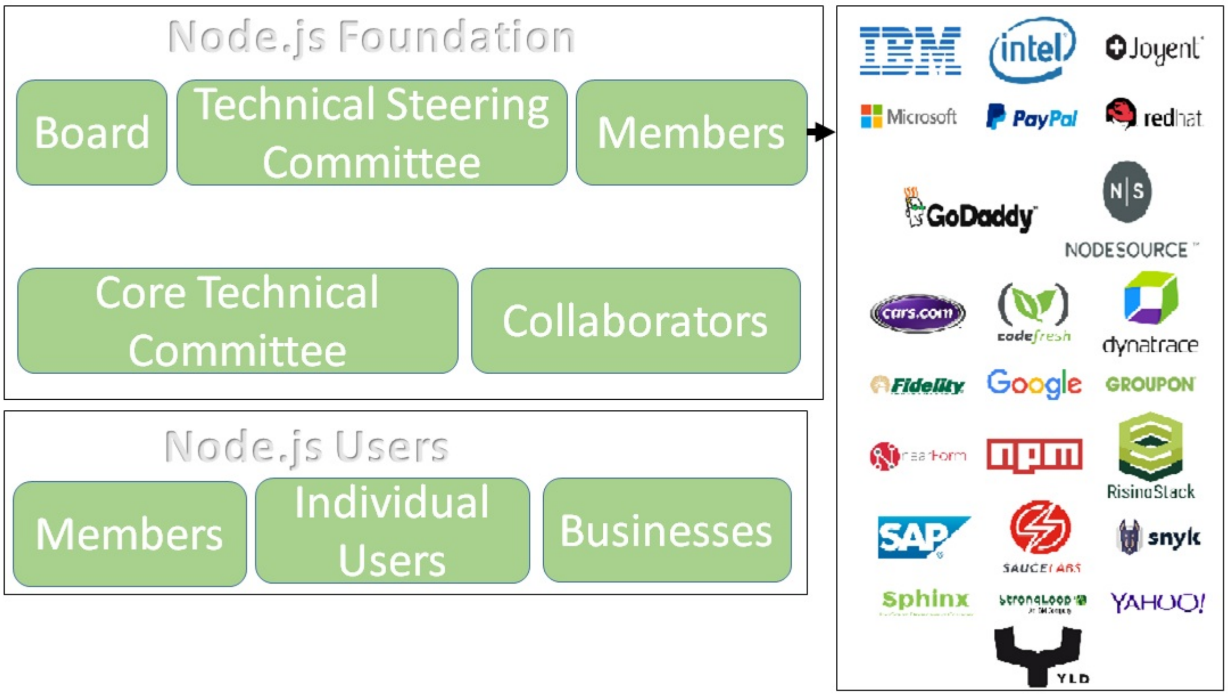


Figure 1: Stakeholder View

In order to provide a more precise and fine-grained view of the roles of the stakeholders, we provide a classification of stakeholders according to Rozanski and Woods [4].

Type	Stakeholders	Description
Developers	Core Technical Committee (CTC), Collaborators, Any developers on GitHub	CTC and Collaborators are actively involved in the development, maintenance and documentation of the project. They are also responsible for reviewing issues and merging pull requests from other Collaborators and GitHub users.
Acquirers	Node.js Foundation	The Foundation decides the business and technical direction of Node.js and is responsible for governance, marketing and sales.
Assessors	Testing Working Group (WG), Benchmarking WG	They test the quality and compliance of Node.js.
Communicators	Website WG, Documentation WG, Evangelism WG, Foundation’s Education Committee	The Website WG maintains the node.org website. Documentation WG is involved in documentation of APIs and the website. Evangelism WG promotes community events and manages social media content. The Education Committee helps users explore and learn Node.js.
Maintainers	Developers	Core Technical Committee makes decisions on the evolution of the project and the Collaborators are responsible for all maintenance tasks.
Support Staff	Collaborators	Collaborators provide support for the development of Node.js by helping users and novice contributors.
System Administrators	Collaborators	Collaborators also assume the role of system administrators.
Suppliers	Windows, Mac, Linux, SunOS and Docker	These stakeholders supply hardware and software that the system runs on.

We have analyzed some of the latest pull requests and issues to identify how the stakeholders collaborate to develop the system and make decisions. Node.js' governance policy suggests that collaborators are responsible for reviewing each other's pull requests and must include the CTC (by labelling it ctc-review) if there is disagreement among the collaborators regarding the proposed change. However, we observed in the last 25 pull requests that at least one member of the CTC was included as a reviewer even if the item was not labelled as ctc-review. This suggests that the CTC are the main integrators of the project. Their focus is to ensure that the commits are of high quality. Also, we observed that pull requests labelled "test" or "build" generally involve a member of the relevant working group.

Figure 2 provides the power grid view of Node.js stakeholders. It is used to classify the stakeholders based on their interest in the system and their power to influence the development of the system. Such a classification can help identify the stakeholders that have the highest influence on the system so that their interests may be prioritized.

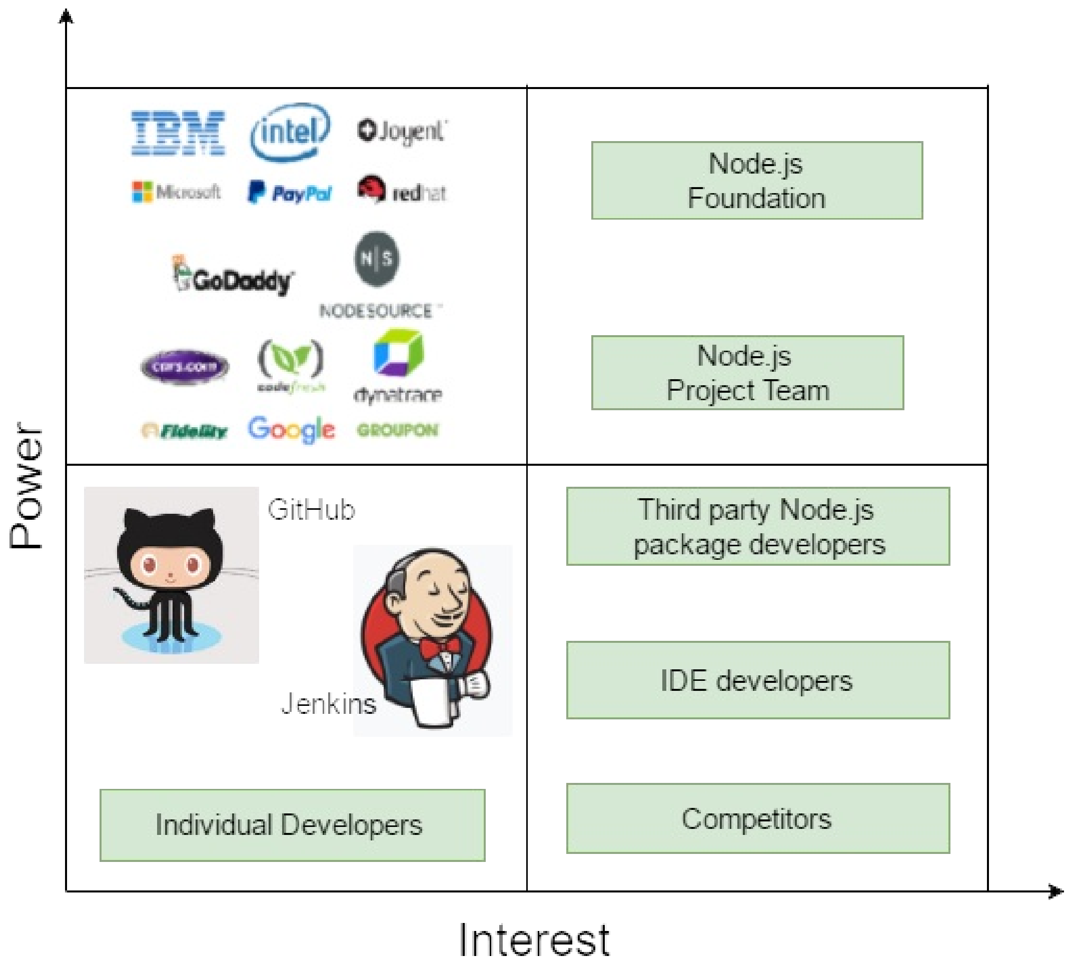


Figure 2: Stakeholder power grid view

High power high interest: Manage closely

These are stakeholders that are responsible for building and maintaining Node.js. It includes the Node.js Foundation and project team which are responsible for deciding the technical direction, development and maintenance of Node.js

Low power high interest: Keep informed

These are stakeholders who do not have much power in influencing the development of Node.js but have high interest in the system. It includes third-party developers that build Node.js packages and companies that build IDEs for Node.js. It also includes competitors of Node.js such as PHP, Golang, Reactor project and so on who have high interest in how Node.js evolves.

High power low interest: Keep Satisfied

This includes businesses who build applications using Node.js. Clients that are members of Node.js Foundation have the power to influence the evolution and technical direction of Node.js.

Low power low interest: Monitor

This category includes stakeholders that do not have any power or interest in Node.js but provide services that are used in the development of the system. This includes services like GitHub for source control, Jenkins for Continuous Integration. It also includes individual developers who build applications using Node.js.

Context View

In this section we will discuss all external entities that Node.js interacts with and the context in which they interact with each other. Through this analysis we aim to get an overview of the ecosystem in which Node.js resides through which we can identify dependencies and end users of interest.

System scope and responsibilities

According to their own website, Node.js was designed to build scalable network applications. Node.js is mainly meant to provide developers with the foundations for common server-side functionalities, for example [5]:

- Binary data manipulation
- File system I/O operations
- Database access
- Computer networking

Node.js is very lightweight and many higher-level functionalities are intentionally relegated to the many packages that are offered through its package ecosystem (called npm), which provides access to the world's largest collection of open source libraries and frameworks.

External entities

We have grouped all the external entities that are related to Node.js into several categories which we discuss sequentially below. An overview of all the entities and their relations to Node.js can be found in Figure 3.

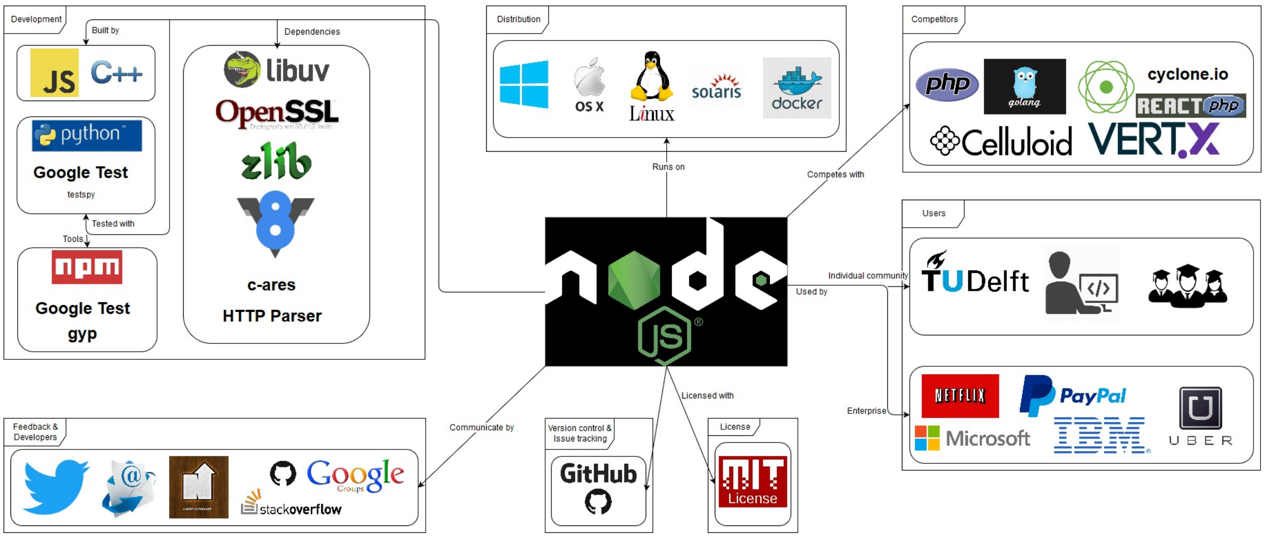


Figure 3: Context View

Development

These are different entities that are related to the actual development of Node.js, such as programming languages and testing.

Programming languages

Node.js is almost entirely written in JavaScript. It uses Google's V8 engine to execute all the JavaScript code, but since this engine is itself written in C++, some parts of Node.js's codebase that interact directly with this engine are also written in C++. Finally, Python is used to run many of the automated tests for Node.js.

Dependencies

There are a number of libraries or products that Node.js explicitly depends upon. Since Node.js was meant to be lightweight it offers only the most basic necessities for a product of its kind out of the box and thus, it does not have too many dependencies. Instead, it relies on the wide variety of additional plugins and libraries offered through npm, which can be used to extend the functionality of a Node.js application with many standardized solutions to common problems.

- libuv: Node.js is asynchronous and libuv provides a consistent interface for common asynchronous tasks across all supported platforms.
- c-ares: a library for asynchronous DNS requests.
- openssl: a library of cryptographic functions for security purposes.
- http parser: parses HTTP requests and responses.
- v8: the Javascript engine used by Node.js to run all of its JavaScript code.
- zlib: a library used for (de)compression.

Tools

In addition to the dependencies mentioned above, Node.js makes use of a couple of additional tools. These are not dependencies in the sense that Node.js cannot work without them, but can be thought of as additional features that enrich the Node.js experience.

- npm: the package manager of Node.js that offers access to a multitude of open source libraries.
- gyp: a build system to build those parts of Node.js and its dependencies that require compilation.
- gtest: a unit testing suite for C and C++ code.

Testing

As we previously mentioned while discussing the dependencies, gtest is used for C++ related tests. For the JavaScript code, the Python library pytest is used to configure and run the tests.

Distribution

Node.js is available on Windows, Mac, Linux, SunOS and Docker. It can be downloaded directly from their own website or through one of many third party package managers that offer it. Those packagers are responsible for packaging the Node.js code base themselves, so Node.js stresses that any issues people run into should be reported to them. If it turns out to be an issue with Node.js itself, those packagers will contact Node.js to notify them accordingly.

Competitors

Node.js is of course not the only platform that provides server-side functionalities. The following is a list of competitors that provide in some way the same functionalities that Node.js provides:

- PHP
- Golang
- Vert.x
- Reactor project
- Celluloid-io
- Reactphp
- Cyclone.io

Users

The users of Node.js can be divided into two subcategories. The individual community and enterprise.

Individual community

The individual community are the types of users that uses Node.js as hobby or for research. They do not intend to make money by using Node.js. Such users are developers and universities.

Enterprise

Enterprise are the users who do use Node.js as a tool in their company to help improve their product. Some major companies that use Node.js commercially are [6]:

- Netflix
- PayPal
- Uber
- IBM
- Microsoft

Feedback & Developers

For real-time discussion about Node.js development there is the #node.js IRC channel on the irc.freenode.net server. For general communication to all people working *with* Node.js and not just *on* Node.js, they also have a number of communication channels:

- The official Node Twitter account through which they keep their followers up to date.
- A weekly mailing list called Node Weekly, detailing the latest events within the Node.js community.
- NodeUp, a podcast that covers the latest Node.js-related news.

Feedback and help can be found on various platforms such as StackOverflow, GitHub and Google Groups, with all three platforms having an active community for Node.js.

Version control & Issue tracking

Node.js is actively being developed on GitHub using Git as its version control system. The same system is also used to track issues, report bugs and discuss features.

License

The Node license closely follows the MIT license.

Development View

The Development View details how the architecture supports the software development process and which development guidelines are to be taken into account by all developers. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system.

Module Organization

Node.js is both a product of its own as well as a service upon which other applications can be built. Because of this, it is useful to consider the design choices made for both in our analysis. Some of the choices made at a basic level in the Node.js architecture affect how applications using Node.js should be developed. Figure 4 shows a diagram depicting the high-level layered structure of Node.js as described in [7].

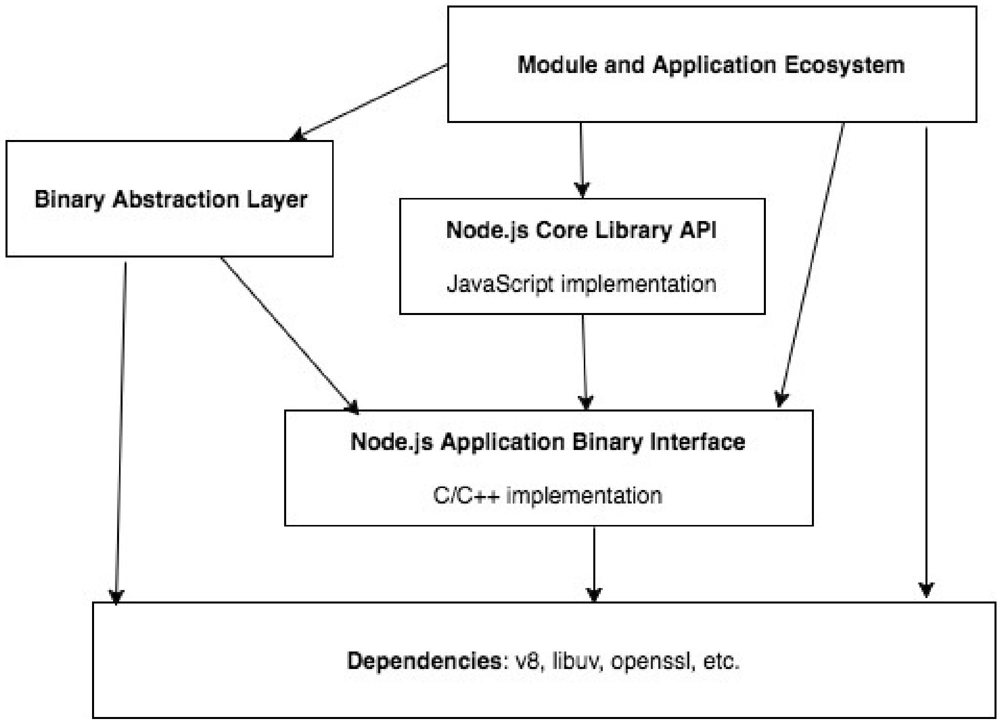

Module and Application Ecosystem
Binary Abstraction Layer
Node.js Core Library API
JavaScript implementation
Node.js Application Binary Interface
C/C++ implementation
Dependencies: v8, libuv, openssl, etc.

Figure 4: Module Organization

The Module and Application Ecosystem refers to the collection of all software that was built using Node.js. It is connected to all other layers in the diagram, which signifies that in theory developers of Node.js applications are free to connect to any of Node.js' layers. In practice most applications limit themselves to accessing only the Node.js Core Library.

This Core Library contains a variety of JavaScript files that simplify the development process for Node.js users. It offers a lot of common functionality out of the box, such as cryptography, network connections, event handling, etc. A part of the code in this library is marked as "internal", which hides a part of the API from the end user. The end user can still call these API functions if they wanted to, but since the format of these APIs can change without notice, they are marked as internal to discourage people from doing so.

The Application Binary Interface (commonly referred to as the ABI) is a relatively new part of Node.js [8]. The idea behind the ABI is to provide the end user with a stable API through which they can access the underlying JavaScript engine. At this point that engine is Google's V8, but the ABI allows Node.js to potentially switch to a different engine in the future. Also the ABI ensures that no new version of Node.js is required if changes are made to Google's V8 engine. The Binary Abstraction Layer serves a similar purpose, but it abstracts the ABI even further. On top of that, it also provides abstracted access to other dependencies aside from the JavaScript engine.

Design Patterns

This section discusses some of the design patterns used in Node.js.

Singleton Pattern

The singleton pattern limits the number of instances of a particular object to just one. Node.js uses module caching to implement the Singleton pattern and caches a module after the first time it is loaded. Every subsequent call to a module using `require(<module_name>)` returns the same instance of the cached module. In that way, these modules can thus be thought of as singletons.

Dependency Injection Container

Application modules built on Node.js typically use a backbone object that acts as a dependency injection container. Services such as logging and database access which are required throughout almost any application built on Node.js are attached to the backbone object and this object in turn can be used by the modules that require these services. In this way, modules have their dependencies injected from the outside through the use of the backbone object. The module is thus isolated from any changes in its dependencies.

Event-Driven Programming

In an event-driven program the flow of the application is the result of events that are fired or states that are changed. In general there is one single, global mechanism that listens for such events and whenever one is fired it will call the corresponding callback function. In Node.js this mechanism is called the Event Loop, which we will discuss in great detail in the Functional View and the Performance and Scalability Perspective.

Codeline Organization

In this section we will give a brief overview of the source code structure of Node.js, also called codeline organization. A well-defined codeline organization allows for automated builds, tests and releases. This has the potential of greatly simplifying the development process. The structure of Node.js' code is as follows:

Directory	Description
benchmark	This directory contains the code and data for benchmarking and measuring the performance of different Node.js implementations. The benchmarks are classified into 25 directories depending on the subsystem they benchmark. It also includes a miscellaneous directory for benchmarks that do not clearly fit in one of the predefined categories.
deps	This directory contains the source code of the third party components that Node.js depends on, some of which are shown in Figure 4.
doc	This directory contains all the documentation for Node.js, such as API explanations, changelogs, development guides, etc.
lib	This directory contains the JavaScript modules used in Node.js. The modules in lib/internal are meant for use in Node.js core and are not meant to be accessed from user modules.
src	This directory contains the bindings that expose the C/C++ libraries to JavaScript.
test	This directory consists of the code used to test Node.js. The `common.js` module in this folder contains a number of helper functions for commonly occurring tasks in tests.
tools	This directory contains additional tools that are useful for development with Node.js, like build functionality, automated testing libraries, etc.

Functional View

In this section we will describe the most important and unique functionalities of Node.js. We will also address the architectural elements and choices that make these functions possible. Node.js indirectly offers vast amounts of functionality through the myriad of applications that have been built upon it, such as chat bots, application monitoring and data streaming. However, since our focus is on Node.js itself we are mainly interested in its architecture and will therefore also focus solely on the functionalities offered directly to developers working with Node.js.

Threading

The threading mechanism of Node.js is quite different from other webservers. For example, webservers based on PHP and ASP.NET typically create a new thread for each client request. That client request thus causes the entire program to be reinstantiated on the thread for that specific request. So while the webserver is definitely multi-threaded, each program instance served to a client operates only on a single thread.

As we mentioned in the Development View, Node.js is built upon libuv, which allows it to perform asynchronous or non-blocking I/O operations. Because of this, Node.js is able to use only a single "calling thread" that serves all incoming client requests without causing independent requests to block each other. Any work that needs to be done is passed off to a thread pool where it is assigned to a separate thread on which it will be executed. After the work has been done, control is ceded back to the calling thread to provide the client with the appropriate response to their original request [9].

In a later section on the Performance & Stability Perspective we will discuss the advantages and disadvantages that these two approaches carry with them. For now it suffices to emphasize the differences from an architectural point of view. We will now elaborate on the calling thread, commonly referred to as the event loop.

Event Loop

As we previously discussed in the Development View, Node.js is based on the event-driven programming paradigm [10]. This becomes clear through the so-called event loop, a process that is constantly accepting incoming requests and providing responses to previously accepted requests. In between accepting a request and responding to it, the event loop will refer work that needs to be done to one of the background workers, as mentioned in the previous subsection on Threading. Node.js makes good use of JavaScript's support for callbacks by allowing a callback to be sent along with each such task. Figure 5 pictures the execution model as we have discussed it so far.

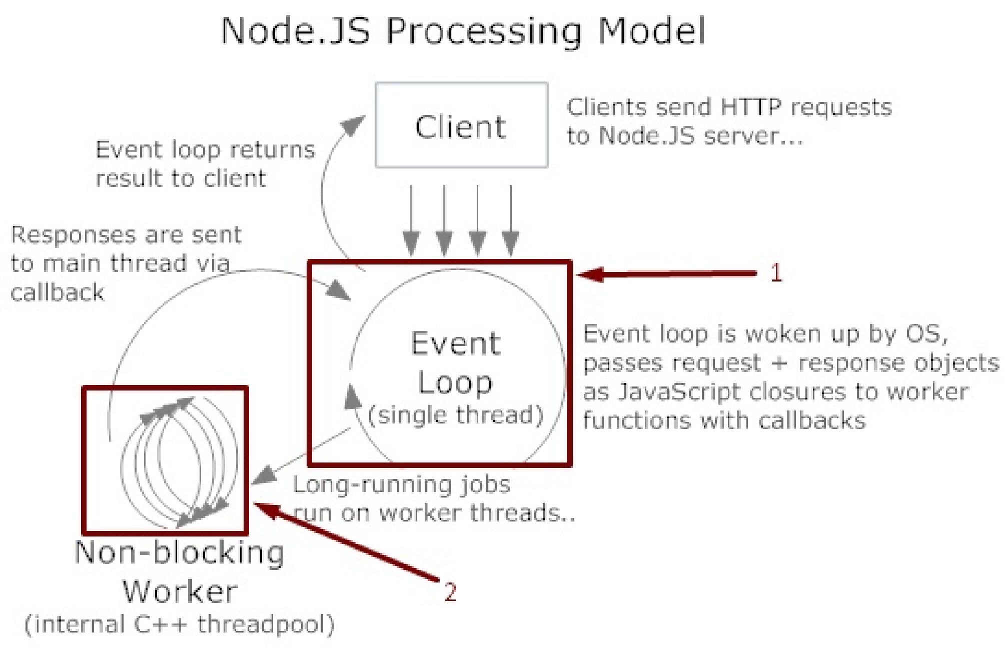

Node.JS Processing Model
Client
Clients send HTTP requests to Node.JS server...
Event loop returns result to client
Responses are sent to main thread via callback
1
Event Loop (single thread)
Event loop is woken up by OS, passes request + response objects as JavaScript closures to worker functions with callbacks
Long-running jobs run on worker threads..
Non-blocking Worker (internal C++ threadpool)
2

Figure 5: NodeJS Execution Model (from http://www.codingeek.com/tutorials/nodejs/is-nodejs-single-threaded/)

Each of these callbacks is registered to an Event Queue, where it waits to be called as soon as the corresponding task has been finished. These callbacks are all executed on the main thread again, as they are responsible for providing the client with a response. Therefore it is advisable to keep the callback functions as lightweight as possible, as they will cause delay for other simultaneous requests. As long as there are callbacks in this queue, the event loop will remain active to respond to all outstanding client requests. Figure 6 shows the interplay between the mechanisms behind the Event Queue, the event loop and the thread pool.

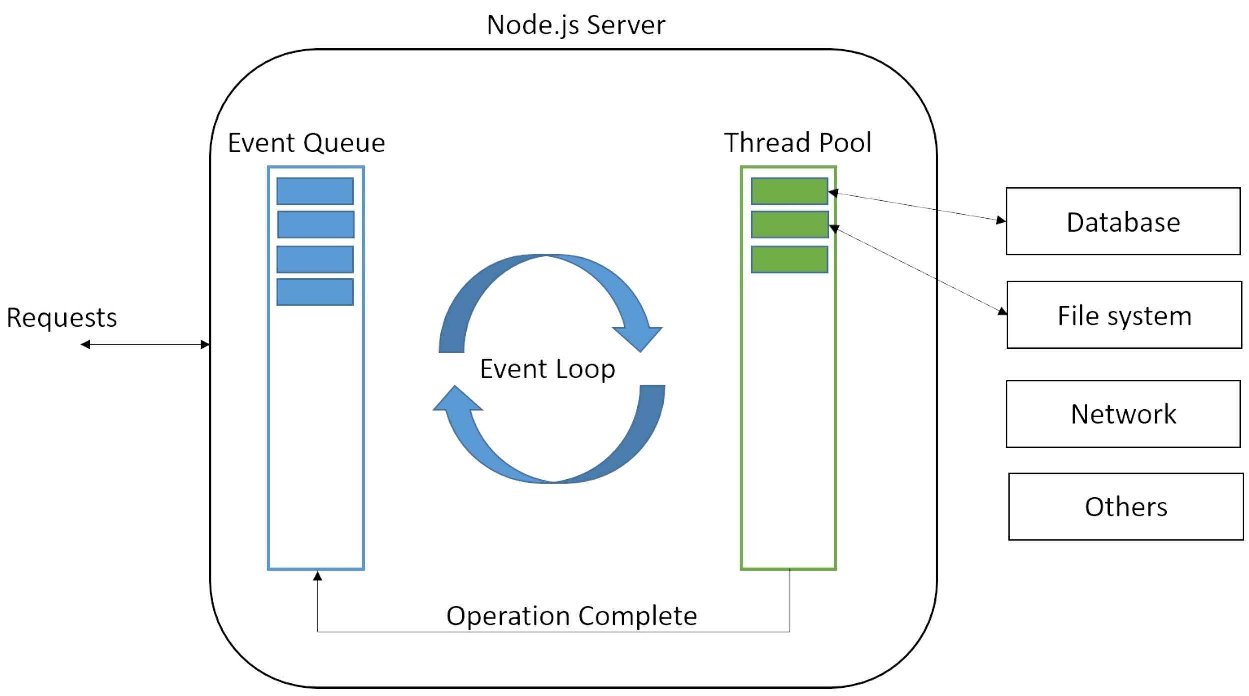


Figure 6: Single Threaded event loop model (from http://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop)

In Node.js every object that can fire events is an instance of the `EventEmitter` class. Each of these objects has an `on()` method in which a type of event can be specified along with the appropriate callback such that each time the named event is fired, the corresponding callback is called. If multiple callbacks have been assigned to the same event, all of them will be executed in a synchronous manner (according to the first in, first out principle). If necessary developers can override this procedure by using the `setImmediate()` method for a callback to switch to an asynchronous model.

Some of the core modules of Node.js that extend the `EventEmitter` class are the `Server`, `Socket`, `http` and `fs` (short for File System) modules. For all of these it is easy to imagine how the event-based way of programming enables the system as a whole to function in a non-blocking way. Without events, the program would have to postpone executing any of its subsequent code until it has received a response from a remote server or until a file has been read completely. By specifying callbacks for such events, the main program can continue being executed, only returning to the callback when new data has become available.

Package Management

Node.js uses a package manager in order for developers to add modules to their applications. These modules add new functionality to existing applications. This new functionality can help developers create their app or enhance their app for the users. Although most packages are modules, there are some packages that are not modules for they have no index.js or main field in the package.json file for use in Node.js programs[11]. This way the Node.js program cannot use the `require` function to load the package and is thus not a module.

npm

When installing node, the package manager called *npm* is automatically installed as well. npm is written in JavaScript and was developed by Isaac Z. Schlueter. He saw that module packaging was not done well in node compared to other platforms. This was the reason for him to come up with npm[12]. npm makes it easy for developers to share, reuse and update shared JavaScript code and uses nested dependencies as shown in Figure 7

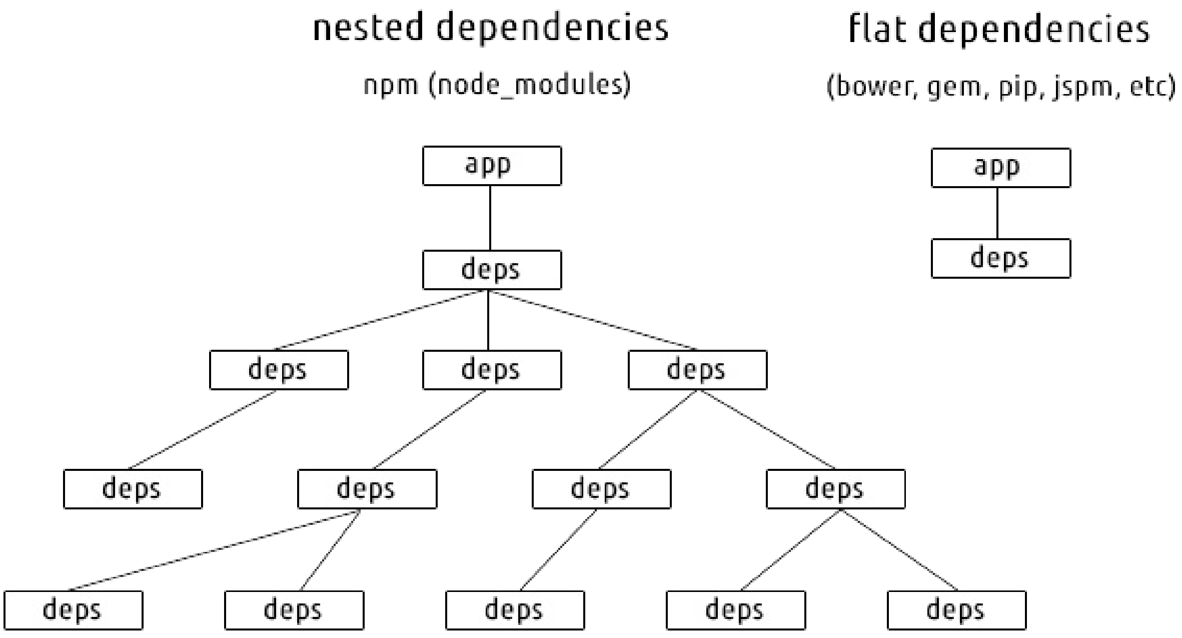


Figure 7: npm nested dependicies (from https://maxogden.com/nested-dependencies.html)

npm comes with a command line client that interacts with a remote registry. The CommonJS format is used for the packages on the registry along with a metadata file, package.json. There is no screening for the packages on the registry, so anyone can upload their package. Because of this, the quality of packages is very diverse. Some security risks are present because of this rule. Although the npm server admins can delete malicious packages,

deleting packages may cause failure of applications using those packages.
npm can also be used for managing applications locally. By defining a package.json file for a node application, dependencies can be automatically downloaded and updated by using npm. Even versions can be set for packages, if an application only works with a certain version, so that npm will not update that package and only installs that version of the package.

Other package managers

Next to npm there are other third-party package managers that can be used with node. Yarn for example is package manager that was released by Facebook. All these package managers use the npm public registry, but are different in the client-side experience.

Performance and Scalability Perspective

This section provides a detailed view on how the architecture of Node.js enables the development of highly scalable server-side web applications. In order to understand how Node.js achieves scalability, it is first necessary to understand the drawbacks of traditional web server architectures when handling a large number of concurrent requests that are I/O or network intensive. Ryan Dahl, the creator of Node.js provides an insight into the design limitations of traditional web server frameworks.

```
 "Turns out, a lot of the frameworks were designed in a way that they made the assumpt
ion a
  request – response is something that happens instantaneously and that your entire we
b
  development experience should be abstracted as a function. You get a request, you re
turn a
  response. That is the extent of your context."— Ryan Dahl
```

As we pointed out in the previous section, this type of request/response architecture uses multi-threading to provide concurrent handling of requests. However, since a new thread is created for each request and because of the use of blocking I/O calls, such an architecture cannot scale up efficiently. The sections below discuss how the event-driven architecture of Node.js differs from this traditional approach with respect to performance and scalability.

Multi-Threaded Request/Response Model with Blocking I/O

Figure 8 shows the request processing mechanism in web servers with multi-threaded synchronous I/O model. Here, a new thread is created for each incoming request at the server. The thread blocks when I/O operations are being executed. Though this type of thread-per-request model provides concurrent handling of requests, it is evident from Figure 8 that a large amount of memory and CPU is tied-up without use when the thread blocks while waiting for I/O or network calls to return. Also, as the number of concurrent requests increases, the overhead of thread management becomes high.

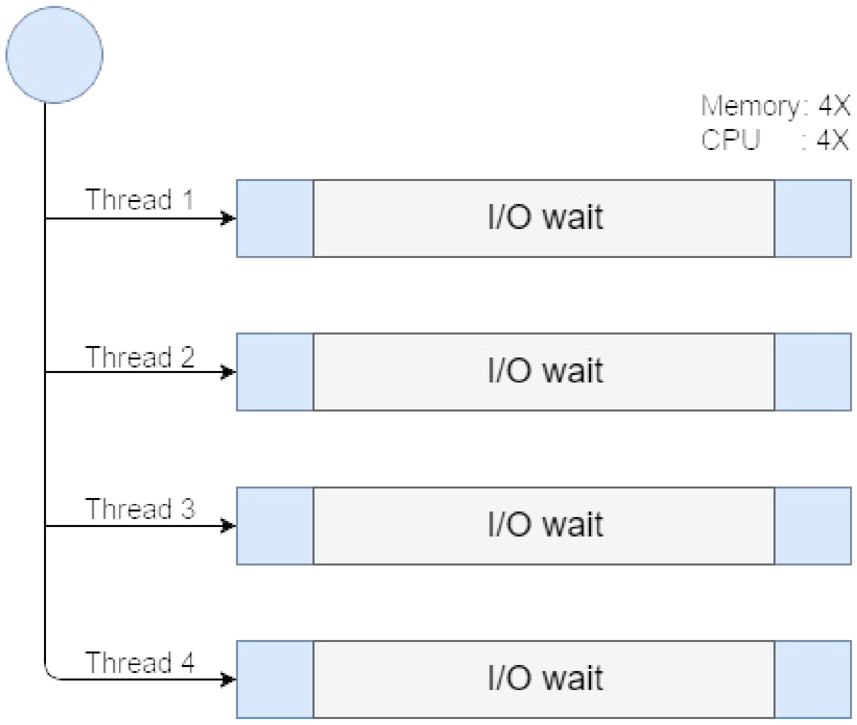


Figure 8: Synchronous I/O (from http://bijoor.me/2013/06/09/java-ee-threads-vs-node-js-which-is-better-for-concurrent-data-processing-operations/)

Single Threaded Asynchronous I/O Model

Figure 9 shows the request processing mechanism in web servers which run a single thread and perform non-blocking I/O calls. Node.js uses a similar concurrency model which makes it more scalable than the multi-threaded model. This model uses a single thread which services all the incoming requests at the web server. The I/O operations are executed as events and do not block the calling thread. It is clear from Figure 9 that this model utilizes the CPU more efficiently than the multi-threaded model. Also, since it uses a single thread, it is more memory efficient compared to the multi-threaded model.

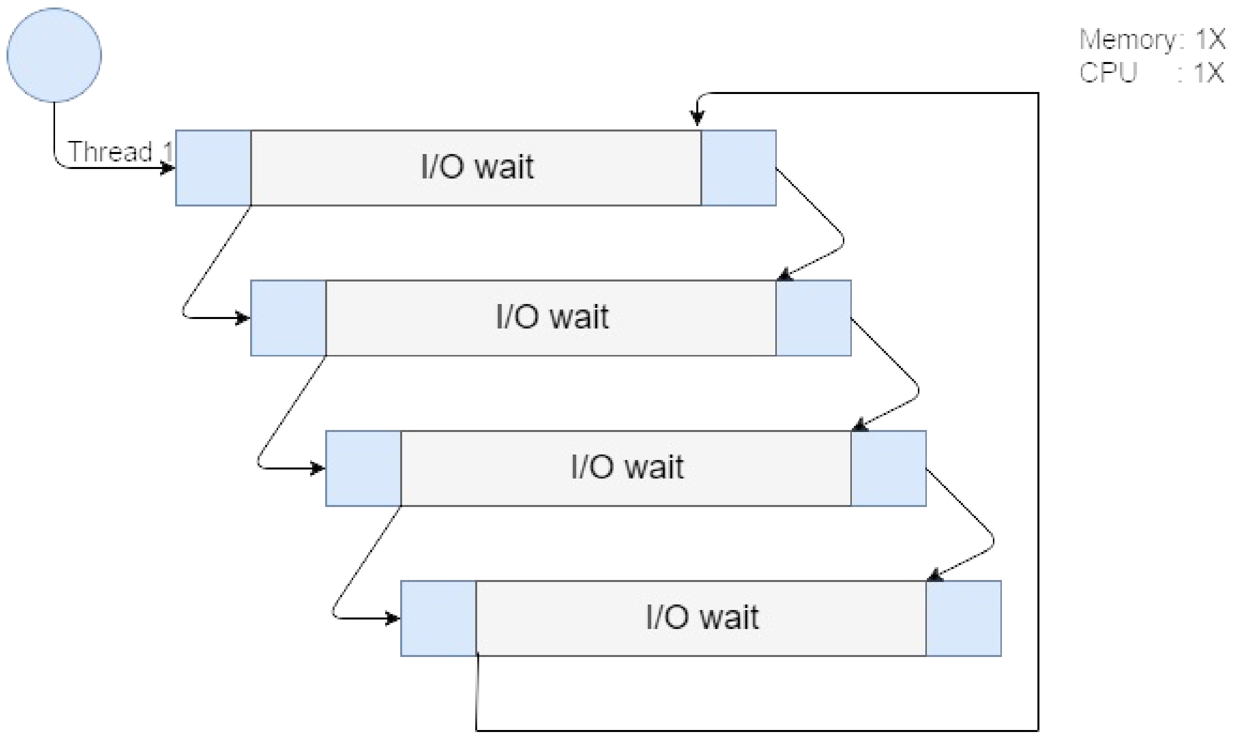


Figure 9: Asynchronous I/O (from http://bijoor.me/2013/06/09/java-ee-threads-vs-node-js-which-is-better-for-concurrent-data-processing-operations/)

At the backend however, threads are still required to execute the various I/O operations in parallel. But this complexity is hidden away from the Node.js application which makes programming on Node.js much easier. Also, since this model moves away from the thread-per-request architecture, it does not incur the overhead of thread management.

Despite its superior concurrency model, the Node.js architecture is not suitable to scale across multiple cores in a system. Since Node.js uses a single thread to service all incoming requests, it cannot leverage multiple cores in the system by distributing the load across cores. Listed below are two mechanisms to overcome this.

- Node.js provides the Cluster API which applications can use to distribute incoming connections across worker processes which run on multiple cores.
- The libuv library which manages the threads in Node.js by default creates 4 threads in the thread pool when the node process starts running. The `UV_THREADPOOL_SIZE` environment variable can be configured to create a maximum of 128 threads which are distributed across cores by the server operating system.

Conclusion

In this chapter we have analyzed the architecture of Node.js on different perspectives and views, so that the readers could have a broad idea of what Node.js is and what is it's structure.

As aspiring software architects, analyzing Node.js structure has been enlightening experience for us. It has provided useful insight as to how complex modular architectures are developed and managed, and the reasons for doing so. We appreciated the opportunity to analyze the various architectural aspects of Node.js which allowed us to become intimately familiar with a project we had all heard of but had never really figured out completely. From the stakeholders and the ecosystem in which Node.js resides to the more technical aspects dealing with the software development process, the functionalities and the scalability.

References

1. Dahl, R (2010-11-09). "Joyent and Node". Google Groups., https://groups.google.com/forum/#!topic/nodejs/lWo0MbHZ6Tc, Accessed on April 3rd, 2017
2. Video: Ryan Dahl: Original Node.js presentation (November 26th, 2009), https://www.youtube.com/watch?v=ztspvPYybIY, Accessed on April 3rd, 2017.
3. Node By Numbers (January 1st, 2017), https://nodesource.com/node-by-numbers, Accessed on April 3rd, 2017.
4. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
5. Li, A. (June 5th, 2016), Architecture of Node.js' Internal Codebase, *Yet Another Node.js Blog*, https://arenli.com/architecture-of-node-js-internal-codebase-57cd8376b71f, Accessed on February 25th, 2017
6. Delgado, A. (October 5th, 2017), Top 5 companies using NodeJS in production, https://www.linkedin.com/pulse/top-5-companies-using-nodejs-production-anthony-delgado, Accessed on February 26th, 2017
7. Node.js Foundation (2017), Development | Node.js, https://nodejs.org/en/get-involved/development/#stability-policy, Accessed on March 6th, 2017
8. Krill, P. (November 29th, 2017), Node.js update makes JavaScript VMs future-proof, http://www.infoworld.com/article/3145428/javascript/node-js-making-strides-in-javascript-vm-independence.html, Accessed on March 6th, 2017
9. Rahul, G. (October 27th, 2017), Is NodeJS single threaded - Let's find out, http://www.codingeek.com/tutorials/nodejs/is-nodejs-single-threaded/, Accessed on April 2nd, 2017
10. Norris, T. (January 5th, 2017), The Nodesource Blog, https://nodesource.com/blog/understanding-the-nodejs-event-loop/, Accessed on April 2nd, 2017
11. Packages and Modules, https://docs.npmjs.com/how-npm-works/packages, Accessed on April 2nd, 2017

12. npm (software), https://en.wikipedia.org/wiki/Npm_(software), Accessed on April 2nd, 2017

Processing - Learn How to Code Within the Context of Visual Arts

By Omar Hommos, Mohammed Al-Rahbi, David Bergvelt, and Mathew Vermeer.

Delft University of Technology.

Abstract

Processing is a simple software sketchbook and a language designed to introduce programming to new learners in the context of visual arts. This chapter includes a brief overview on several aspects of Processing, precisely its stakeholders, context, deployment, and evolution. It further presents an analysis of its architecture development and its technical debt. The presentation of these aspects is inspired by the methods followed in the Software Systems Architecture book by Rozanski and Woods [1].

Table of Contents

I. Introduction

Processing (nicknamed P5) is an open source computer programming language and integrated development environment (IDE) built for the electronic arts, new media art, and visual design communities with the purpose of teaching the fundamentals of computer programming in a visual context. Processing is also widely used by hardware hackers, as it is quite easy to use and supports important features like serial communications, with many related libraries and tutorials [2].

The Processing Project was started in Spring 2001 by Ben Fry and Casey Reas, who still contribute largely to the project [3]. in 2012, Dan Shiffman joined them to start the Processing foundation [4]. Since then, the project continues to grow a user community of thousands of people.

The Processing IDE is called a *sketchbook*. Processing language files are called *sketches*. Every sketch is a subclass of *PApplet*, a Java Class that implements most of the Processing language features. Thus, Processing language is more of a simpler version of Java. Processing code is translated into Java before compilation. Processing classes are all treated as inner-classes. As such, the use of static variables and methods in classes is prohibited unless Processing was set to run in pure Java mode. The IDE and the language create an ecosystem that is quite powerful for new learners [2].

This chapter begins with a discussion on the stakeholders, the context, the deployment, and the evolution of the Processing environment. Afterwards, it focuses on the architecture development view of the IDE, and its technical debt. It later ends with a conclusion, followed by an interview with a P5 user in the Appendix.

Our Favorite P5 Projects!

Petting Zoo is the latest work developed by experimental architecture and design studio Minimaforms. The project simulates life using a robotic environment. The whole project makes you question environments, life forms, and communication. The creatures can learn and explore by interaction with viewer. It's an example of cool hardware art projects!

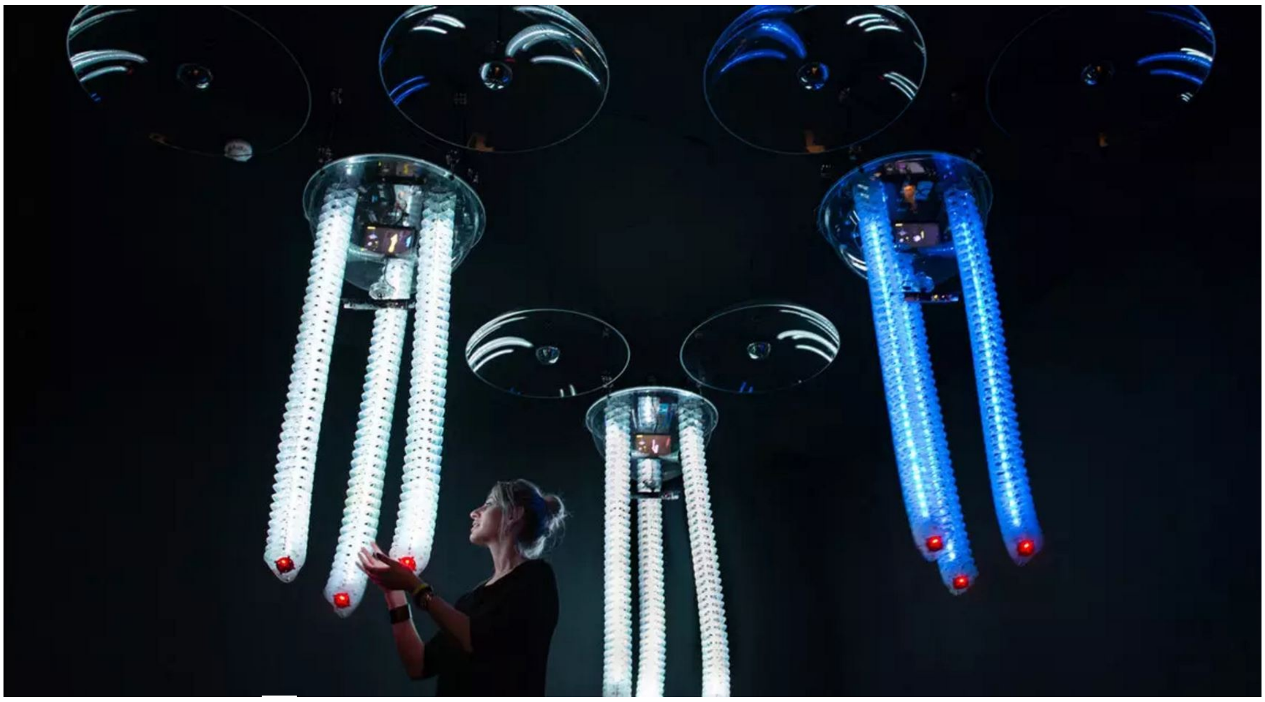


Figure 1: Project Petting Zoo

unnamed soundsculpture is a 3D motion piece visualizing sound by the movement of the body. A dancer was used to visualize a musical piece (Kreukeltape by Machinenfabriek) as closely as possible with movements of her body. She was recorded by three Kinects using Processing language the resulting views were aligned and merged to form a complete 3D point cloud and imported into 3D Studio Max.

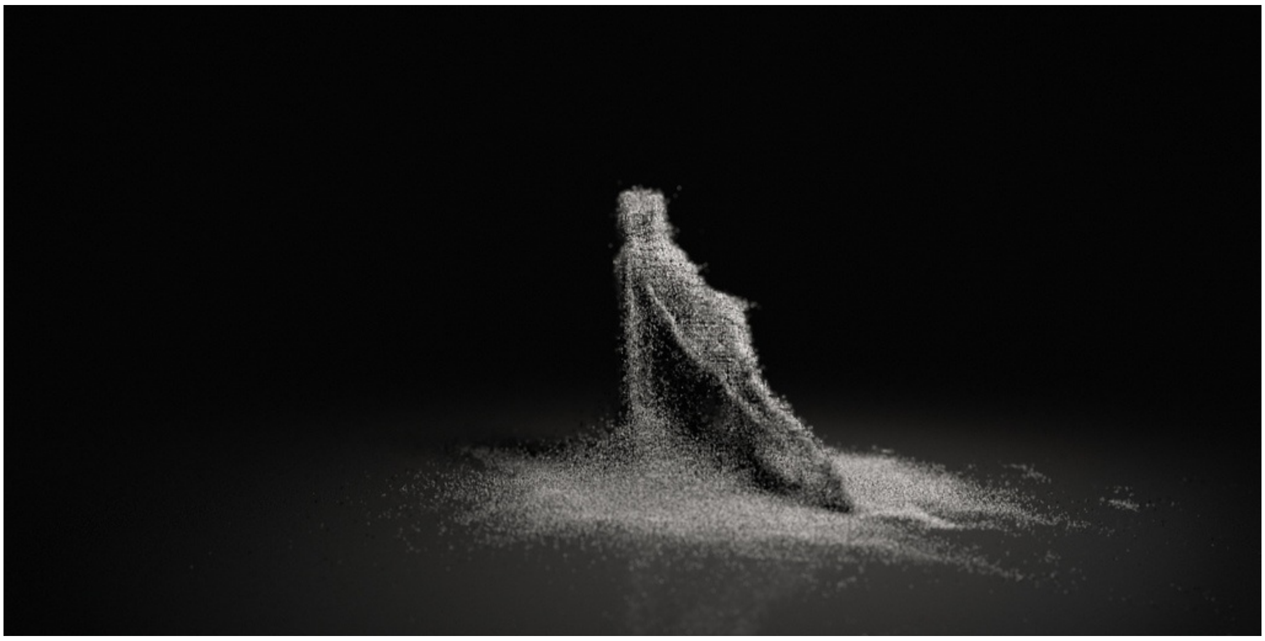


Figure 2: Project Unnamed Soundsculpture

Partitura Created by Abstract Birds and Quayola, Partitura is a custom software for creating real-time graphics aimed at visualising sound. The term “Partitura” (score) implies a connection with music, and this metaphor is the main focus of the project. The main characteristic of the system is its horizontal linear structure, like that of a musical score. It is along this linear environment that the different classes of abstract elements are created and evolve over time according to the sound.

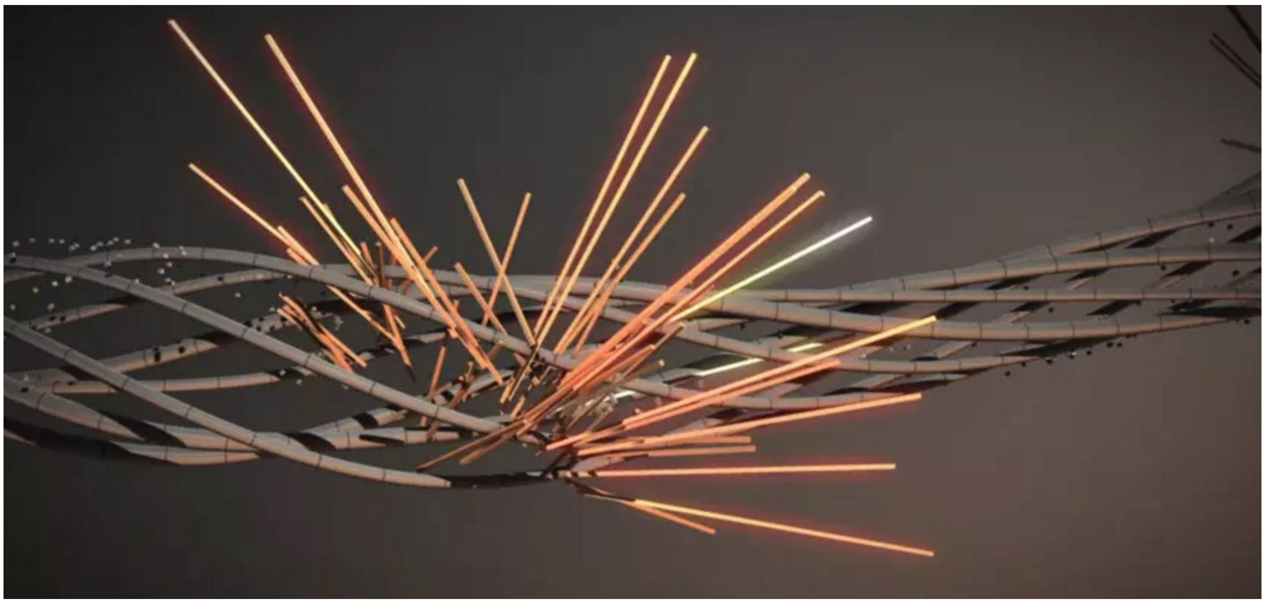


Figure 3: Project Partitura

II. Stakeholder Analysis

In order to understand a project like Processing, it is beneficial to identify the various individuals and groups affiliated with the project, and understand how their needs and motivations influence Processing. These individuals and groups are known as *stakeholders*, and can be categorized into classes by their role in relation to Processing. In this section we will identify several stakeholder classes and discuss their influence on Processing.

Figure 4 shows a graphical representation of the Processing stakeholders. Detailed discussion on stakeholder classes will follow.

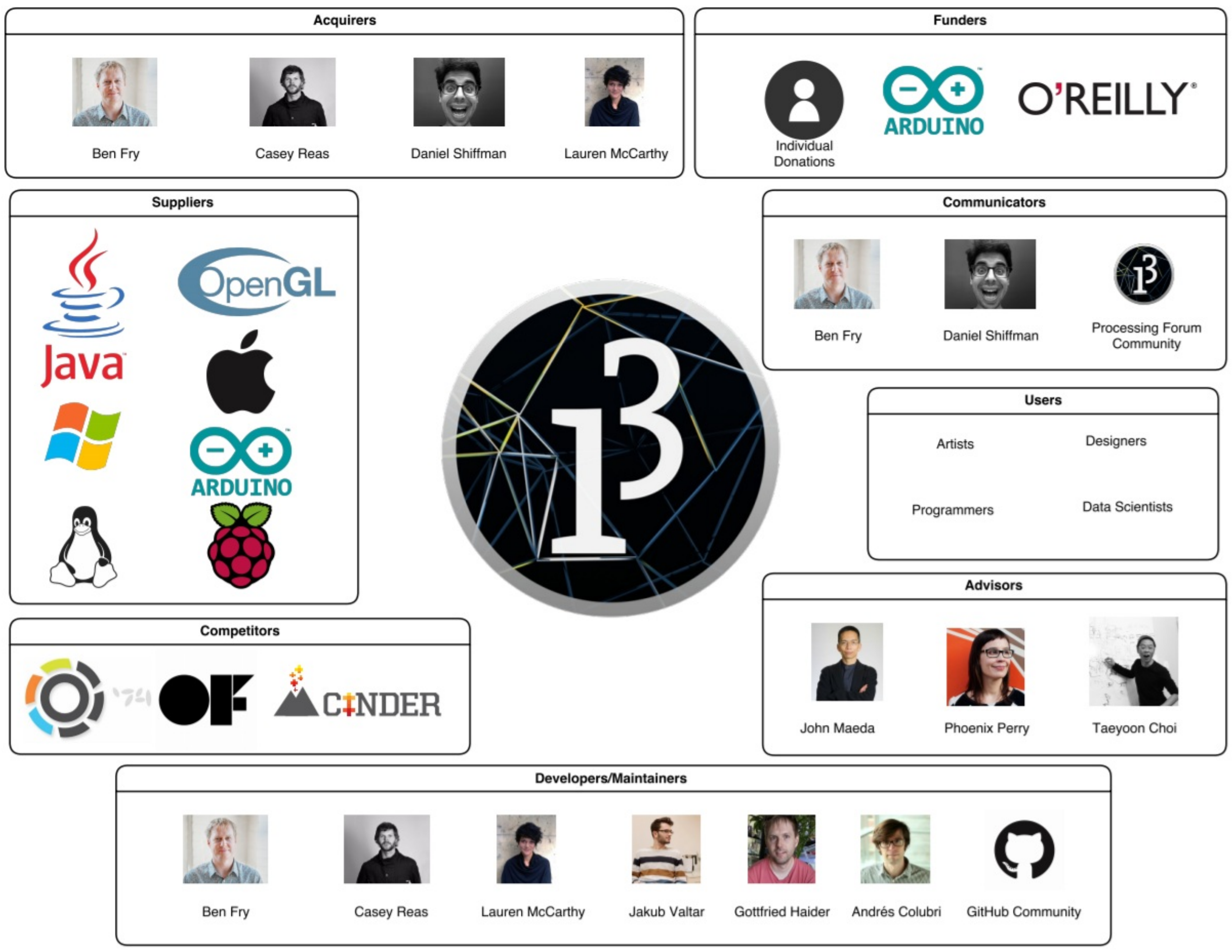


Figure 4: Processing IDE Stakeholder Diagram

Classes of Stakeholders

First, we will examine several classes of stakeholders originally identified in *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* by Rozanski and Woods [1].

Acquirers

The major stakeholders of Processing in the acquirer class are senior members of the Processing Foundation. As members of the Processing Foundation Board of Directors, these individuals oversee the ongoing development of Processing and related projects at a high level. It is important to note that many of the acquirers for Processing also belong to other stakeholder classes, such as developers and communicators. To reflect this, the table below lists the major acquirers of Processing along with information on other stakeholder classes they may belong to.

Table 1: Processing Foundation Board of Directors

Name	Class	More info
Ben Fry	Acquirer, Communicator, **Developer**, Maintainer	http://benfry.com/, https://github.com/benfry
Casey Reas	Acquirer, Communicator, Developer, **Maintainer**, User	http://reas.com/, https://github.com/REAS
Daniel Shiffman	Acquirer,**Communicator**, Developer, Maintainer	http://shiffman.net/, https://github.com/shiffman
Lauren McCarthy	Acquirer,Communicator, **Developer**, Maintainer	http://lauren-mccarthy.com/, https://github.com/lmccart

Communicators

There appear to be two major online platforms through which Processing communicators can interface with other stakeholders. The first is GitHub, where developers can communicate with acquirers and other developers about issues and proposed modifications to the source code. There is significant overlap between the primary communicators on GitHub and the main developers, namely Ben Fry, Jakub Valtar, and Gottfried Haider.

The second platform used for communication is the official Processing forums, which is a more general-purpose platform where end users can ask questions and discuss Processing. The two forum administrators are Philippe Lhoste and Cedric Kiefer [3].

Developers and Maintainers

Since Processing is an open-source project that is actively contributed to on GitHub, we can get a good idea of the stakeholders in the developers class simply from examining the repository commit logs. In particular, this can help us narrow down which of the individuals listed on the Processing site as developers are relevant stakeholders to our project (i.e. the Java variant of Processing).

Here are three of the top GitHub contributors in the last year (who were not already included above as part of the Processing Foundation) :

Table 2: Top Contributors

Name	Class	More info
Jakub Valtar	**Developer**, User	http://www.jakubvaltar.com/, https://github.com/JakubValtar
Gottfried Haider	**Developer**, User	https://github.com/gohai
Andrés Colubri	**Developer**, User	http://andrescolubri.net/, https://github.com/codeanticode

Suppliers

Processing has numerous stakeholders who can be classified as suppliers. These suppliers can be broken into two major categories: software dependencies (existing software developed by third parties which Processing utilizes) and hardware platforms (that Processing may be run on).

Software Dependencies

- Java 8
- JavaFX
- OpenGL

Hardware Platforms

- Windows
- Mac
- Linux
- Android
- Arduino
- ARM-based platforms (e.g. Raspberry Pi)

Additional Classes of Stakeholders

In addition to the previous stakeholder classes from Rozanski and Woods, we have identified three additional classes present in Processing: funders, competitors, and advisors.

Funders

Individuals or groups that directly contribute to the project financially.

The support page of the Processing Foundation details the sources of funding for Processing and all related projects [4]. First and foremost, Processing is supported by individual donations of 5 to 100 USD. However, Processing also receives funding from corporations such as Arduino and O'Reilly Media, and numerous academic and design institutions.

Competitors

Groups involved with the development of projects with similar functionality or goals.

There are a number of projects and products which share Processing's goal of uniting programming and creative arts. Some of the more popular ones include Max, a visual programming language for audio manipulation and synthesis, openFrameworks, a C++ toolkit for visualization and artistic expression, and cinder, another C++ visualization library.

Advisors

Individuals or groups who provide high-level advice and guidance regarding major design decisions, but are not necessarily intimately familiar with the use or implementation of the software like an end user or developer. Can include individuals who are experts in fields related to the project, as well as organizations or individuals that use the software.

The Processing Foundation Board of Advisors serves to guide the decisions of other major Processing stakeholders. It does this in two ways: by drawing on the knowledge of highly experienced individuals in the field of design (namely John Maeda) and by examining the needs and suggestions of some individuals who represent the typical Processing user (Phoenix Perry and Taeyoon Choi).

Table 3: Processing Foundation Board of Advisors

Name	Class	More info
John Maeda	**Advisor**	https://maedastudio.com/
Phoenix Perry	**Advisor**, User	http://phoenixperry.com/
Taeyoon Choi	**Advisor**, User	http://taeyoonchoi.com/

III. Context View

In order to properly analyze a system, it is naturally convenient to know in which environment, and with which actors, the system interacts, both internally and externally. Such a high-level overview can be obtained by creating a context view of said system. This context view describes a system's dependencies, as well as its interactions with external entities. See below for the context view illustrating the mentioned information in the case of the Processing IDE.

System Scope

Programming is seen by many beginner students as black magic. The fact that every *Hello World* example in many popular languages does not produce much visual feedback makes learning to program an even more daunting and challenging task.

Processing aims to make programming much more accessible by adding this dimension of visual feedback to the beginners who used it. It does this by taking Java, simplifying its syntax, and adding extra functions that simplify creating visualizations programmatically. However, it does not lose its flexibility and power because of these simplifications.

The scope of Processing is therefore to ease non-programmers into the world of programming by allowing them to experiment with a language that is able to provide instant visual feedback.

Visualization

Due to the simplicity of the Processing language, and the fact that it runs on most platforms, the language is mostly used as an introduction to the world of programming. Its importance as an introductory language is further illustrated by the multiple educational companies and institutes that fund the project, such as O'Reilly and New York University. Processing has a very active community, as can be seen observed from the thousands upon thousands of threads on the official Processing forum [5]. While Processing has an official community forum, mainly GitHub is used for the interaction between users and developers. It is the site where all development takes place, manuals and tutorials are hosted, and issues are tracked.

The state of this diagram is expected to remain quite constant throughout further development of Processing. The system will continue to be implemented in Java, which means that its cross-platform capabilities will not change and will continue to run on the current multitude of platforms. Its dependencies, JavaFX and OpenGL are actively maintained and developed software systems, meaning that they will not be replaced any time soon. Finally, given GitHub's position as market leader in version control repositories and the sheer amount of work already invested in working with GitHub, it is extremely unlikely that any switch will be made to another competitor. Figure 5 shows a visualization of the context view of the system.

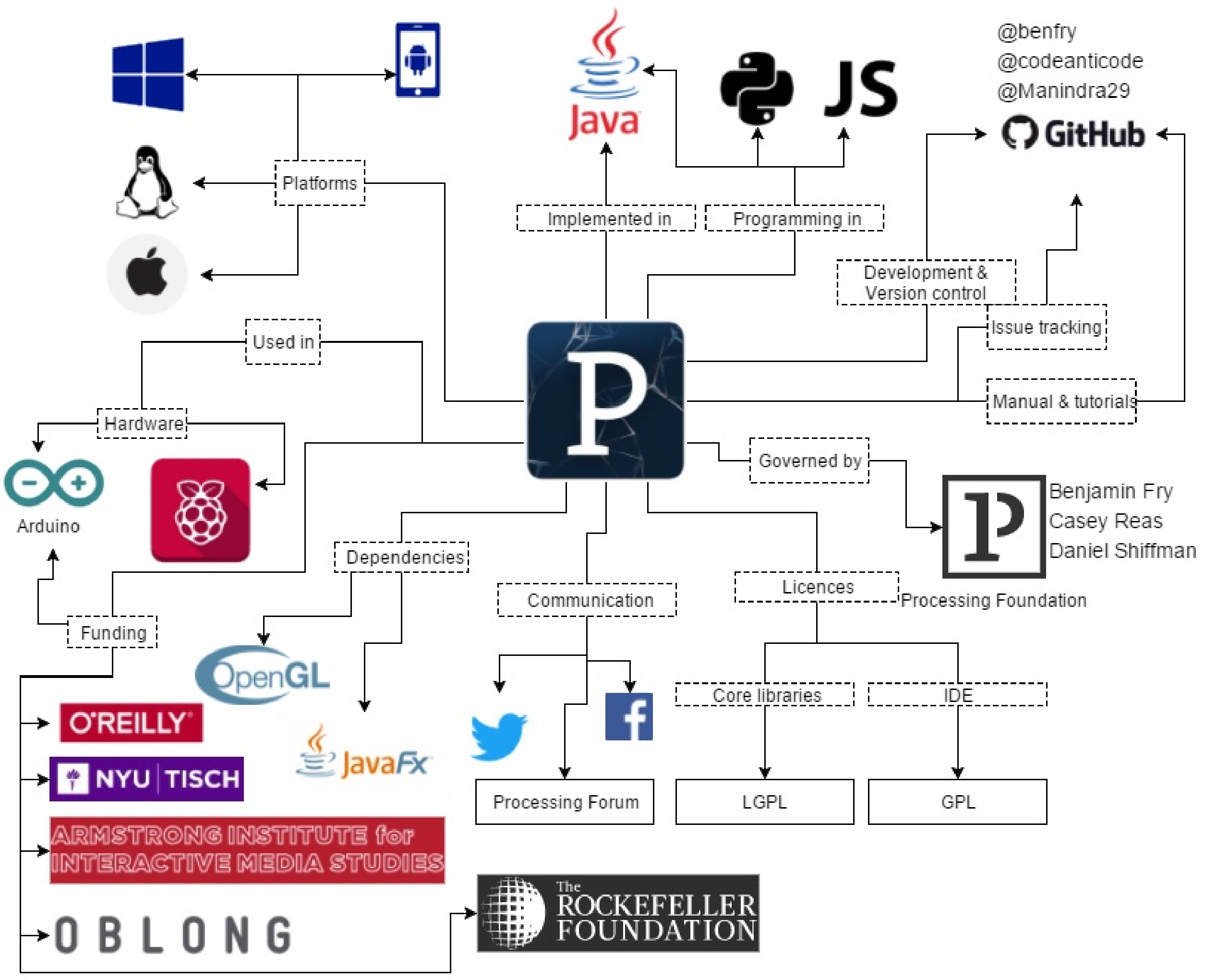


Figure 5: Visualization of Processing's Context View

IV. Deployment View

The Processing IDE is not a stand-alone piece of software. That is, it requires additional software to be able to run correctly and successfully. Furthermore, in order for said additional software to run, other system requirements must be met. This section will mention these requirements and constraints. Figure 6 illustrates the deployment view of the system.

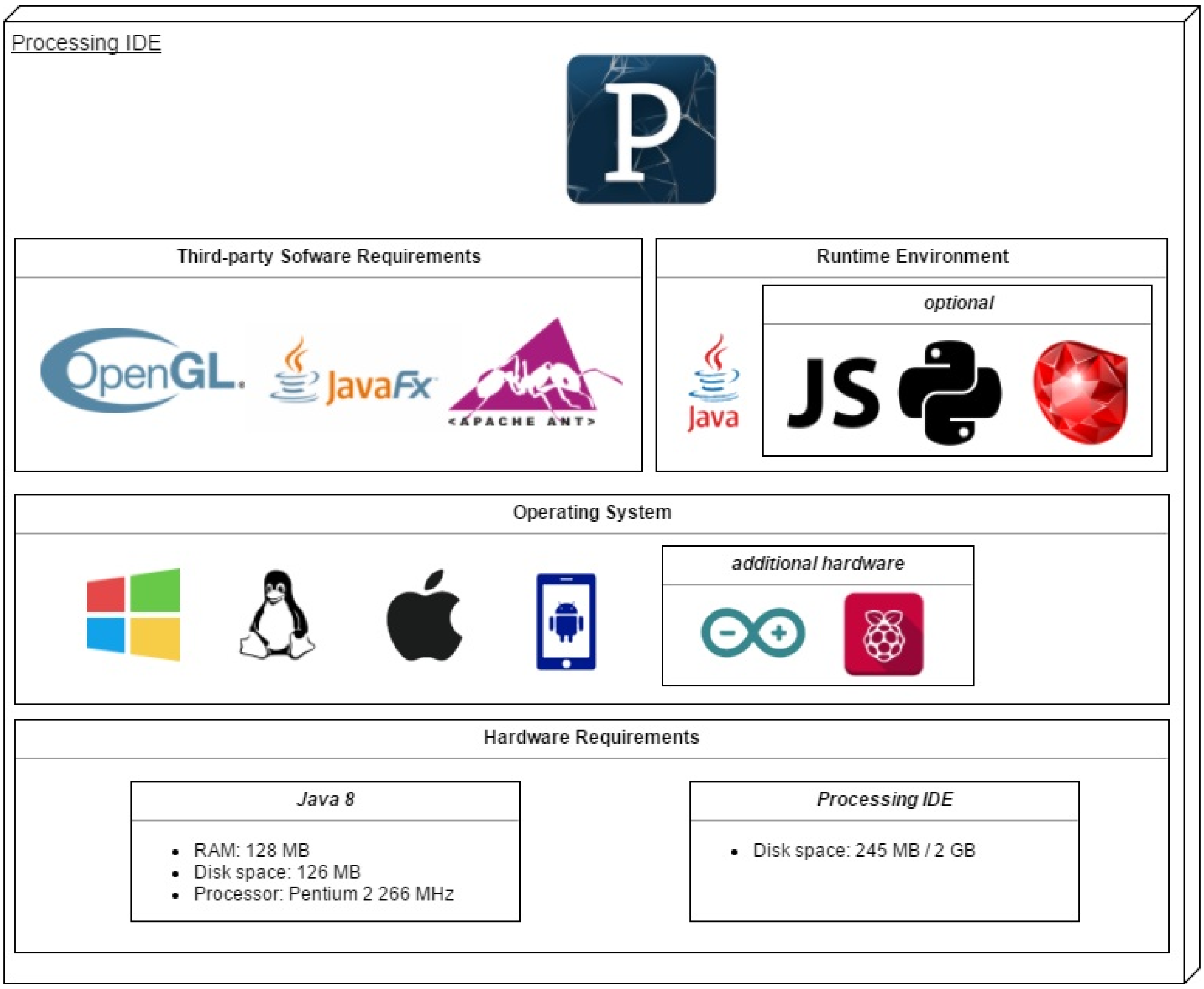


Figure 6: Deployment view Processing

Third-party Software Requirements

Processing needs a few third-party software packages in order to be used. Apache Ant is needed in order to build the software from its source code. JavaFX is also a necessity, since it is used to build the GUI of the system. Since the release of Java 7 update 6, however, the JavaFX libraries have been bundled together with the standard Java SE, and as such, does not have to be downloaded and installed separately [6]. Lastly, Processing depends on OpenGL for the creation of images and visualizations, which is one of the core selling point of Processing [7].

Runtime Environment

Since Processing is an extension of the Java language, the Java Virtual Machine (JVM) is ultimately used to run Processing code. This, however, is just the default option. Processing also offers different *Modes*, which allows users to write their programs in different languages, or make them able to run on different environments. These *Modes* include JavaScript, Python, Ruby, and Android *Modes*. All of these previously mentioned languages have their own runtime environment.

Operating System

As this software is built on Java, which is cross-platform, the software itself is cross-platform as well. It is able to run on Windows, Linux, and macOS. Being able to run on Linux also gives it the ability to run on Raspberry Pi systems, which support the Linux operating system. Processing applications can also be run on Android systems, if Android *Mode* is used when creating the application.

A special case is Arduino. The Arduino board does not run any operating system, yet Processing code is still able to run on the hardware [8]. This is because Processing programs are first compiled to binary files before being executed [9].

V. Architecture Development View

The development view of a system illustrates aspects of the software development process. These aspects are code structure and dependencies, build and configuration management of deliverables, system-wide design constraints, and system-wide standards [1].

This section will describe the system from a developer perspective, and will be concerned with module organization, common processes, standardization of design and testing, and codeline organization.

Module Organization

The module structure of Processing deals with the system's source code in terms of modules. Processing has developed a well structured model. There are six main modules in Processing namely: Core, Data, Language, App, Third party and build. Figure 7 illustrates the different modules and their components.

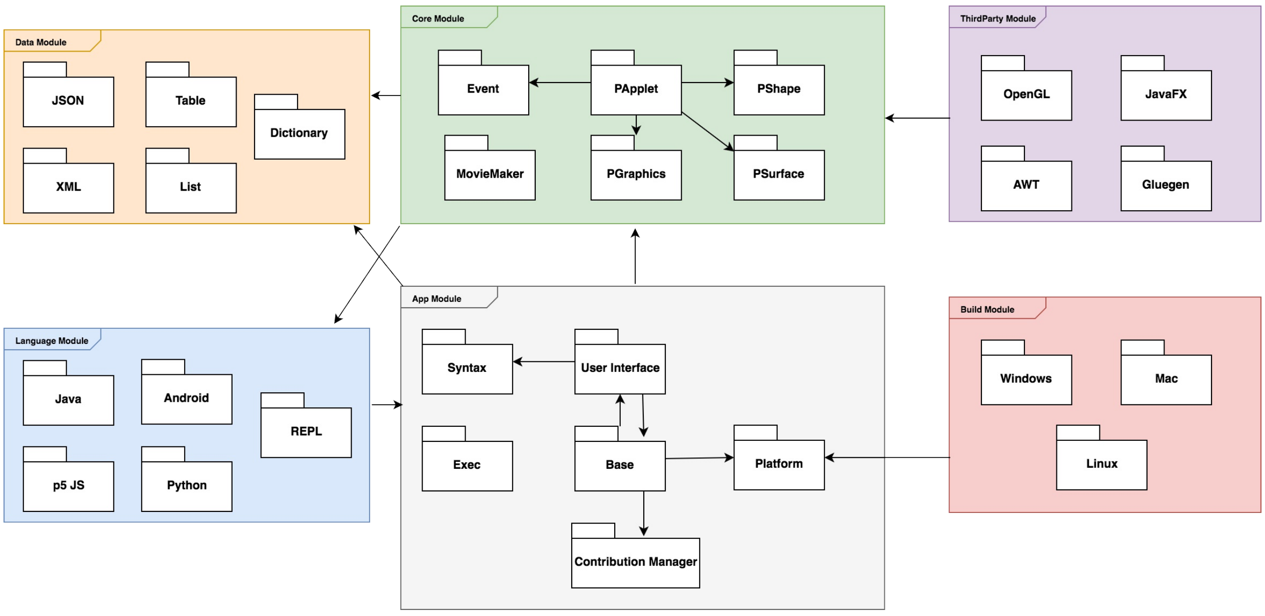


Figure 7: Processing's Development View

Core Module:

This module handles the rendering and displaying of 2D and 3D graphics including drawings and animations. It is considered the core module as it contains PApplet, PGraphics, PShape, PSurface, Event and MovieMaker.

Table 4: Core Module Contents

Component	Details
PApplet	This is base component for all sketches in Processing. It includes drawing tools, handling animations, window sizing with sketches.
PGraphics	It had all main graphics and rendering context, as well as the base API implementation for Processing "core"
PShape	It is responsible for drawing, loading and saving shapes. The component supports SVG (Scalable Vector Graphics) and OBJ shapes.
PSurface	It handles the interaction with the OS (creation of a window, placement on screen, getting mouse and key events) as well as the animation thread.
Event	This component deals with user events such as key event, mouse event and touch event.
MovieMaker	It concerns on making a QuickTime movie from a sequence of images. Options include setting the size, frame rate, and compression, as well as an audio file.

Third Party Module:

It manages external libraries and resources such as OpenGL, JavaFX, AWT and Gluegen.

Table 5: Third Party Module Contents

Component	Details
OpenGL	An API used to render 2D and 3D vector graphics.
JavaFX	This package used to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.
AWT	Abstract Window Toolkit (AWT) is a set of APIs used to create graphical user interface (GUI) objects.
Gluegen	It used for generating the Java and JNI code necessary to call C libraries.

App Module:

This module mainly deals with the graphical user interface (GUI) of the application including Base, Syntax, Platform, Contribution Manager and Exec.

Table 6: App Module Contents

Component	Details
User Interface	This package includes all the GUI components in the application such as Editor, Welcome, Find and Replace, Color Chooser ...etc.
Syntax	This is syntax highlighting package which used for syntax and style checkers. It depends on the language module which deploys the program in a different programming language such as Java and Python.
Base	This is the base component which manages all the UI components, build the core components using the selected mode language.
Platform	Platform handlers for Windows, Linux and Mac.
Contribution Manager	Allows for installing, uninstalling and updating third party libraries, modes and tools.
Exec	This is an executor component which executes a given task, and handles input and output of the processes.

Data Module:

This module represents data structure which is used in the core and app module. It consists of JSON, XML, Table and List.

Table 7: Data Module Contents

Component	Details
JSON	It handles all the JSON objects and arrays.
XML	This component handles the XML files.
Table	Manages tabular data such as a CSV, TSV, or other sort of spreadsheet file.
List	A list component for all the data type including int, float and String

Build Module:

This module builds all the configuration files for the target platform such as Window, Mac and Linux. It utilizes Apache Ant to build the binaries.

Table 8: Build Module Contents

Component	Details
Windows	Configuration related to Windows platform.
Mac	Configuration related to Mac platform.
Linux	Configuration related to Linux platform.

Language Module:

This module consists of different packages of programming languages such as Java, Android, Python, p5 JS and REPL (Read Eval Print Loop). This means the user has the opportunity to choose the preferred language to code. Therefore, each component has its own compiler which then the app module is communicating with the core module to build the desired mode.

Table 9: Language Module Contents

Component	Details
Java	A package for Java programming language
Android	A package for Android
Python	A package for Python programming language
p5 JS	A package for JavaScript
REPL	A package for Read Eval Print Loop (REPL)

Dependencies

As seen in the module organization previously, the core module depends on the data module which provides its data structure for the drawings and rendering in the PGraphics, PSurface and PShape component. Also, it depends on third party module which provides external libraries in graphics such as OpenGL and JavaFX. The PApplet component, which acts as the main controller in core module, relies on PGraphics, PSurface and PShape. The App module depends on the language module to compile the chosen programming language by the user. Also, the setup of Processing depends on the build module which provides the necessary packages to run the application in different platforms.

Common Processing

When designing the system's software elements, it is desirable to define a set of design constraints in order to maximize commonality across element implementations. The reason for this is because code duplicates are avoided, and it can improve the system's overall technical coherence which makes it easier to maintain and understand [1].

Message Logging

One of the common design models is message logging. The Processing project has ensured that all components log human readable messages. Each log record has a date, time, sequence, logger, class, method, thread and message. A record also has different levels such as fatal, error, warning, information and debug. An Error message gets displayed with its trace, which makes it easier for the developer to debug and fix the issue later. In general, log records are clear and structured.

Internationalization

The Processing team made sure that hard coded strings are not used. Instead, strings are fetched from one of 12 "property files", where each file includes the used text in a certain language. On application start-up, keys and values are loaded and then mapped to their respective UI elements.

Standardization of Design

As the project grows and has more contributors, the Processing team needed design standardization in order to maintain the project. Processing achieves this by using some design patterns, style guidelines and philosophical principles.

The second design is the plugin modules. The Contribution Manager component allows to add different modes, third party libraries and tools. This design enables third party developers to create abilities which extend an application. Also, it supports easily adding new features which provides flexibility and scalability.

Processing team stated a principle that "GUI components do not live in the base package" . The base package is the main package of processing project, and all the GUI components should be declared at the user interface package. The primary role of the base package is for platform identification and general interaction with the system (launching URLs, loading files and images).

Another design principle is that inner classes should not be accessed by classes that are not in the same file. This rule will allow other developers to maintain the project, and save performance time especially when it is being used a lot.

Instrumentation

Similarly to testing, the Processing IDE has no proper instrumentation implemented. The only thing that comes close to instrumentation is message logging, which is discussed in the *Common Processing* section above. And, as is also the case with testing, verifying that code runs properly and does not consume unnecessary resources is completely left to the author of said code.

In the case that neither the author of the code nor the developer who accepts the pull request spot the faulty code, it will continue to live on in the system until another user comes across it and is able to identify the flaw. An example of this is issue #4825. In this particular case, faulty code that was written several years ago that produces an out-of-memory error in certain cases. This memory leak could have been identified much sooner if proper instrumentation was implemented to track system resource usage.

Codeline Organization

The Processing source files are organized based on their functionalities as shown in figure 8. Every folder mostly consists of Java classes and a build.xml file. In order to speed the build process, Apache Ant is used build the binaries. In addition to automating the build process, Ant makes it easy to add JUnit test cases. Test cases have been added to the App folder.

This structure (in figure 8) makes it possible for Processing developers to do continuous integration. For instance, as there are a lot of changes in the Core folder, it is separated into different components such as `data` , `event` , `awt` , etc., depending on the functionalities. This will ensure that the system's code can be managed, built and tested when using an iterative development and release process.

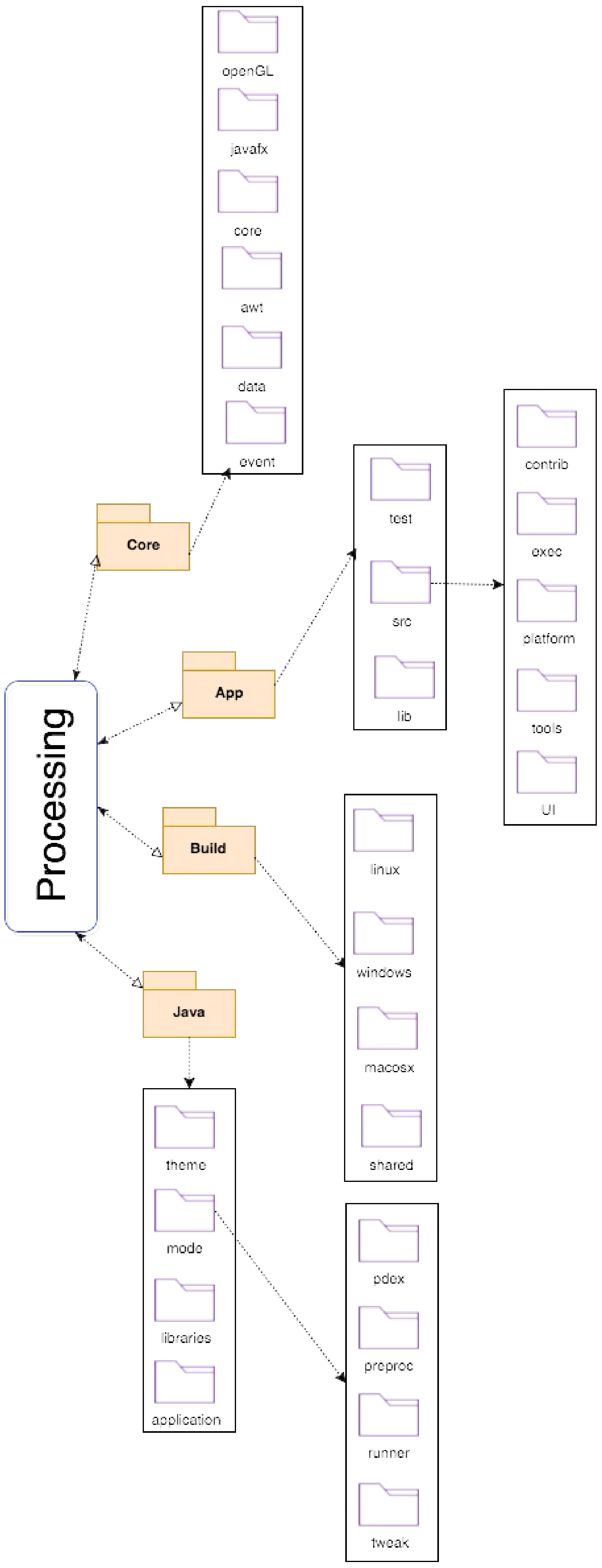

Processing
Core
openGL
javafx
core
awt
data
event
App
test
src
lib
contrib
exec
platform
tools
UI
Build
linux
windows
macosx
shared
Java
theme
mode
libraries
application
pdex
preproc
runner
tweak

Figure 8: Codeline Organization of Processing

VI. Technical Debt in Processing

Technical debt is a metaphor for incomplete and inadequate artifacts in the software development which cause higher costs and lower quality in the long term [11]. Technical debt can be in many forms such as design/code debt, defect debt, documentation debt and testing debt. This section will discuss these debts in Processing project.

Identifying Technical Debt

One of the ways to identify technical debt is the overall quality of design and code. There are a number of techniques used to quantify technical debt such as SOLID (**S**ingle responsibility principle, **O**pen/closed principle, **L**iskov substitution principle, **I**nterface segregation principle, **D**ependency inversion principle) violations, code smells and ASA (Automatic Static Analysis). This section will identify the design/code debt in Processing using some of these techniques.

Single Responsibility Design (SRD) - Discovering Violations

A direct violation of the Single Responsibility Principle is having a god class. A god class is an object that controls way too many other objects in the system and has grown beyond all logic to become the class that does everything. In processing, the PApplet class acts like a controller for the data module, as well as the PGraphics, PShape, PSurface and Event classes. There are many possible actions which PApplet performs such as reading and writing different types of files such as XML, JSON and Table. It can also load images, and do mathematical calculation such as sin, cos, tan, etc... Also, it does all the drawings and animations which depend on the PGraphics class. God classes can be classified as code smells too.

Dependency Inversion Principle (DIP) - Discovering Violations

This principle states that "high level modules should not depend on low level modules" [12]. The interaction as seen in figure 9 is between two level modules (Base and PApplet).

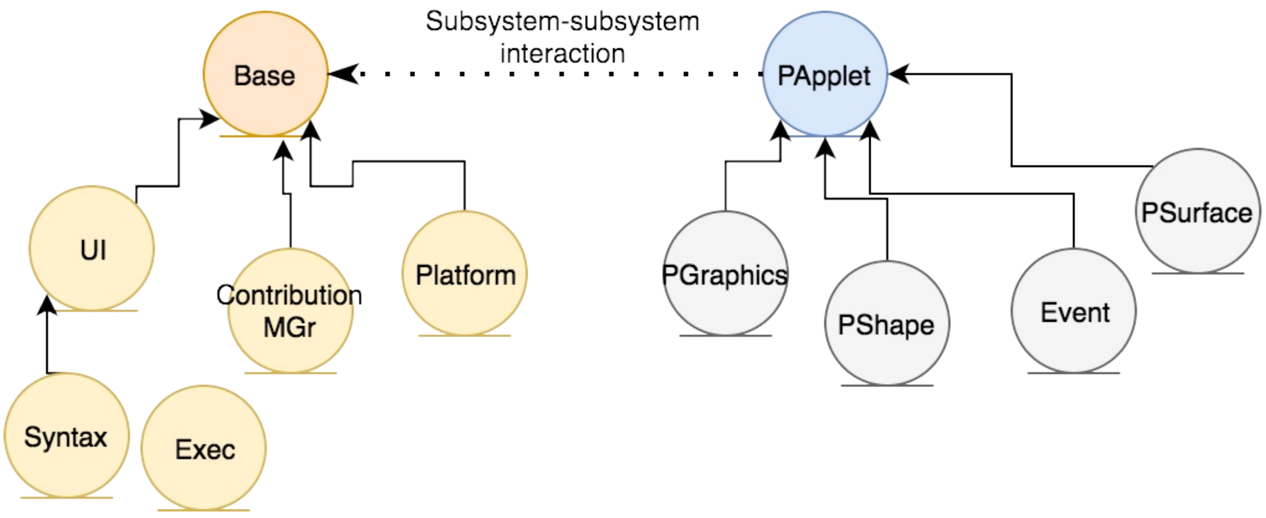


Figure 9: DIP violation in Processing

Because the interaction between these subsystems is at different module levels, the Base needs to interact with a higher module than the PApplet. The refinement of the design would be having an abstract interface class called "Core Service Interface" as illustrated in figure 10.

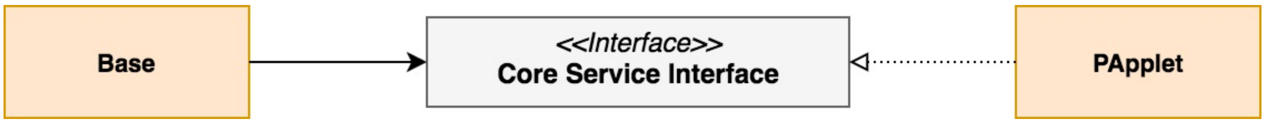


Figure 10: Resolving the DIP violation

Now, the base module is not infected by the changes in PApplet because of dependencies inversion which created a structure that is more flexible and durable.

Code Smells

In addition to the god class that exists, another code smell, namely **Large Classes**.

The PApplet class has more than 13,000 lines of code with more than 100 methods. The PGraphics class has more than 7,000 lines of code. The table below shows the four biggest classes in the project. Methods are mostly small and tractable, and a portion of the lines of code is dedicated to documentation. However, further decomposition into several classes make the codeline organization better, and solves this issue, in addition to the God Class issue mentioned above.

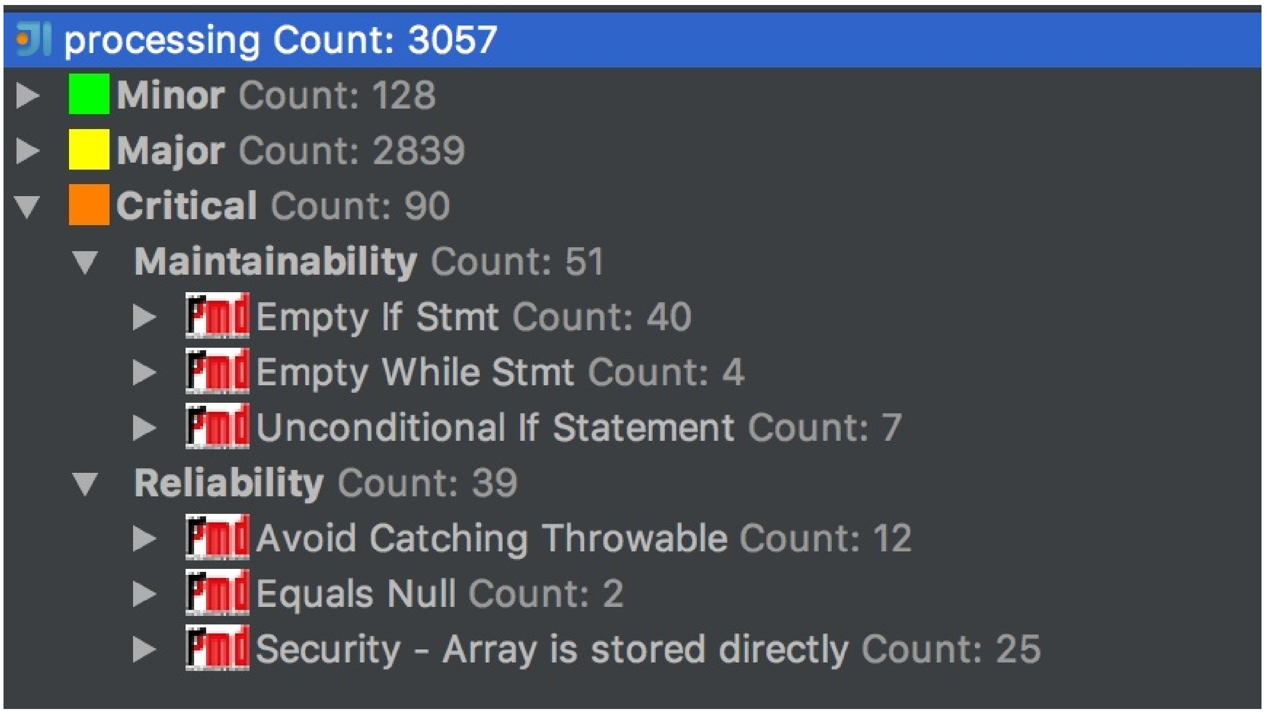
It can be argued that for PGraphics, PImage, and PShape, a large amount of documentation exists, significantly increasing the lines count, and that graphics-related code quickly accumulates lines of code due to the overhead required for functionality e.g. settings up colors, lines, vertices, etc, meaning that the LoC count isn't very large relative to the application. However, decomposition is still a viable option to increase code structure quality. This will further help avoid issues similar to these high priority issues #4897, #4894, and #4895, later on, and make development pace faster.

Table 10: Lines of code in the largest Processing classes

Class	Lines Of Code (LOC)
PApplet	13,545
PGraphics	7,188
PImage	2,974
PShape	2,720

Static Code Analysis

The QAPlugin tool has been used to test the code quality of the project. It helps statically analyze the code to identify potential problems. Figure 11 shows a snapshot of the results.



```
processing Count: 3057
▶ 🟩 Minor Count: 128
▶ 🟨 Major Count: 2839
▼ 🟧 Critical Count: 90
    ▼ Maintainability Count: 51
        ▶ Empty If Stmt Count: 40
        ▶ Empty While Stmt Count: 4
        ▶ Unconditional If Statement Count: 7
    ▼ Reliability Count: 39
        ▶ Avoid Catching Throwable Count: 12
        ▶ Equals Null Count: 2
        ▶ Security - Array is stored directly Count: 25
```

Figure 11: QAPlugin output for Processing

The tool found about 90 critical statements in the project. There are 40 empty `if` statements which only evaluate the condition, and do nothing else. Also, there are about four empty `while` statements which are not even timing loops. No reason can be thought of that justifies having these problems. Some of them are intentional for sure. However, documentation to include these must be improved, and unintentional critical statements should be addressed. These probably have an indirect effect on performance. Theoretically, these do not affect the maintainability, as those parts of the code probably do not get touched quite often (or at all), or the developers would have noticed and fixed them.

Testing Debt

Testing is a very interesting aspect of the Processing project, namely because the amount of testing throughout the code is minuscule. The project does not make use of any continuous integration system that makes sure the project builds correctly after each commit. It is the sole responsibility of the contributor to verify whether the proposed code alterations do not break the system's build process.

From a code coverage perspective, Processing's testing situation is not the best. The entire project is composed of around 174,000 (174 thousand) lines of Java code, including comments. This number can be reduced to 150,000 lines of code if 20,000 lines are assumed to be comments. The `PdePreprocessor.java` file, which is the only tested class in the entire project where it focuses on the preprocessing engine and compiler, contains 1,295 lines of code, excluding comments. This means that total code coverage is around 0.9%. Although 100% code coverage is generally not necessary [13], properly testing barely a single percent of the total amount of the code is not appropriate, particularly for a project of this size.

The project's wiki page does not mention any testing procedures or methodologies [10]. In summary, testing is left to the code contributors themselves. Since there is no standard methodology, testing is done in a very inadequate, ad-hoc manner. This is mainly due to the main developers' time constraints, and even though skipping testing saves quite some time, this sets a dangerous precedent that will likely create significant technical debt.

Evolution of Technical Debt

Technical Debt in Processing is visible when new features are added. The design of the Processing core has been around for 13 years, and there have been significant changes. This section will analyze how the processing system evolved in terms of technical debt.

Since version 3 of Processing, developers have started migrating functionality away from old and outdated code. Speeding up this process, as well as properly documenting the new code will significantly help the progress towards improved testing debt. This way, old, untested, and undocumented code could be replaced by well-written, documented code that will be much easier to test by official contributors as well as external developers.

Core

The first version (1.0) was almost stable as there were only a few components and fewer people working on the project. However, at version 2.0, the Processing team had the project out of control as there were drastic changes made in the graphic features such as OpenGL, VolatileImage, BufferStrategy...etc. Not only did they need to cope with these changes, but also they had a performance issue in PApplet. Applet at that time was the base class for PApplet.

At version 3.0, the Processing team decided that it was time to maintain "clean code". PApplet is now no longer the base class, and they had to redo the entire rendering and threading model for Processing sketches. This shows how hard it was to make an improvement change.

OpenGL (JOGL vs LWJGL)

Team Processing were also confused about which OpenGL library they should be using (JOGL or LWJGL). JOGL had some major issues with the development and was stopped in 2014. The team had been trying out LWJGL2 to see how it fared. Then, the LWJGL project moved all their development effort to LWJGL3. The Processing team spent almost a week rewriting OpenGL to use LWJGL3. They decided to go back to JOGL and fix issues since LWJGL was too unstable to use, and would require major reworking of PApplet to remove all uses of AWT.

VII. Evolution Perspective

Many are familiar with the following old adage:

> The only constant is change.

It holds true for many things, and certainly for software development. Software is continuously changing. This section will discuss how Processing has evolved over time with each of its major releases. Bug fixes and minor changes will be left out in favor of the changes that altered the functionality or are relevant to the architecture of the system. In other words, changes that affect the stakeholders that work with the system. Versions 1.0, 2.0, and 3.0 will be discussed.

Processing 1.0

- Sound and XML libraries were added.
- Jikes compiler switched with ECJ compiler.
- Updated to Java 6
- New global Processing functions `loadShape()` and `requestImage()`

Processing 2.0

- New global Processing functions `rect(x, y, w, h, radius)` and `rect(x, y, w, h, tl, tr, br, bl)`.
- Own P2D and P3D renderers replaced with OpenGL renderer.
- Upgraded to OpenGL 2.
- OpenGL is built into Processing core.
- Added *Modes* to make coding possible with other languages.
- Support for Java **Applet** is removed.
- List of imports is now hardcoded and no longer read from an external text file.
- Rewrite event handling to support OpenGL events. Removed support for `java.awt.*` events.

Processing 3.0

- New global Processing functions `pixelDensity()`, `fullScreen()` and `displayDensity()`.
- Video and Sound libraries removed by default.
- Variables `displayWidth` and `displayHeight` deprecated.
- UI code moved from `processing.app` to `processing.app.ui`.
- Several (static) utility functions from `Base` have moved into separate utility classes.
- Update to Java 8.
- Added "Contributions Manager" for downloading compatible third-party libraries and modules.

The quality of Processing's evolution is mixed. On the one hand, one can see that some effort is being made to clean up the codebase, and therefore reduce the technical debt of the system. This can be observed from the refactoring of the UI code in release 3.0. Additionally, developers seem to value up-to-date software, given their tendency to remove support for outdated classes/features and update the used third-party software to the latest version: OpenGL to version 2 and Java from version 6 to version 8. Some design decisions, however, are not in the best interest of code and architecture quality. One of these bad decisions is continuously adding global functions to the scope. Another is creating static utility functions, as well as the decision to split them up into separate utility classes, as this violates the principles of object-oriented design.

As the software gains popularity, the quality of development, no doubt, increases. From an architectural point of view, every major release has made significant improvement upon the state of the software, and it is expected that this will continue down the line. Undocumented, obsolete, out-of-date code will be replaced by documented code that is more maintainable and testable. Though it will take a lot of time and effort, the necessary steps are already being taken.

VIII. Conclusion and Discussion

This chapter provides a brief overview of Processing. It begins with identifying stakeholders, and how their motives started and keep Processing up-and-running. It later clarifies the role Processing plays from various aspects -its context, deployment, and evolution. Afterwards, technical discussion begins with an in-depth analysis of the architecture of this software, and the technical debt it accumulated throughout its years.

Processing is a great and very handy software. Our love for it and its unique set of features is what motivated this chapter, and is what motivates others to keep using it, as can be seen from the interview available in the Appendix. However, some issues in the development process need to be addressed. First of all, the amount of technical debt is staggering. Testing should be added to allow for a better grasp of the project functionality, and make sure that changes are not breaking anything. This will also allow contributors to submit higher-quality pull requests that do not break undocumented aspects, which only the experienced developers know (e.g. here).

The mentioned issue is largely a consequence of another out-of-hand issue, which is the lack of strong community development support due to the demographics of the project users. Take a developer-focused open source project -Node.js for example; since it's a tool for developers, developers will report well-documented issues and submit meaningful pull requests to help advance the project for their own advantage and the whole community. In contrast with Processing, where most users lack experience in software engineering, submissions from the community are scarce, and often extremely simple. The developers of this project would have to choose between feature improvement and development on one side, and adding tests or extra documentation on another.

In the end, a well-deserved salute is addressed by the chapter authors to the developers of Processing for their continuous work and stamina that keeps this project alive and ever amazing (and useful!).

IX. References

[1] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.

[2] Processing Homepage. Accessed 4/2/2017. Retrieved from: https://www.processing.org/

[3] People \ Processing.org. Accessed 20-3-2017. Retrieved from: https://processing.org/people/

[4] Processing Foundation -- Support. Accessed 20-3-2017. Retrieved from:

https://processingfoundation.org/support

[5] Processing 2.x and 3.x Forum (Accessed 2-4-17). Processing.
https://forum.processing.org/two/

[6] Oracle. JavaFX FAQ. Accessed 19-3-2017. Retrieved from
http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#5

[7] Overview \ Processing.org (Accessed 2-4-17). Processing.
https://processing.org/overview/

[8] Arduino Playground - Processing (Accessed 2-4-17). Arduino.
http://playground.arduino.cc/Interfacing/Processing

[9] Build Process - arduino/Arduino Wiki (Accessed 2-4-17). Arduino.
https://github.com/arduino/Arduino/wiki/Build-Process

[10] Processing. processing/processing Wiki. Accessed 5-3-2017. Retrieved from
https://github.com/processing/processing/wiki

[11] Advances in Computers, Volume 82. (2011).

[12] Robert C. Martin, Micah Martin. (2006). Agile Principles, Patterns, and Practices in C#.

[13] TestCoverage. Fowler, M. Accessed 12-3-17. Retrieved from
https://martinfowler.com/bliki/TestCoverage.html

X. Appendix

A1: An Interview with a user

This interview was done with Yahya Al-Homsi, an Electrical Engineering student at Qatar University. He is currently doing his graduation project on an ECMO Machine Simulator. The simulator is a hardware piece that simulates the exact functionality of a real ECMO Machine. Since those machines are ridiculously expensive, hospitals rarely have money to dedicate some of them for training purposes. Thus, a simulator would be a (very) cheap substitute to train nurses on, while giving them same experience. He is doing his work with three more colleagues, and it is co-supervised by Hamad Medical Corporation, and is being funded by Qatar National Research Fund.

- Why do you use Processing and not other tools? What is its advantages with respect to other similar software? And what do you love the most about it?

I like the built-in GUI functions that is has. This makes it really easy to create animations and shapes (2D or 3D). Furthermore, since I am a hardware guy, it's really nice to connect Processing with micro-controllers using serial communications and display information on the computer screen. In addition, what is really nice about Processing is that you can program in it using Java, Python, and more. This reduces the learning curve, since one can already use the language he knows. It also makes it really easy to have a full working program and execute codes to be a product level software.

- How did you learn about Processing?

I was searching for a way to connect the Arduino to the internet without buying the expensive Ethernet shield. I found a blog that was talking about connecting the Arduino to a processing sketch that is connected to the internet. So I stared learning about Processing. To be honest, it is always easy to do stuff with Processing. In general, the open-source hardware community prefers Processing, and most examples you find online are using Processing + Arduino together. This made my life very easy when working on projects.

- Oh, cool. Can you please give a brief on projects that you're doing/have done with Processing. Pictures would be nice!

I built an home automation system from scratch using Processing. Using it, I was able to write a software that does voice recognition, Text-to-speech, communicate with Arduino board using serial, and get data from a PHP server. You guessed it, it is indeed inspired by Jarvis from the Iron Man movie, though its replies were programmed to sound like those of Gladiators, not of a super-human artificial intelligence. I have had just finished Spartacus when I started working on the project.

Another project I have worked with involved making a small smart home controller. I built the code base using Processing, used Arduino to control the lights and the air conditioning in my room (useful when it's quite hot in Qatar), and made a webpage to control those. Here's a pic!

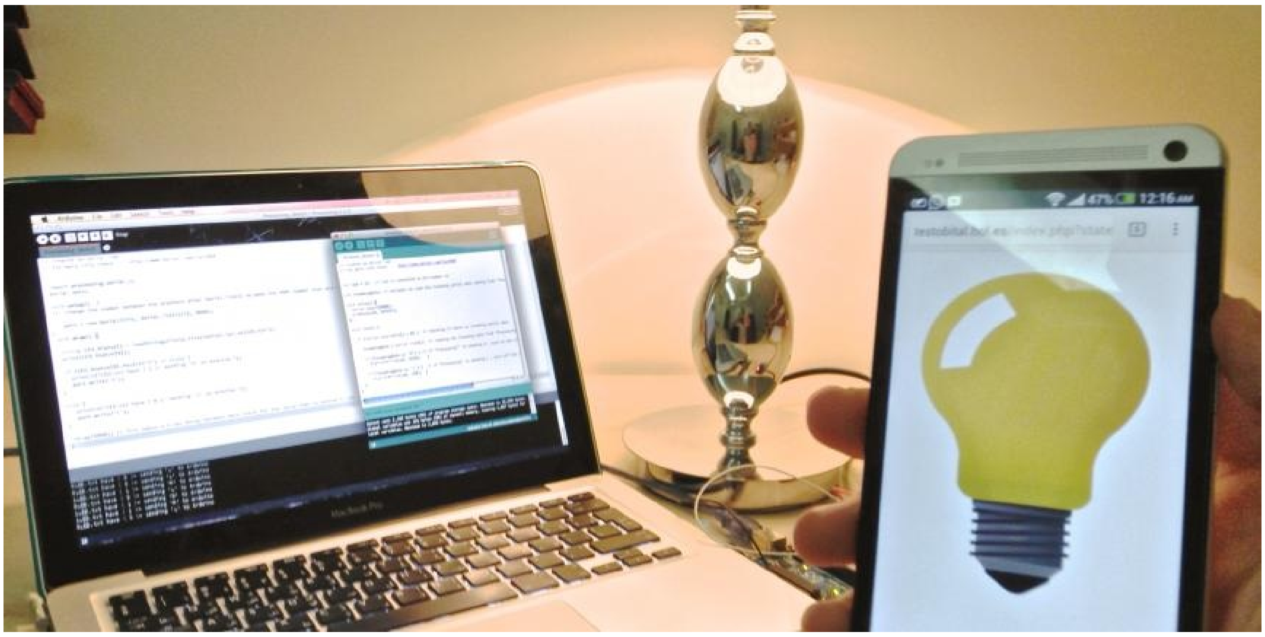


Figure A.1: Costume Smart Home Controller

The other project I am doing now is for my senior design project. The goal is to built a GUI that runs on Raspberry Pi 3. The sketch must get some values from the Firebase Database and then display them on the screen that is connected to the Raspberry Pi. The sketch should be also able to get inputs from 4 push buttons and a rotary encoder. Here's another pic! It's of the ECMO simulator control panel. An exact replica of a real ECMO machine panel.

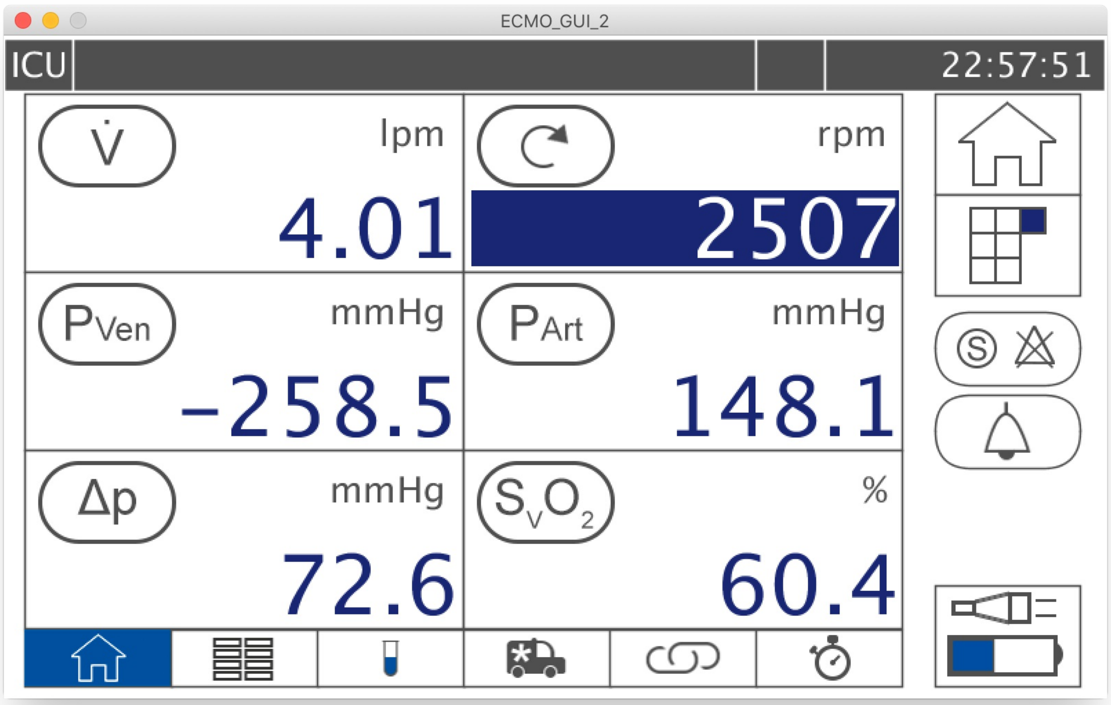


Figure A.2: Control Interface of the ECMO simulator

Scikit learn

By N.Bakker, R.Kharisnawan, B.Kreynen and C.M.Valsamos

Delft University of Technology, 2016

Abstract

Scikit-learn started as a Google Summer of code project by David Cournapeau 9 years ago. Currently it is one of the most used libraries in python regarding machine learning due to its efficiency and simplicity. Despite its small size in its core team, external developers contribute daily to the project and the project keeps growing. Being passionate data scientists we were eager to explore its framework, and thus conducted a stakeholder analysis to see who is involved with scikit-learn. We then describe scikit-learn from various viewpoints and perspectives to understand its software architecture. Finally, this chapter closes with the analysis of technical and testing debt.

Introduction

This chapter describes scikit-learn from the software architecture perspective. Scikit-learn is an open source machine learning library in the Python programming language. It has been growing and becoming more popular because of its efficiency and simplicity in use. Also, the fact that it is an open source library makes scikit-learn not only used by companies but also by individuals.

The chapter starts with a stakeholder analysis of the scikit-learn library. In this section, stakeholders are identified and an analysis is given on how they influence the library. This is done through Rozanski and Woods' classification, an additional stakeholders analysis, and a power-interest grid analysis. In addition, the project integrators are also identified. The section is followed by software views, which consists of the context, development, and deployment view. In this section, scikit-learn is explored from different kind of views to give a better understanding on how the library works, both internally and externally. Perspective on computational performance gives further explanation on performance of scikit-learn as machine learning library. The next section is software debts, which discusses technical debt and testing debt. Lastly, the chapter is closed with the conclusion of the whole analysis.

Stakeholders Analysis

Core team members and contributors

In this subsection, stakeholder analysis is explained based on a chapter in Rozanski and Woods' book; Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives Chapter 9 [2]. Stakeholders are classified into:

Acquirers

Acquirers oversee the procurement of the system or product and foresee funding [2]. Some companies and institutions have provided funding or use scikit-learn. These will have to make decisions about the acquisition of scikit-learn and might expect a return on investment of some sort. Examples include INRIA (Institut National de Recherche en Informatique et en Automatique), Paris-Saclay Center for Data Science, NYU Moore-Sloan Data Science Environment, Télécom Paristech, Columbia University, Google, Tinyclues.

Assessors

Assessors oversee the system's conformance to standards and legal regulations [2]. Contributors who review pull requests ensure that the proposed changes adhere to the standards of scikit-learn contributions. However, someone with the clear role of assessing legal regulations could not be identified.

Communicators

Communicators explain the system to other stakeholders, provide training and create manuals [2]. Scikit-learn's website is the main communication tool where its content is also managed by contributors and maintainers. There are several communication channels besides the website, such as Stack Overflow [4], a mailing list, and an IRC channel.

Developers

Developers construct and deploy the system from specifications [2]. Contributors on GitHub are developers. Some of them are sponsored by the institutions and organizations mentioned in the acquirers section.

Maintainers

Maintainers manage the evolution of the system when it is operational, they focus on developing documentation, instrumentation, debug environments, preservation of knowledge, ... [2]. This role is mainly performed by Andreas Müller who has the role of release manager[1] at the time of writing. Other maintainers are contributors in GitHub who are willing to maintain the documentation and code of the library.

Production engineers

Production engineers design, deploy and manage the hardware and software environments in which the system will be built, tested and run [2]. Andreas Müller performs this role as a release manager. Staff in the companies who use scikit-learn can also fall under this category.

Suppliers

Suppliers build and/or supply hardware, software or infrastructure to run the system [2]. They can also provide staff for its development or operation. Some of the suppliers of scikit-learn are:

- **Rackspace**: Provides cloud services for automatically building documentation [1].
- **Shining Panda**: Provides CPU time on continuous integration servers [1].
- **Github**: Provides hosting of version control.
- Acquirers of project: Some provide funding to developers, for example Columbia University [1].

Support staffs

Support staff provides support to users when the system is operational [2]. Communicators and support staff are the same group of stakeholders in scikit-learn.

System administrators

System administrators run the system once it is has been deployed [2]. As scikit-learn is a library used in python, it runs locally on whichever system the users wants to use it on. Therefore, system administrators are those people the users of scikit-learn appoint to do this task.

Testers

Testers systematically test the system to see if it is suitable for deployment and use [2]. When an enhancement is proposed to scikit-learn via a pull request, high-coverage tests are required for them to be accepted [3]. In addition, continuous integration tools automatically run these tests to ensure they pass each PR as well.

Users

Users are those who eventually use the system. Several examples of users are Spotify, betaworks, evernote, booking.com, AWeber, YHat, Research teams at INRIA, Télécom ParisTech. Scikit-learn is also suited for individual users who want to do research, or do a project involving data analysis.

Additional Stakeholders

With the categorization of stakeholders in the method proposed by Rozanski & Woods [2], we can find relevant stakeholders for most categories. However, it is noticeable that many stakeholders appear in many different categories in some way. In this section, a new categorization of the stakeholders is introduced to make the categories more distinct and the roles less intertwined. This categorization is less generally applicable, since it's tailored to scikit-learn. The different categories are contributors, users, funders and competitors.

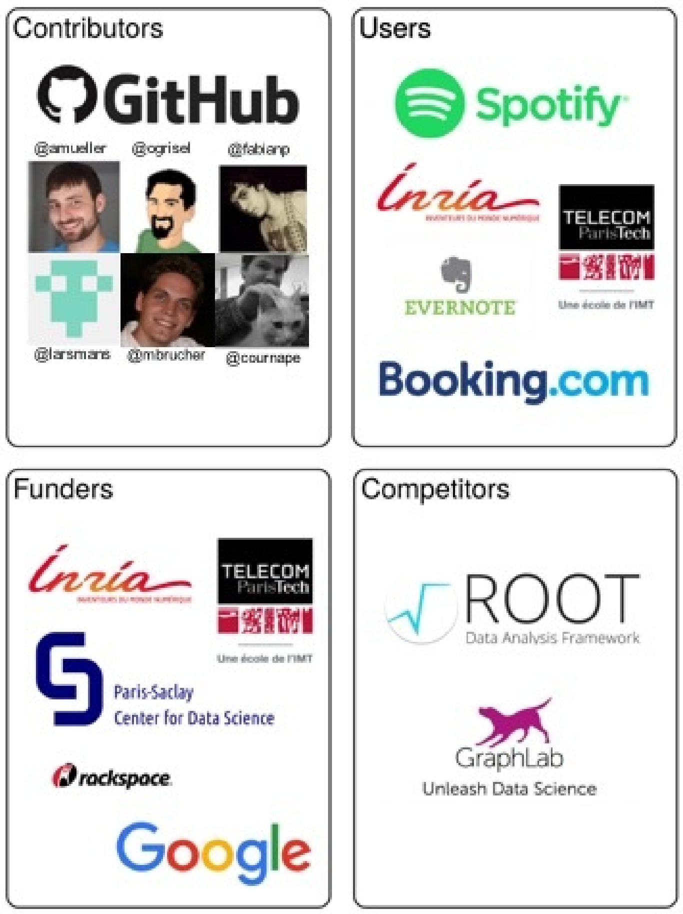


Figure 1: Visualization of the stakeholder categories for scikit-learn.

Contributors

The first category is the contributors category. These stakeholders consist of everyone who contributes on Github. In March 2017, there were 798 contributors on the Github repository of scikit-learn. They are actively involved in raising issues, development, documentation, maintenance, and testing of scikit-learn. They also help with roles that belong to the communicators and support staff in Rozanski & Woods, since the contributors also create tutorials and contribute on Stack Overflow, the mailing list, and IRC channel.

For this group, it is important that they collectively understand the architecture and functionality of scikit-learn. Since they maintain and develop the system, they are concerned with the maintainability, flexibility, documentation and preservation of knowledge of the library.

Users

As opposed to the categorization by Rozanski & Woods[2], we do not make the distinction between acquirers and users. Due to the simplicity of acquiring scikit-learn, an acquirers category seems less relevant to scikit-learn.

The users of scikit-learn are the various companies and organizations who utilise scikit-learn. There are quite a few variations between the users of scikit-learn but overall users will be concerned with documentation and ease of use since they will want to know how to use the methods implemented in scikit-learn. Reliability and correctness will also be a big concern for the users of scikit-learn. Finally, they may also be concerned about the maintainability of code which implements scikit-learn as opposed to the maintainability of scikit-learn itself.

Funders

Funding to scikit-learn can be in the material or monetary form. This category corresponds somewhat to the acquirers category of Rozanski & Woods[2]. Generally, they expect some type of return on investment but this can come in many forms. Research institutions, for example, might want to be able to influence the developtment of scikit-learn to benefit their research and might be interested in the knowledge gained from working on the development of scikit-learn. An example of this would be INRIA. Companies might also want to influence the development of scikit-learn through funding but might be more concerned with influencing it so that they can use it to get a return on investment. Finally, companies like Rackspace and Shining Panda, who provide free services, might be interested in the publicity gained from working with scikit-learn.

Competitors

Competitors to scikit-learn are other libraries which provide machine learning methods. Their developers could be interested in the development of scikit-learn to see which methods of scikit-learn they can utilize as well. However, they have very little influence on the development of scikit-learn. Examples of competitors are GraphLab [25] (machine learning library in C++) and ROOT [26] (data analysis framework in C++).

Project Integrators

The project integrators are those people whose job (or volunteered work) it is to guarantee different qualitative properties and validate changes to the project. On March 2017, there were 40 people that have the capability of doing this[5].

These integrators face a specific set of challenges for scikit-learn. Starting with scikit-learn being a very theory heavy project, which is reflected by the length of discussions on issues. The discussion are often about whether pull requests are properly implementing the expected theory correctly. Examples are PR 8253 [6] and PR 8280 [7].

Because of this complexity, they also need to handle their Integrators with extra scrutiny, requiring two of them to write off on each and every pull request. Integrators also use automated testing and demand high coverage before accepting pull requests. The process is done to further guarantee the quality under pull request checklist [8].

Power-interest grid

Figure 2 shows the quadrants of power and interest of scikit-learn stakeholders. The x-axis determines interest of stakeholders to scikit-learn which is divided into low and high interest. The interest of stakeholders is demonstrated by their willingness to explore, use, or contribute to the library. Contributions could be in the form of funding or taking part in development. The y-axis determines the power of stakeholders which is also divided into low and high power. Power is related to how influential the stakeholder is in scikit-learn's past, current, and future development. Therefore, the most powerful entities are in the upper-right quadrant and the least powerful ones in the opposite lower-left quadrant. As an example, Andreas Müller is in the upper-right quadrant because he has been the release manager since 2016, which indicates his high interest and influence in the development process. On the other hand, David and Matthieu were founders of scikit-learn [1] but they are not active in the development any more. Thus, they are classified in the high interest and low power area.

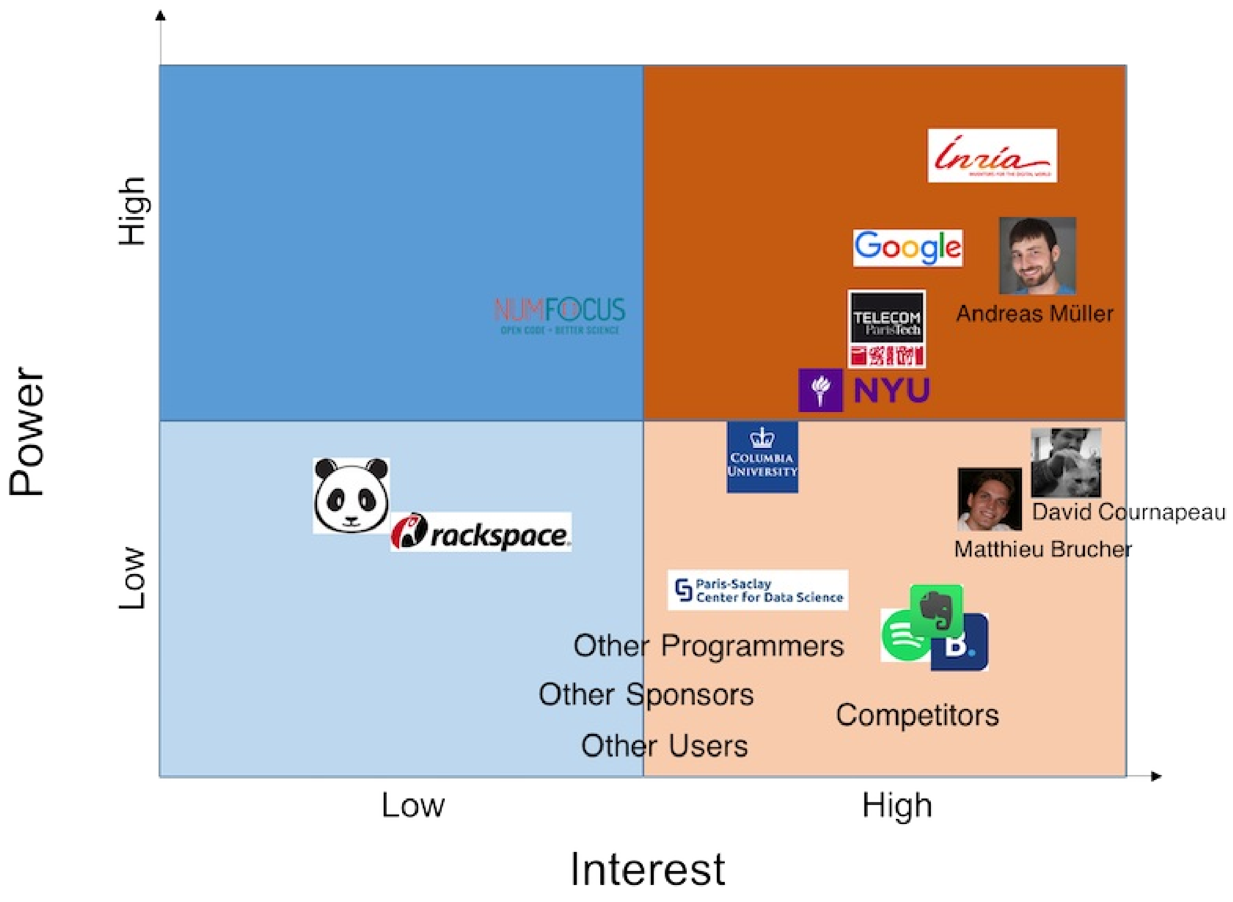


Figure 2: Power-Interest grid.

Views

In this section, we describe three views on the scikit-learn architecture: context, development, and deployment view. A view can be seen as a model of the architecture, with each view capturing different aspects of the architecture.

Context View

This section contains the relationship between scikit-learn and other related entities. It helps to identify the purpose of the system and to understand its relationship with the environment.

Design Philosophy

Scikit-learn was developed to provide easier implementation of data analysis methods, so individuals without much prior knowledge can use it. As a machine learning library, it provides several techniques in both supervised and unsupervised learning. Also, to make the library easy-to-use, scikit-learn provides examples of implementations and datasets. It is important to provide standardized datasets, not only for showing implementation examples, but also to provide training data to create classifiers. Additionally, a machine learning library will not be a useful library without visualization. Therefore, there are visualization examples to illustrate the performance of a classifier by using matplotlib as a basis of visualization.

Diagram

Figure 3 describes the relationship between scikit-learn and its environment. It consists of ten external entity types which are related to scikit-learn. Each entity has a specific relationship to the library, for example users use scikit-learn or GitHub manages versioning and issue tracking for scikit-learn. Each specific entity inside an entity type may have a different weight of closeness to the library depending on their interactions, for example INRIA may have a stronger relation compared to Paris-Saclay Center for Data Science because they are still sponsoring scikit-learn at the time of writing.

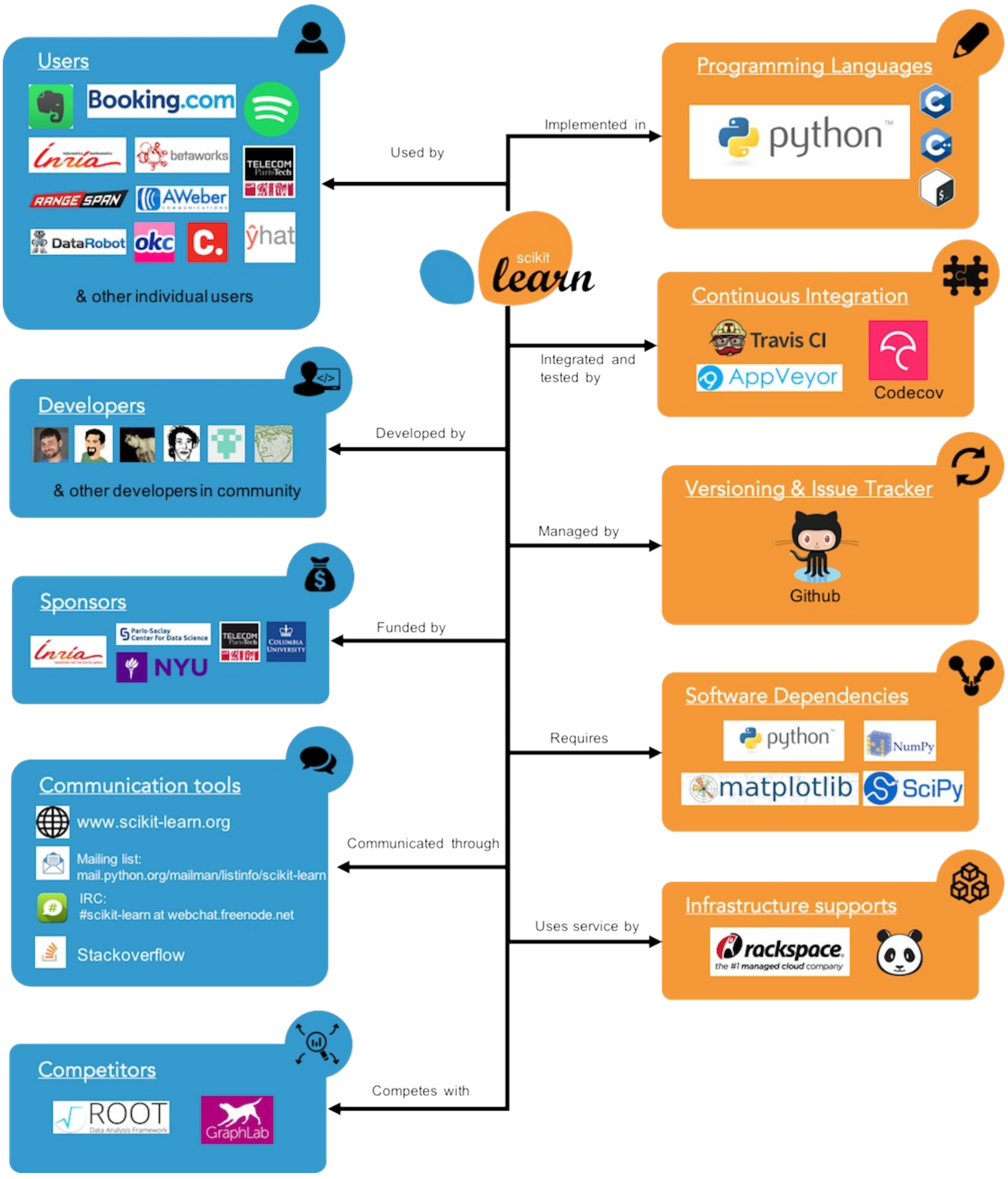


Figure 3: Context View.

Development View

The development view is what gives developers (and testers) a birds eye view of the architecture. It should not be too detailed or descriptive, but still cover the most important bases.
The development viewpoint discussed here is about scikit-learn's module structure model.

Module Structure Model

The module structure model defines the organization of the system’s source code and related external systems, in terms of the modules into which the individual source files are collected and the dependencies among these modules [2]. In Figure 4 layers are identified for scikit-learn with each layer consisting of one or more module(s). These layers are:

- Domain layer: consisting of all main functionality modules: data transformations [29], data loader [30], model selection [31], supervised learning [32], and unsupervised learning [33].
- Utility layer: consisting of modules that support basic functionality that can be used in domain layer, such as testing, validation, preprocessing, sparse tool, and external configuration.
- Platform layer, which contains modules of the required packages, such as python [34], NumPy [35], and SciPy [36].
- Build tool layer, which contains build modules [37] to build the library. Each module consists of files to download, install, testing, or setting the required library. Dependency of one layer to the other layer(s) is demonstrated by a dashed arrow which points to the destination of the required layer. As an example, the utility layer uses all libraries available from python, NumPy, SciPy, and Pandas by importing them in the module. In addition, there are explicit intermodule dependencies for all modules in the domain layer for python because all files under each module requires python.

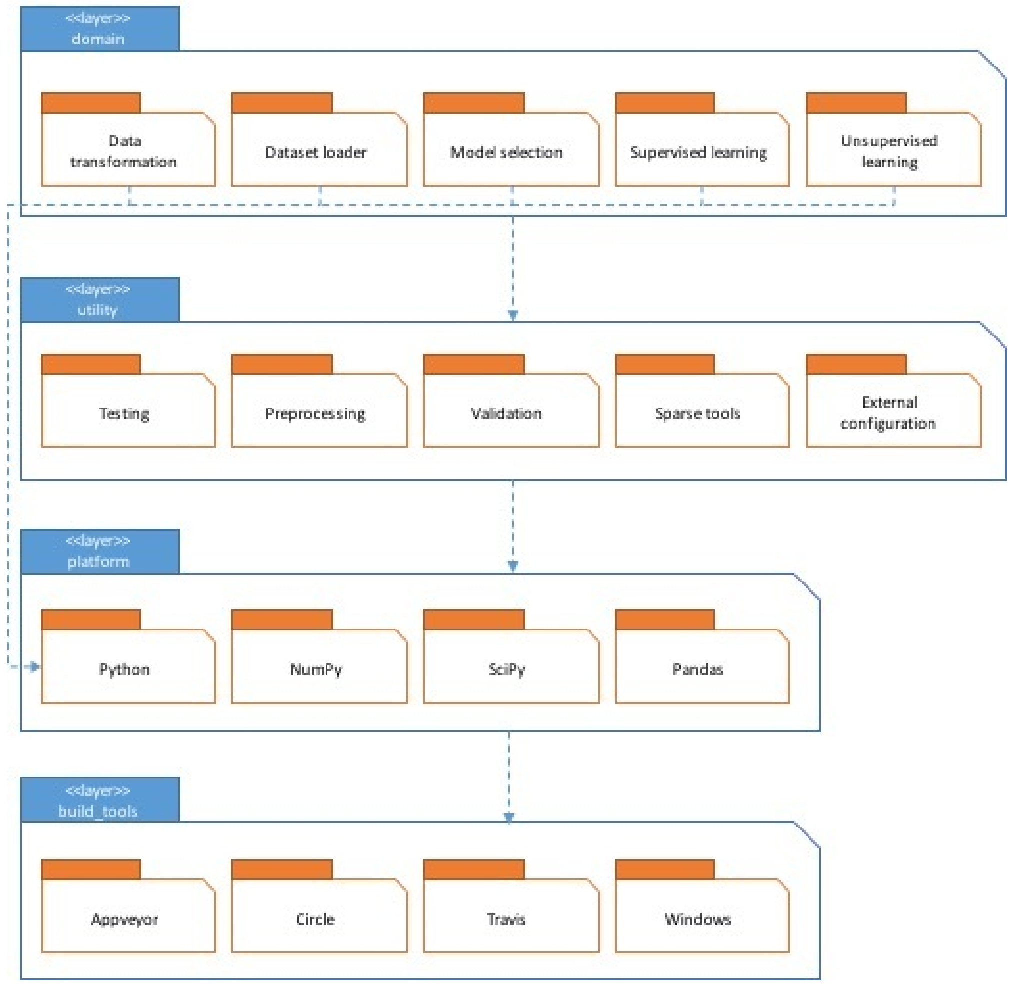


Figure 4: Modules Structure Model.

This model answers the first concern of Rozanski and Woods addressed by the development view [2] by giving a better understanding of the module organization. Each module consists of hundred, possibly thousands, of source files and even more lines of code, which are used to implement libraries or functional elements. As a software architect it is useful to know the generic view of a system before going too much into detail. By analyzing through this model, we understand better in which way scikit-learn has been managed and the depencies between modules are clearly highlighted. In this library, a module is usually representated by a folder in the sklearn directory[9].

Another good thing that can be inferred by this model is how to arrange code in a logical structure. Thereby, it will help to manage dependencies and cultivate a better understanding with developers of these interdependencies without affecting other modules in unexpected ways.

Deployment view

The deployment view is what looks into how the program is expected to operate in live operation. It will show what "hidden" dependencies scikit-learn has, it's runtime environment and lastly the required specialist knowledge for (parts of) scikit-learn.

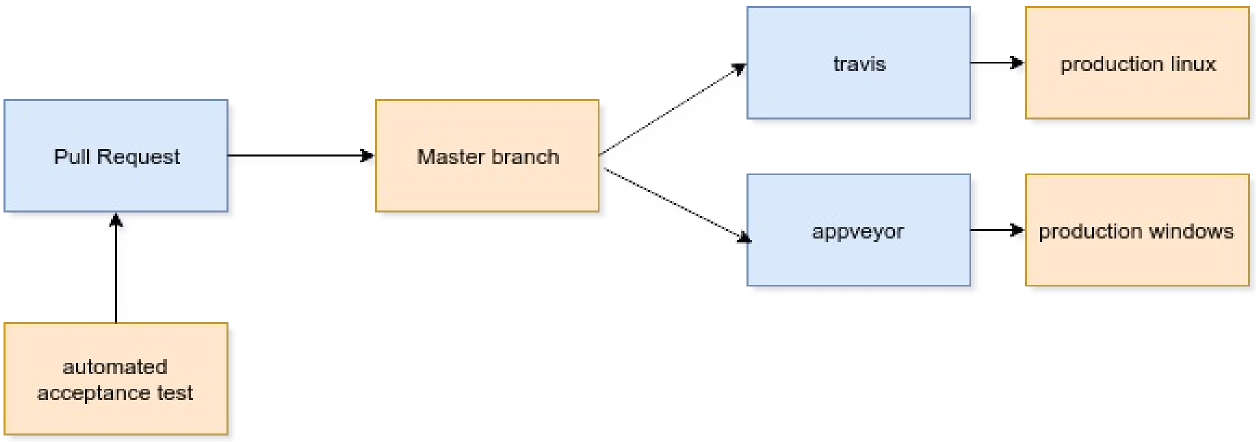


Figure 3A: the deployment view for scikit-learn

Dependencies

Running a scikit-learn instance requires several supporting libraries and programs [10]. These are, at the time of writing:

- Pip, while not a hard demand, in general is the way scikit-learn gets installed
- Conda, an alternative to pip to install scikit-learn
- Python, (> =2.6 or > =3.3) scikit-learn requires an instance of python to run it's scripts.
- NumPy (>=1.6.1)
- SciPy(>=0.9)

there are some technology compatibility conflicts between pip/conda and the required libraries. As the former tends to compile from source whereas the libraries tested are the provided binaries. Which can lead to differences.

Runtime environment

As scikit-learn runs entirely within a single system, and on this system it runs inside python. Most of its runtime environment is either highly simplified or defined by its hosting programs.

- A computing system (computer/phone etc.) with support for python (<1GB hdd space and a supporting processor architecture, preferably a bit of ram as well (say 128MB+))
- the prerequisites mentioned in dependencies

These limits are, however, rather unrealistic when using scikit-learn. As in most cases, scikit-learn is used on rather large datasets, which would increase RAM/HDD usage accordingly.

Specialist knowledge

To use Scikit-learn('s full potential) a lot of specialist knowledge is required. These required types of specialists are people a major company may want to have before adopting scikit-learn.

- Linear algebra, practically a demand for using scikit-learn as it is used almost everywhere that constitutes actual functionality. In addition, it is often a prerequisite for the other knowledge areas.
- Machine learning, scikit-learn allows use of neural network techniques, k-means and other techniques to provide most of its functionality.
- Set/collection theory, operations on and properties of sets are (implicitly) used as the basis for certain operations (such as biclustering, mean_shift and kmeans)

Perspective

Computational performance perspective

One of important perspective of implementing machine learning library is computational performance. There are two computational performance metrics that can be used: latency and throughput at prediction time. Optimization is usually done to minimize latency and maximize throughput but it can hurt prediction accuracy.

Prediction latency

Prediction latency is measured as the elapsed time necessary to make a prediction (e.g. in micro-seconds) [38]. There are four main factors that influence the prediction latecy: number of features, input data representation and sparsity, model complexity, feature extraction. Number of features affects memory consumption, which shows number of basic operations, such as multiplications for vector-matrix products. Matrix of M instances with N features will result O(NxM) space complexity. In sparse input data representation, optimization using sparse format is essential to make performance better by not storing zeros which will lead to less memory consumption. As a rule of thumb you can consider that if the sparsity ratio is greater than 90% you can probably benefit from sparse formats [38]. Model complexity will lead to more predictive power and latency. Increasing predictive power is usually interesting, but for many applications we would better not increase prediction latency too much [38]. In many real world applications the feature extraction process (i.e. turning raw data like database rows or network packets into numpy arrays) governs the overall prediction time

[38]. In many cases it is thus recommended to carefully time and profile your feature extraction code as it may be a good place to start optimizing when your overall latency is too slow for your application.

Prediction throughput

Prediction throughput is defined as the number of predictions the software can deliver in a given amount of time (e.g. in predictions per second) [38]. There are several ways to improve prediction throughput, such as spawn additional instances that share same model or add more machines to spread the load.

Software Debts

There is an increasing amount of danger of not being able to add or enhance features as the project evolves. Technical debt needs to be looked at from the start of the project because small issues can lead to bigger problems later in the development of the project.

Technical debt

Although, there is extensive maintenance from the beginning of the project, technical debt is still evident in the project as we are going to point out in this section. Several aspects regarding technical debt will be covered.

Solid principles

For identifying technical debt, a good method is to look into the SOLID principles and how these are handled for scikit-learn. These principles could give an indication of arguably bad design being used, preventing or making it more difficult to make changes in the future.

Single Responsibility Principle

The Single Responsibility Principle states that a class should only have a single responsibility [27].

Whether this principle is broken in a widespread manner is difficult to say for scikit-learn due to the size of its code base. In general, all classes such as neural networks [51], kdtree [53] and label propagation [20] model a certain specific theory, and therefore tend to model this in a rather direct way.

This is a fine way to separate concerns, given one important assumption: The theory either does not change or a change in theory fundamentally requires a new class. For instance, a new theory is not an update of the old, therefore requires a new class.

```
class email:
    function setSender(email_as_string)
```

Figure 5: A bad implementation of single responsibility principle.

```
class email_address:
   function getAddress():
       return this.address

class email:
    function setSender(email_as_email_address):
        this.address = email_as_email_addres.getAddress()
```

Figure 6: A proper separation of concerns, this allows for example checking of format of an email in the constructor of the email_address class.

This is also mostly true for places where modularization breaks down or other problems exist, as will be addressed further in this chapter, such as utils. the mocking file [19] is responsible for mocking functionality, and arrayfuncs [18] provides functions for arrays.

There are probably still splits that are possible, but the division of responsibilities makes a lot of sense for what scikit-learn is trying to do.

Open/Closed Principle

The Open/Closed Principle states that software entities should be open for extension, but closed for modification.

```
class drawer:
    drawCircle():
    drawSquare():
    drawShape(x):
        if(isinstance(x, circle):
            drawCircle()
        else:
            drawSquare()

class shape:

class circle(shape):

class square(shape):
```

Figure 7: Example of bad application of the open/closed principle, drawer needs knowledge about every extending class. requiring changes for each new class implementing shape.

```
class drawer:
    drawShape(x):
        x.draw()

class shape:
    draw(): # python does not have interfaces, but duck-typing.

class circle(shape):
    draw():

class square(shape):
    draw():
```

Figure *8: A good implementation of the open/closed principle, drawer can simply take any new implementation of shape as well.*

For implementations that have similar roots (such as neural networks [15]) a parent class will be made defining basic behaviour that does not depend on specifics of the child to be useful. This closes the parent class, but keeps the child class open for extension

Liskov Substitution Principle

The Liskov Substitution Principle states that if S is a subtype of T then objects of type T may be replaced with objects of type S

```
class rectangle:
    setHeight(self,x):
        self.height = x

    setWidth(self, x):
        self.width = x

    getArea(self):
        return self.width*self.height

class square(rectangle):
    setHeight(self,x):
        self.height = x
        self.width = x

    setWidth(self,x):
        self.height = x
        self.width = x

s = square()
s.setWidth(10)
s.setHeight(5)
print(s.getArea) # no valid expectation on area (either 25 from square, or 50 from rec
tangle)
```

Figure 9: Bad implementation of Liskov substitution principle, square has additional restrictions on it that rectangle does not have, leading to weird situations when functions for rectangles get called on it.

This principle does not seem to be violated in scikit-learn. Lower classes do not tend to introduce additional restrictions and can replace their parent class where necessary. This is also because scikit-learn tends to implement proper duck-typing, which -when done properly- ensures the Liskov Substitution Principle.

Duck-typing says that the suitability of a certain class S to replace T is that it should offer at least the same functionality as T. i.e. "If it walks like a duck and quacks like a duck, it must be a duck." [16].

Interface Segregation principle

The Interface Segregation Principle states that no client should be forced to depend on methods it does not use.

This is again true because scikit-learn applies good duck typing. A duck does not talk, therefore its parent classes will not demand this of it.

This could also be explained as being caused by using python, as it does not support interfaces, fundamentally ignoring this principle. And therefore any program written in python is incapable of applying it [28].

Dependency Inversion Principle

Dependency Inversion Principle states that high level modules should not depend on low level modules, but instead both should depend on abstractions. Furthermore, abstractions should not depend on details but details on abstractions.

This does happen on occasion inside scikit-learn for example multilayer perceptron[17] creates a LabelBinarizer, meaning it is now dependent on how LabelBinarizer works for its own behaviour. Instead of passing a LabelBinarizer through the constructor.

```
if not incremental:
            self._label_binarizer = LabelBinarizer()
            self._label_binarizer.fit(y)
            self.classes_ = self._label_binarizer.classes_
```

Figure 10: Example of breaking Dependency Inversion Principle.

Reducing these kinds of instances could reduce the amount of technical debt, because it would be easier, and more explicit, when the class is passed through the constructor to change the specific class delivering the functionality in case a NewBetterLabelBinarizer class is required.

Debt From Structure

Scikit-learn has several different ways of formatting its code. It uses both the classes and their respective imports [17].

```
# filename: duck.py
class duck
    def quack():
        return 'Quack!'
```

Figure 11: Example of new importing.

```
from duck import duck
whatDoesTheDuckSay = duck().quack()
```

Figure 12: Another example of new importing.

and defs with file based imports

```
# filename: duck.py
def quack:
    return 'Quack!'
```

Figure 13: Example of old importing.

```
import duck
duck.quack()
```

Figure 14: Another example of old importing.

This means different files function differently for the same type of functionality, possibly restricting flexibility and this could therefore be seen as technical debt.

Debt From Modularization

For the most part scikit-learn has a logical document structure. An important exception to this is the utils folder, which seems to contain a lot of very different modules such as math [21], testing [22] and array functions [23]. This might make it more difficult to find specific functionality, increasing technical debt.

Discussion of Technical Debt - In the Source Code

This section will look at how the contributors of Scikit-learn discuss technical debt. In some projects (maybe even most) occasionally technical debt is being discussed in the source code itself. This is often in the form of *TODO* or *FIXME* comments. In general, we find this a bad way of discussing technical debt as it is often forgotten about. However, in Scikit-learn it still happens occasionally.

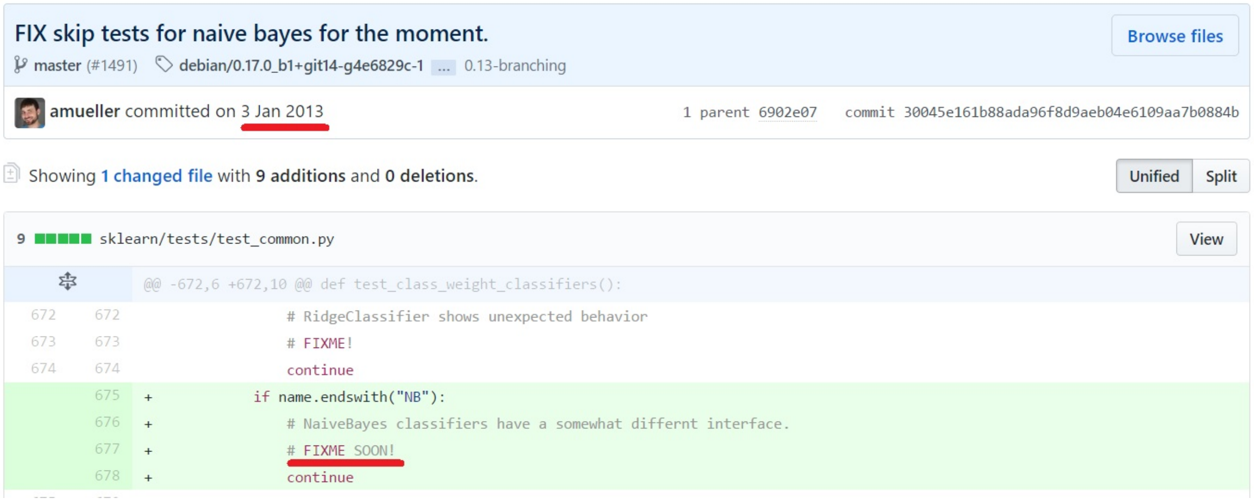
We used grep to look for any *TODO* or *FIXME* comments in the whole Scikit-learn repository (this includes the documentation). The following commands were used in the root directory of the scikit-learn repository:

> grep -r "FIXME" . --ignore-case
> grep -r "TODO" . --ignore-case

Figure 15: Grep commands.

We found 76 occurrences of *TODO* comments, spread out over 48 files. Four of these files are documentation files, but these have two purposes. Sometimes a *TODO* is added in the documentation to indicate that a feature for example should still be implemented in the code, other times it actually means that something is missing in the documentation. We found 20 occurrences of *FIXME* comments, spread out over 16 files, 2 of which are documentation files and 4 test files. The fact that there are both *FIXME* and *TODO* comments in the testing and documentation code indicates some form of testing and documentation debt present in the project.

The following picture sums up how well this method works for discussing technical debt:



FIX skip tests for naive bayes for the moment.

master (#1491) debian/0.17.0_b1+git14-g4e6829c-1 ... 0.13-branching

amueller committed on 3 Jan 2013 1 parent 6902e07 commit 30045e161b88ada96f8d9aeb04e6109aa7b0884b

Showing 1 changed file with 9 additions and 0 deletions.

9 sklearn/tests/test_common.py

```
@@ -672,6 +672,10 @@ def test_class_weight_classifiers():
                  # RidgeClassifier shows unexpected behavior
                  # FIXME!
                  continue
+             if name.endswith("NB"):
+                 # NaiveBayes classifiers have a somewhat differnt interface.
+                 # FIXME SOON!
+                 continue
```

Figure 16: "FIXME" comment added long ago, which is still present in the code now.

Testing debt

Code Testing

The code coverage for scikit-learn on 16 March was 94.76% which is very good. We can see a breakdown of the code coverage for different modules in the barplot in Figure 17. The full information can be found here [24]. Despite having modules where the code coverage is low(e.g. datasets) overall the system is tested very well.

The major contributors of scikit-learn have agreed upon approximately 90% coverage. Also, the project is well tested by implementing unit-testing. Unit testing as the developers in scikit-learn mention is "a corner-stone of the scikit-learn development process" [2].

We have also made a barplot where it is easier to see the different coverage scores for the different modules. We can identify the big difference of code coverage between the datasets module and the other modules.

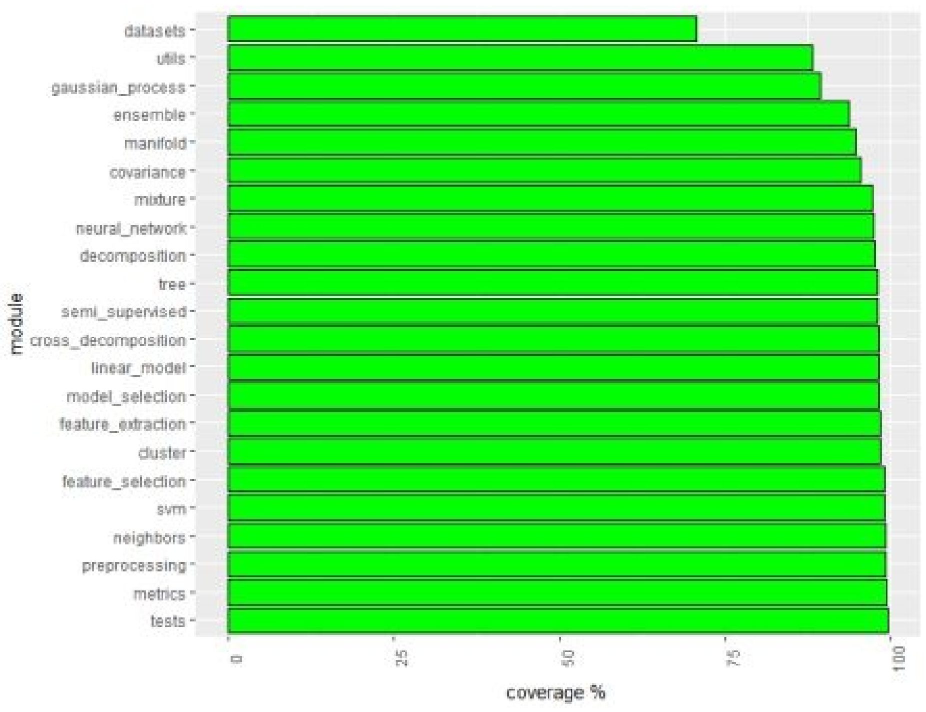
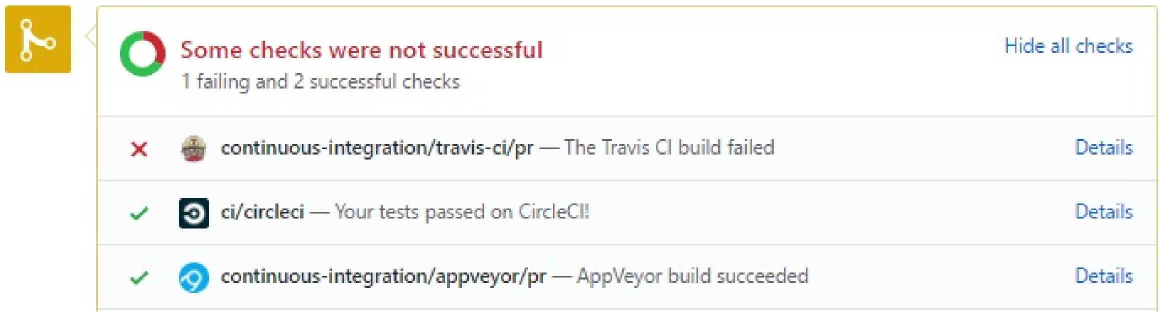


Figure 17: Code coverage about different modules ordered.

In addition, the project of scikit-learn relies on integration testing services like Travis CI [14], circleci [12] and AppVeyor [13]. It is a common procedure of the scikit-learn testing procedure to report the results of tests on continuous integration (CI) platforms. We can see an example of successful and unsuccessful testing on a PR in the figure below. All PRs need to pass all five check tests from different CI platforms before it gets merged: ci/circleci, codecov/patch, codecov/project, continuous-integration/appveyor/pr, and continuous-integration/travis-ci/pr. Thus, all changes in scikit-learn are well-tested and it reduces the risk. Also, it helps the reviewers to make decisions about which PR should be merged and to give constructive advice to the authors in order to fix PR.



Some checks were not successful
1 failing and 2 successful checks
Hide all checks

✗ continuous-integration/travis-ci/pr — The Travis CI build failed Details
✓ ci/circleci — Your tests passed on CircleCI! Details
✓ continuous-integration/appveyor/pr — AppVeyor build succeeded Details

Figure 18: Tests on different CI platforms.

Testing results on CI platforms.

As is mentioned above, unit testing is also performed with the nose package [15]. The tests are functions with appropriate names, located in tests subdirectories. The tests check the validity of the algorithms and the different options of the code. The full scikit-learn tests can be run using 'make' in the root folder. Alternatively, running 'nosetests' in a folder will run all the tests of the corresponding subpackages. The code coverage of new features are expected to be at least around 90%.

Proposal for Removing Debt

It is dangerous to have this form of technical debt in a place where contributors might be less aware of it. To remove this form of technical debt we would firstly propose to keep track of it in a better way, so that the advantage of being an open source project can be utilized better to get rid of it. In issue 8581 [11] we proposed the following work-flow to remove it:

1. Create an issue on GitHub for each file still containing TODOs/FIXMEs.
2. File by file create issues for each TODO/FIXME comment.
3. Either remove the comments in a new PR linking the newly created issues or if it is preferable to keep the comments in the code as well, instead of removing the comment add a link to the issue in the comment.

This issue attracted the attention of a couple developers in scikit-learn. They highlighted that this is tedious work and it is more preferable to focus on existing issues. Also, these TODO/FIXME comments require expertise not currently available in the project.

Conclusion

In the beginning of the chapter the different stakeholders of scikit-learn were introduced. Initially, the categories as defined by Rozanski & Woods were used, but using four main categories of stakeholders were more applicable to scikit-learn, which are contributors, users, funders and competitors. This section was followed by the views sections where three different aspects of the architecture were discussed through the use of the context view, the Development View and Deployment View. The Development View consisted of the Module Structure Model. The Module Structure Model identified the structure of the code in terms of how the code was grouped into modules. Finally, the Deployment View looked at the system in operation. It identified dependencies required to run and install scikit-learn, the environment needed to run it in and it also identified the need for specialist knowledge to utilize scikit-learn to it's fullest. The perspective on computational performance is the additional section to explore more about performance of scikit-learn as machine learning library. The last section looked at two main aspects of software debt, technical and testing debt. It started off with a section about looking at technical debt through the SOLID principles. This concluded that the SOLID principles are not fully applicable to this project due to using Python. In this section, we also found that one prominent form of technical debt were *TODO* and *FIXME* comments. To mitigate this debt, we proposed a work-flow for adding these issues to GitHub one by one so that they could be tracked and solved more easily. Through our interactions with the scikit-learn contributors, we learned that they struggle with having enough expert knowledge to deal with these issues. When looking at the testing debt, it was evident that scikit-learn has high test coverage (94.76%) and that the contributors put in a lot of effort to keep this coverage high when reviewing pull requests. High test coverage really is a corner-stone of their development process. This is also noticeable when submitting a pull request since various continuous integration tools need to pass before a pull request can be approved.

References

[1] http://scikit-learn.org/stable/about.html#people

[2] Rozanski, _Nick, and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives._ Upper Saddle River, N.J.: Addison-Wesley, ©2012.

[3] http://scikit-learn.org/stable/developers/contributing.html

[4] http://stackoverflow.com/tags/scikit-learn/topusers

[5] https://github.com/orgs/scikit-learn/people

[6] https://github.com/scikit-learn/scikit-learn/pull/8253

[7] https://github.com/scikit-learn/scikit-learn/pull/8280

[8] https://github.com/scikit-learn/scikit-learn/blob/master/CONTRIBUTING.md#user-content-pull-request-checklist

[9] https://github.com/scikit-learn/scikit-learn/tree/master/sklearn

[10] http://scikit-learn.org/stable/install.html

[11] https://github.com/scikit-learn/scikit-learn/issues/8581

[12] https://circleci.com/

[13] https://www.appveyor.com/

[14] https://travis-ci.org/

[15] http://nose.readthedocs.io/en/latest/

[16] http://www.dictionary.com/browse/duck-typing

[17] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/neural_network/multilayer_perceptron.py

[18] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/extmath.py

[19] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/neighbors/kd_tree.pyx

[20] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/semi_supervised/label_propagation.py

[21] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/extmath.py

[22] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/testing.py

[23] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/arrayfuncs.pyx

[24] https://codecov.io/gh/scikit-learn/scikit-learn/tree/603ff1a61d2d3d08624e18b32d05c177711d7299

[25] https://turi.com

[26] https://root.cern.ch

[27] [http://www.oodesign.com/single-responsibility-principle.html

[28] [https://www.python.org/dev/peps/pep-0245/

[29] https://github.com/scikit-learn/scikit-learn/blob/master/doc/data_transforms.rst

[30] https://github.com/scikit-learn/scikit-learn/tree/master/doc/datasets

[31] https://github.com/scikit-learn/scikit-learn/blob/master/doc/model_selection.rst

[32] https://github.com/scikit-learn/scikit-learn/blob/master/doc/supervised_learning.rst

[33] https://github.com/scikit-learn/scikit-learn/blob/master/doc/unsupervised_learning.rst

[34] https://www.python.org

[35] http://www.numpy.org

[36] https://www.scipy.org

[37] https://github.com/scikit-learn/scikit-learn/tree/master/build_tools

[38] http://scikit-learn.org/stable/modules/computational_performance.html

Scrapy

Joren Hammudoglu (@jorenham), Johan Jonasson (@jojona), Marnix de Graaf (@Eauwzeauw)

Delft University of Technology, 2017

Abstract

Scrapy is an application framework for crawling websites and extracting structured data which can be used for a wide range of web applications, like data mining, information processing or historical archival. The chapter will start by analysing the Scrapy framework from different perspectives. First, the context view which describes the environment around Scrapy such as dependencies and responsibilities. Then the development view with the architectural concerns related to Scrapy. The chapter will also describe the operational viewpoint, performance and scalability. The chapter will end with analysing the technical debt related to the code and the evolution of the technical debt.

Table of contents

1 Introduction

Scrapy is an open-source framework for crawling websites and extracting structured data from them. Crawling websites concerns the cycle of downloading a webpage, following its links and downloading those pages. It can be used for a wide range of web applications, like data mining, information processing or historical archival. The general use-case is to extract data from a webpage, which does not provide a public API for its data. Core functions include, but are not limited to, fetching webpages (crawling), processing them (scraping) and saving the extracted data (exporting). While Scrapy can do a lot out of the box, it is also designed so one can easily extend existing functionality.

The project architecture is built around the use of ‘spiders’. The spiders are self-contained crawlers which are given a set of instructions on how to crawl pages and which data to extract from them. Spiders are implemented to crawl fast and concurrently while also being fault-tolerant and polite to crawled servers. A general overview on how Scrapy works can be seen in Figure 1. Scrapy is written in Python and can be run on Linux, Mac OS and Windows.

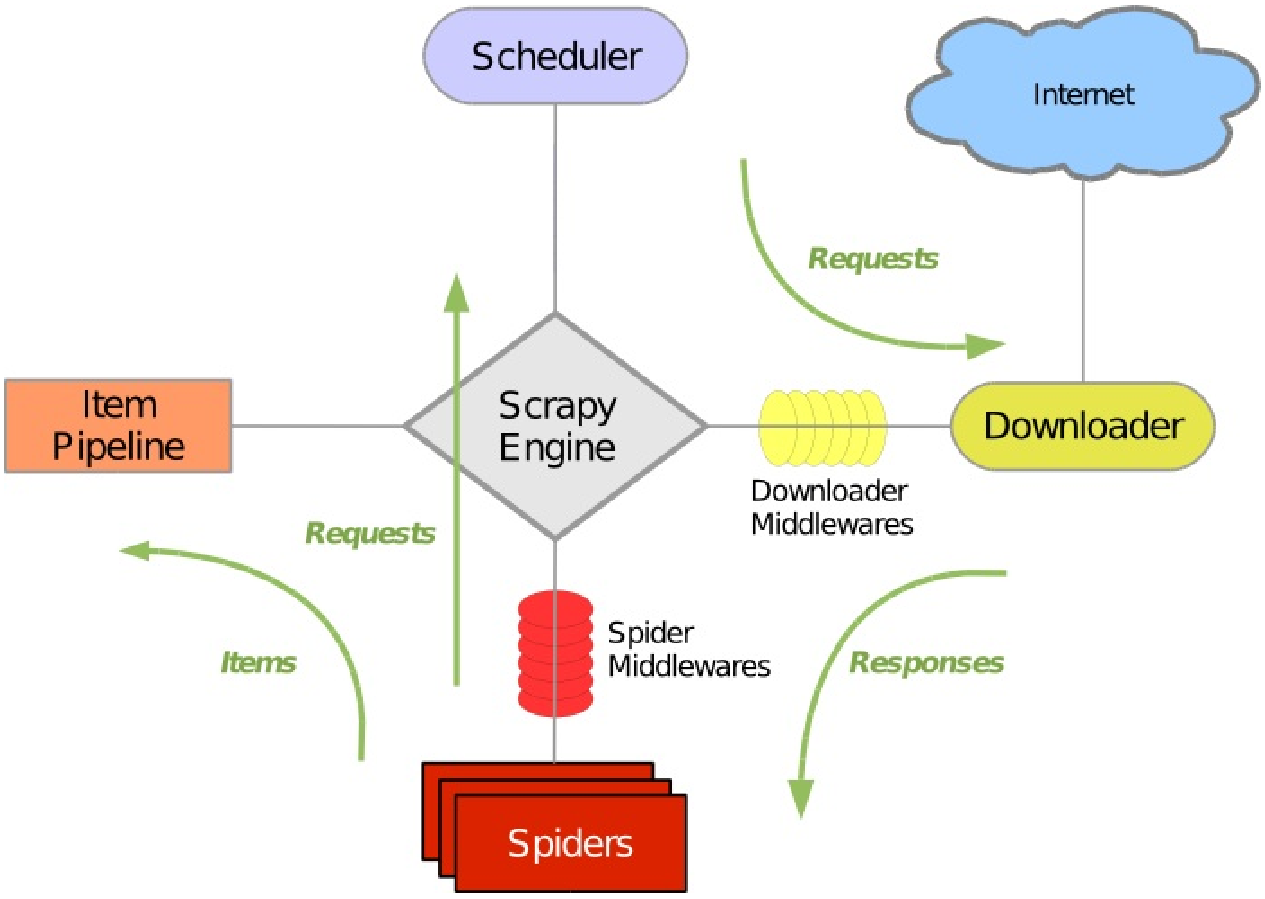


Figure 1: General overview of Scrapy's workings. Adaptation of image from documentation 13.

This chapter takes the reader through the workings and development of Scrapy, next to the possible problems in the project. It starts with the context of the project itself, describing the stakeholders, integrators, system responsibilities and dependencies. Next, more attention is paid to the code itself and we dive deeper into the structure of Scrapy. The development viewpoint includes code structure, modules and dependencies, configuration management and coding standards. This is continued by the operational viewpoint which describes how Scrapy is operated in the production environment by identifying concerns related to control, managing and monitoring the system. After this, Scrapy is analysed from a performance and scalability perspective followed by a description of the technical debt build-up over the years.

2 Stakeholders

Several stakeholders are present for widely-used projects like Scrapy. Chapter 9 of Rozanski and Woods[1] defines a Stakeholder as follows:

> A stakeholder in the architecture of a system is an individual, team, organization, or classes thereof, having an interest in the realization of the system.

Every type of stakeholder has its own wishes, requirements and influence for the project. In the following sections the stakeholders of Scrapy will be described according to their type, class and role.

Types and classes of stakeholders

Several types and classes of stakeholders are categorised in Chapter 9 of Rozanski and Woods. To better understand the stakeholders, an analysis follows in Table 1 and Table 2 on how they apply to Scrapy.

Type	Description
End-users without influence	Users who use Scrapy without contributing to the development or decision-making process.
End-users with influence	Company employees or professionals using Scrapy in their business, who also contribute to the code and (architectural) decisions of Scrapy to help their own product work better. Examples of these companies are Scrapinghub (current maintainer), parse.ly and any other company found on the Scrapy companies page.
Specialists knowledge	Most active developers and founders with a lot of insight into the system, which again would be employees of Scrapinghub. Other external contributors with specific knowledge about some part of the system are also part of this type.

Table 1: Types of Stakeholders

Class	Description
Maintainers	Most active (and arguably the most important) class, who manages the evolution of the system once operational. For Scrapy this class consists of a few members described more in detail below and they are the ones accepting and reviewing pull-requests. Notable is that most of them have a connection to Scrapinghub.
Developers	Construct new functionality, write tests for that and give suggestions on how to build the system. For Scrapy this class consists of all the contributing members on GitHub.
Communicators	Explain the system to other stakeholders via documentation and training materials. In Scrapy these are the people who write the documentation and those that answer questions on forums, GitHub and StackOverflow. Without this active community it would be unclear for users how to use the framework for their unique project.
Testers	Tests the system to ensure that it is suitable for deployment. Scrapy does this by using Tox for (unit)testing. New features must be accompanied by test cases so that tests can easily be run before merging a new code contribution. This class in Scrapy contains the reviewers to the pull-requests on GitHub but also all the developers who are expected to test their code before they make a pull-request.
Users	Ultimately makes use of Scrapy and provide requests for new functionality and provide bug reports.
Assessors	Oversees conformance to standards and legal regulation. This class consists mostly of the reviewers and thus overlaps with the Maintainers class.
Acquirers	Oversees the procurement of the product. As Scrapy is free to use and uses open-source free modules, no entities can be found belonging to this class.

Table 2: Classes of Stakeholders

To get more insight into the Scrapy framework and development, an analysis of issues and pull-requests was made. The results were that the most active commenters and reviewers are Kmike and Redapple who are employees of Scrapinghub. A few other actively contributing members were also present by creating issues, pull-requests and comments. Most of the issues and pull-requests are related to fixing existing code. There are also issues related to fixing technical debt and implementing new features.

One example of an interesting issue is #2568. The issue is a discussion about removing old deprecated code versus keeping it for backward compatibility. As discussed more in the technical debt section, Scrapy tends to keep outdated code for a long time. From looking at the pull-requests it is found that Scrapy often uses one reviewer and an automatic Codecov report. A final overview of the stakeholders is displayed in Figure 2.

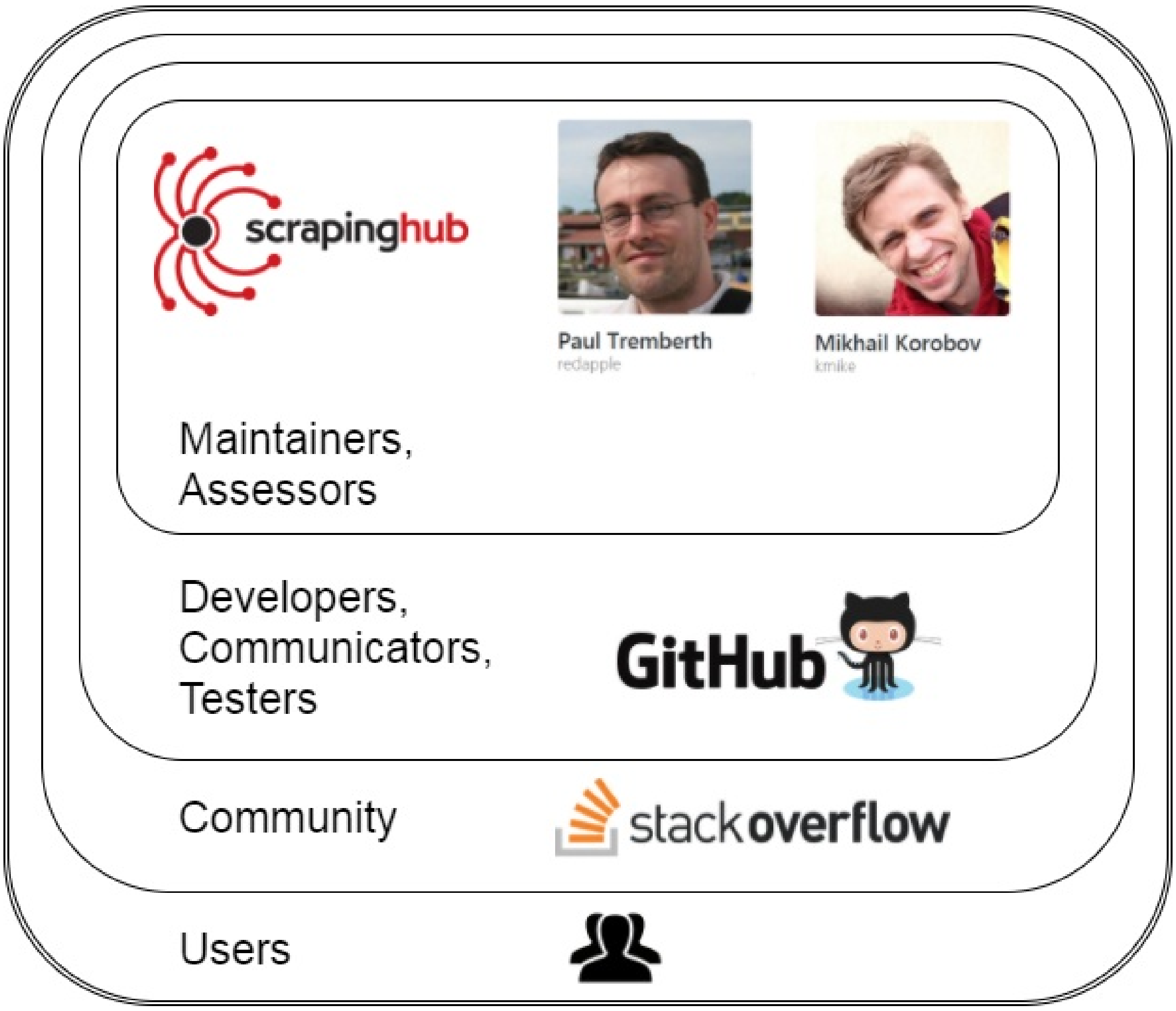


Figure 2: Stakeholders overview

Integrators

From the analysis the integrators @Kmike, @Redapple and @Dangra can be identified, who are active in reviewing and accepting pull-requests. The founders @Pablo Hoffman and Shane Evans also greatly contributed to the project. Pablo Hoffman is still the biggest overall contributor to the Scrapy GitHub but is not active anymore. All of the above is part of the Scrapinghub organisation on GitHub.

The method for contributing new patches to Scrapy can be found on the ReadTheDocs of Scrapy. They encourage small patches, require all tests to pass and that new tests cases are created for new features. If the patch changes the API, the documentation changes should be added together with the feature.

From the analysis on the pull-requests and issues it is found that pull-requests are accepted after one member has reviewed it. All pull-requests are checked with an automatically executed Codecov report. Pull-requests are used for code reviews and resolving GitHub issues. Discussion on features or bugs is done in issues and there is almost always an issue referenced in a pull-request. Most of the pull-requests seem to be corrective and fixes errors or other issues.

3 Context view

In this section we will assume Scrapy to be a black box. This will illustrate the project in its context, while not looking at the internal workings. We will focus on which environmental factors Scrapy depends on, and what influence Scrapy has on its environment. This formalisation of context is described by Rozanski and Woods, Chapter 16 [1].

External entities, services and data

In this subsection we will peek into the black box and look at how it handles its dependencies and what it does with other external entities and services.

Internal systems of the Scrapy project include the documentation website, the user interaction channels such as Twitter, an email list and an IRC channel. The original development environment for Scrapy is Python 2 and in the past years Scrapy has gained Python 3 support. This increases the complexity because there have been breaking API changes in Python 3 with respect to Python 2. Aside from this, Scrapy has many dependencies on external libraries, which are listed in Table 3.

Dependency	Purpose	Type
lxml	Efficient XML and HTML parser	Parsing
parsel	HTML/XML data extraction library written on top of lxml	Parsing
w3lib	Multi-purpose helper for dealing with URLs and web page encodings	Parsing
cssselect	Parse CSS(3) to XPath1.0 expressions, maintained by the Scrapy devs themselves	Parsing
Twisted	Asynchronous networking framework, which the entire Scrapy codebase is built on top of. This framework also allows for unit-testing	Networking and Security
cryptography and pyOpenSSL	Deal with various network-level security needs	Networking, Security and Testing
pyOpenSSL	Python wrapper around OpenSSL	Networking and Security
service_identity	Checking the validity of certificates.	Networking and Security
PyDispatcher	Process signals with multiple consumers and producers	Helper libraries
queuelib	Collection of persistent (disk-based) queues, originally part of Scrapy	Helper libraries
six	Python 2 and 3 compatibility	Helper libraries
Tox	Tests Scrapy on Python versions 2.7, 3.3 - 3.7 and Pypy	Testing
Coverage	Measurement tool for code coverage of tests	Testing

Table 3: Scrapy's dependencies

Scrapy's source code and documentation are maintained on GitHub. Here, stakeholders also discuss bugs, fixes and propose new features. The code is available for use and modification, although the license needs to be included and the name Scrapy cannot be freely used for promotion.

Scrapy is (unit)tested using the Tox and coverage libraries, described in more detail below. While Scrapy is written in Python and theoretically executable on any platform that runs Python, there are some platform-specific considerations. An example of this is #2561. This pull-request makes it evident that running Scrapy requires a different setup and dependencies on an operating system other than Linux.

There are many web crawling frameworks, and defining competition is a matter of choosing how wide to cast your net. Scrapy is the most complete and mature web crawling framework in Python, but in other languages and for other scales there are many other frameworks [2]. Scrapy is intended for a scale of crawling from one site up to a couple of thousand, but not intended for example as a broad web crawler to use as a basis for a search engine, such as those employed by Google, Bing etc.

The users of Scrapy consist of individuals and companies. Some of these companies are listed on the Scrapy website.

When a pull-request is submitted to the Scrapy repository, automated services check the quality of the submitted code. The tests are run using the Travis continuous testing service. If tests fail in any of the enabled Python versions, a red cross will appear in the pull-request, indicating that the proposed code change should be fixed. For measuring the number of code lines that are covered by the tests, Codecov is used. This service is publically accessible and displays exactly which lines are tested and the percentage of those lines. This percentage is the ratio of test-covered lines and the total amount of lines. It also offers a very useful sunburst diagram of the coverage distribution of the code structure.

An overview of the relations Scrapy has with external entities is depicted in Figure 3

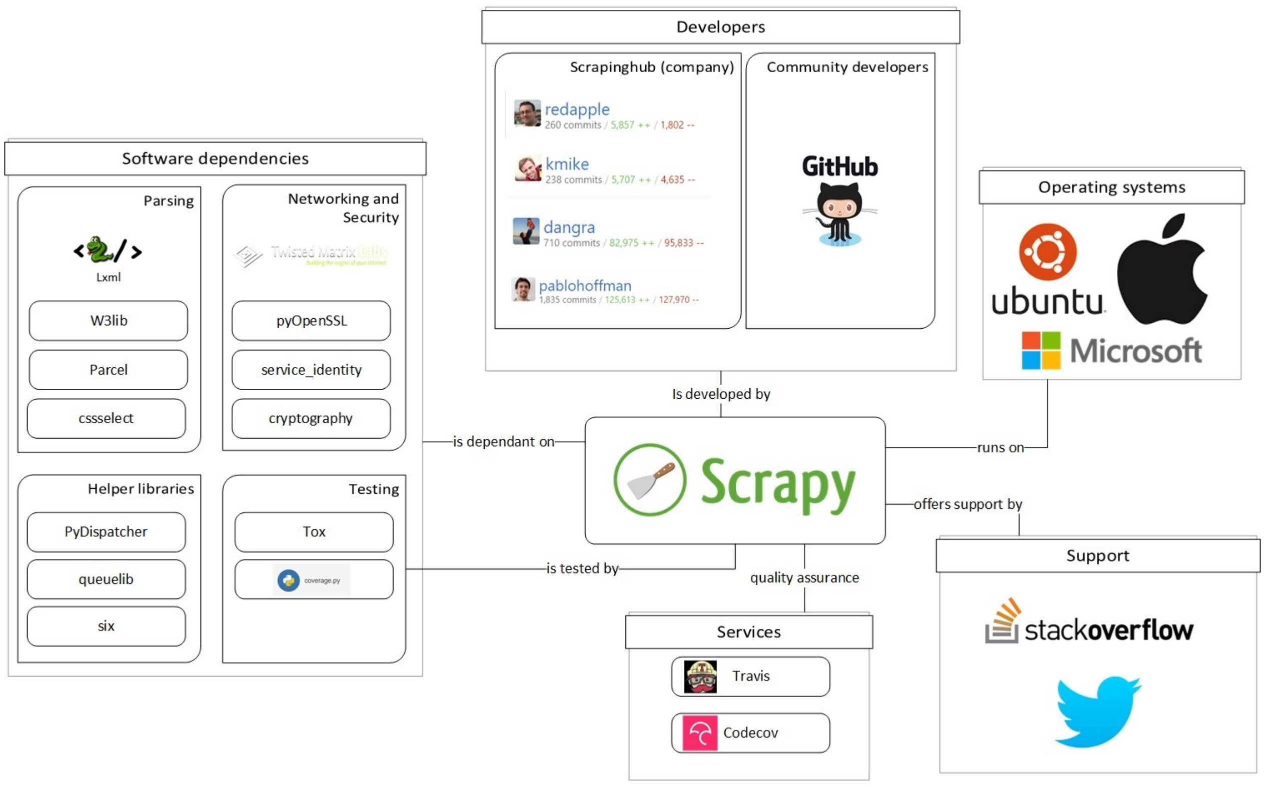


Figure 3: External entities

Environmental impact

Web crawling can make a lot of requests to the web. Crawling a single webpage with many requests can result in a DDoS effect on the website and make it unavailable or slow. Crawling multiple webpages requires many DNS lookups to the crawlers DNS server.

A result could be a slow network and a response from web servers could be an IP block. These problems can be avoided by limiting the amount of requests to a server within a time limit and using a local DNS server.

4 Development viewpoint

The development view focuses on the architecturally significant concerns, as described by Rozanski and Woods, Chapter 20 [1]. In this section we will describe how the code is structured, how the modules and their dependencies are organised, the way different configurations are managed and the standardisation of design and testing. Additionally, we will investigate the planning and design of the software development environment which is used to support the development of the system.

Codeline model

This subsection will describe how source code is organised, tested and managed.

Configuration management

Scrapy uses GitHub for version and configuration management. Different versions are maintained using tags, different configurations with branches. The latter are generally used for big experimental features or code rewrites, where multiple developers contribute to. Once they are sufficiently tested and approved by the maintainers, they can be merged to the master branch. An example branch is asyncio. Although not maintained anymore, it illustrates the joint work on the support of a fundamental architectural feature. In contrast, small changes to the current configuration (e.g. a bugfix or feature) can be pull-requested directly to the master branch.

Release process

A release in Scrapy is defined by a release number with 3 numbers, A.B.C. A is the major release number where changes are rare and are meant for large changes. Scrapy is still in version 1.X and A has only changed from 0 to 1 when Scrapy was defined to be production ready and with a stable API. B is the release number for new features and updates which might break backward compatibility. C is only for bugfix updates.

Every new Scrapy release is accompanied with release notes explaining new features, bug fixes, documentation changes and dependency changes.

Releases are handled on GitHub using the release system and tags. The latest release 1.2.2 was released on dec 6 2016 and there are currently tags for 1.2.3 and 1.3.2 on GitHub.

Documentation updates are required to be created by the developer with the pull-request. Other documentation changes are handled with the same process as a code update with pull-request and issues.

Test build and integration approach

When a contributor adds a new feature or fixes a bug, it is required to write tests as specified in the documentation. They should cover the added or changed lines and they should pass.

The source code is automatically tested with unit-tests using the Twisted unit-testing framework [3] in combination with Tox. Due to the multiple Python versions (2.7 and 3.3 - 3.6) that are supported, the tests are run on each of these versions. To ensure that all the tests pass on each environment before pushing code to the master branch, Travis is used.

Analogously, coverage.py and Codecov ensure that the number of tested code lines is sufficiently high. The dropping of coverage for a new feature could indicate that it is poorly tested, which is why this is automatically displayed in the corresponding pull-request.

Code structure

We will discuss the structure of the code by describing the folders inside the Scrapy master branch for version 1.3.2.

The artwork directory contains the Scrapy logo and fonts used for that logo. In the debian directory are several files responsible for packaging Scrapy into a Linux executable. The docs folder contain the markdown sources for the offical Scrapy documentation. Miscellaneous scripts for i.e. bash completion and coverage report can be found in the extras folder. Scrapy Enhancement Proposals (SEP) are placed in the sep directory. These markdown files are mostly old feature proposals. Automated tests are placed in the aptly named tests folder. Inside, there are a few folders for keys, sample data and with test cases for a few modules. The main source code is all located in the Scrapy folder. Most modules are placed into subdirectories which are listed below. All of the smaller modules are placed directly into the `scrapy` folder. There are also old files from moved modules. For example `spider.py` , the spider code is now in a directory but the spider.py still exists in the top folder with deprecated warning code.

Inside of the source code directory, there are three deprecated modules: contrib, contrib_exp and xlib. They are still there for backward compatibility. The contrib form `contrib` is now placed in the extensions module.

The core module is for several essential sub-modules; the Scrapy engine, scheduler, scraper, spider middleware and downloader. Another integral part of Scrapy are the spiders.

Creating a Scrapy project requires one to implement a spider to define which webpages should be crawled and what data should be extracted, as explained in the documentation. The data extraction is done by selecting a part of the HTML with either css or XPath expressions. The implementation for this is in the selector module (docs).

Extracting links is a very common use-case, which is why there is the linkextractors module (docs).

Crawling website's requires HTTP functionality, which is mostly found in the http module. The HTTP requests and responses can be processed for e.g. caching and compression purposes with the downloadmiddlewares module (docs). The spidermiddlewares also offers similar functionality but for the output and input of the spiders (docs).

The extracted data of the spiders is put in containers called items that are managed by the loader. To extract files and images from these items, a pipelines can be used. The pipelines implement features to avoid re-downloading files and outputting extracted data.

The contracts feature is used for testing spiders (docs). Scrapy can be controlled from the command-line with the the commands module (docs). However, we will see in the Testing Debt section, this directory is likely to be removed in the near future. The settings contains the default settings that can be overridden in a Scrapy project.

A big module is utils. It contains different utilities. For example, help to deprecate functions or helper functions for HTTP objects.

Module organisation

Large systems are often organised into modules to be able to group related code. Modules help with arranging dependencies and lets developers work with modules without affecting other modules and causing unexpected errors or changes to the system.

We have identified five module categories for Scrapy; the engine, downloader, spiders, item processing and user interface. The modules for each category and their high-level dependencies can be seen in Figure 4. The modules and their dependencies are listed in

Table 4 and Table 5. The inter-module dependencies have been omitted for clarity.

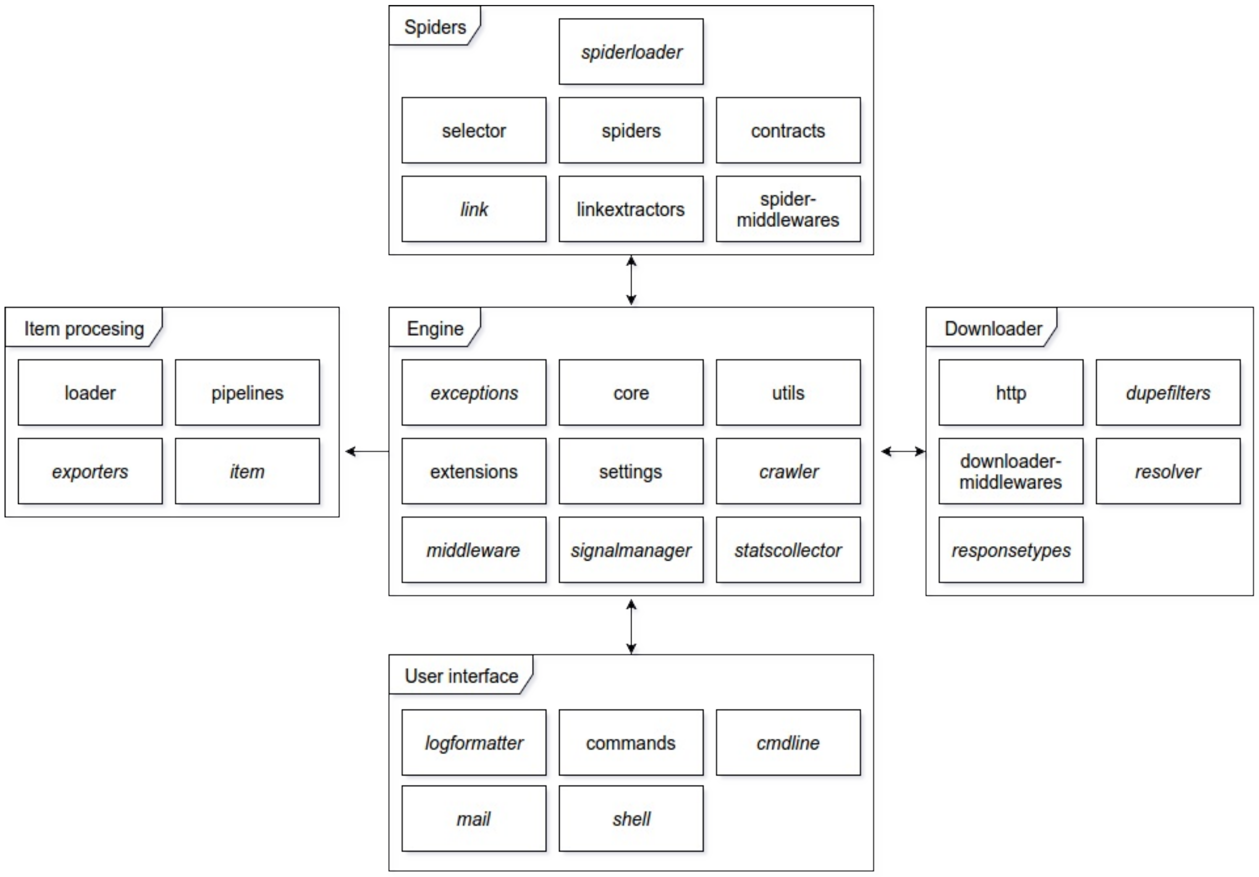


Figure 4: Scrapy dependency diagram

Standardisation of design

Standardisation of design is important since systems are developed by a team. A coherent design will benefit maintainability and reliability since it will be easier to understand or compare different parts of the system. The design can be standardised by using design patterns or common software.

Some standardisations made by Scrapy are to encourage the use of PEP 8 coding style, a documentation policy. Even though the PEP 8 coding style is encouraged it is not heavily enforced and the code contains violations. Some examples are #2429, #2147 and #2368.

Common design model

The common design model lays out some common practices used throughout the project to create coherence and reduce duplication.

Logging

The logging has five different levels: critical and regular errors, warning, info and debugging messages, in decreasing order of severity. Messages are written with a short description of what happened plus a passed object to show the state. Scrapy uses Python's own logging. Previously, Scrapy used its own logging implementation. According to the documentation, this has been deprecated in favour of explicit calls to the Python standard logging, but parts remain for backward compatibility.

External libraries

External libraries should be imported at the top of the file, to adhere to the PEP 8 standard. When importing internal classes, only the used parts are imported. There are exceptions to this rule, one of which is the importing of optional libraries. In this case the import is not guaranteed to work, so this has to be dealt with. An example of this is in httpcache.py, where the `leveldb` import is placed in the class that uses it, to avoid exceptions from being thrown.

Interfaces

Scrapy interfaces with spiders through numerically ordered layers of middleware which can be customised or removed at will through specification in a middlewares dictionary. Each middleware layer needs to implement a spider input, output, exception and startup method. Similarly, Scrapy interfaces with a downloader which does the actual HTTP requests through layers of middleware. These downloader middleware layers need to implement request, response and exception methods. Scrapy also allows for extensions to override the default behaviour.

Module	**Dependencies**
commands	utils, http, *exceptions*, *item*, settings, linkextractors, contracts, shell
contracts	utils, http, *exceptions*, *item*
core	utils, http, *exceptions*, *item*, responsetypes, middleware
downloadermiddlewares	utils, http, *exceptions*, responsetypes, core
extensions	utils, http, *exceptions*, responsetypes, *mail*
http	utils, *exceptions*, *link*
linkextractors	utils, *exceptions*, *link*
loader	utils, *item*
pipelines	utils, http, *exceptions*, settings, selector, middleware
selector	utils, http, *exceptions*
settings	utils, *exceptions*
spidermiddlewares	utils, http, *exceptions*
spiders	utils, http, *exceptions*, selector
utils	http, *exceptions*, *item*, settings, selector, spiders

Table 4: Scrapy modules

Module	Dependencies
cmdline	utils, settings, commands, crawler
crawler	utils, settings, core, extension, resolver, interfaces, signalmanager
dupefilters	utils
exceptions	
exporters	utils, exceptions, item
interfaces	
extensions	utils, middleware
item	utils
link	utils
logformatter	utils
mail	utils
middleware	utils, exceptions
resolver	utils
responsetypes	utils, http
shell	utils, http, exceptions, item, settings, spiders, crawler
signalmanager	utils
spiderloader	utils, interfaces
statscollectors	

Table 5: Scrapy weak modules (`.py` files living directly in the `scrapy` package)

5 Operational viewpoint

The operational viewpoint will describe how Scrapy is operated in the production environment. Here we will identify how Scrapy handles concerns related to control, manage and monitor the system. According to Rozanski and Woods Chapter 22 [1] the operational viewpoint is often not completely designed until it is needed and the system is already constructed. This is also the case for Scrapy, the installation methods, logging and settings have changed during development.

Installation

One operational concern is installing and upgrading the system on users hardware and how to configure the installation.

Scrapy is included in the PyPi repository for Python packages. This makes it possible to install, upgrade or uninstall using pip. The recommended method to install Scrapy is to use a virtual environment to avoid conflicts with system packages. On Windows Anaconda can be used for this but it only works for Python 2.

Live monitoring

Scrapy can be controlled by the command-line tools. It lets the user create, edit and start projects. The tools can also be used to run different Scrapy features without a project such as scraping a single webpage.

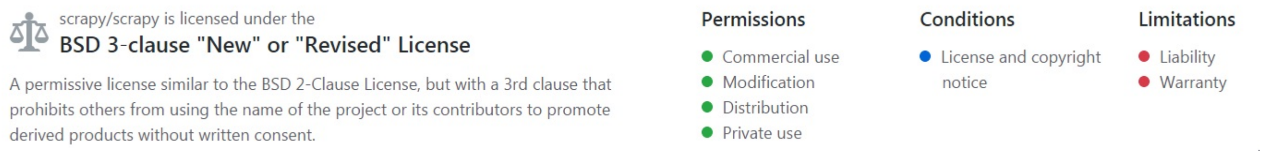
To monitor the system during runtime or production, Scrapy uses logging, described more in Logging. This is also used to capture performance metrics.

Support

Scrapy has two support groups, developers and users of the framework. Developers mostly have problems with understanding the code or tips on how to solve programming problems. The developers get support from each other on GitHub. Users of the framework can get support from StackOverflow or other online communities. The users often have problems with installing Scrapy, connection problems and implementation problems.

Operation in third-party environments

Scrapy may be forked for any use as long as the name Scrapy is not used without consent, as can be seen in Figure 5 which shows the licence information on GitHub.



scrapy/scrapy is licensed under the
BSD 3-clause "New" or "Revised" License

A permissive license similar to the BSD 2-Clause License, but with a 3rd clause that prohibits others from using the name of the project or its contributors to promote derived products without written consent.

Permissions	Conditions	Limitations
● Commercial use	● License and copyright notice	● Liability
● Modification		● Warranty
● Distribution		
● Private use		

Figure 5: Licence of Scrapy.

6 Performance and scalability perspective

Scrapy is a web crawling framework, which generally means that a lot of HTTP requests need to be made when it runs. The big issue with that, is that each request takes a while to get a response. What also could happen is that a response won't be received, which will take even longer.

With this thought in mind, the Scrapy developers decided to build the framework on top of the Twisted framework [5], to be able to handle HTTP requests in an asynchronous manner, without the need to write complex code.

Twisted is a rather heavy-weight networking framework which has been around since 2002. It is written in Python and heavily relies on the event-driven programming paradigm [4]. That means that one can request a webpage, specify the callback functions that should be called when it completes or throws an error, and then run other code while Twisted is making the request. This requires no threads or processes to be spawned, as it can run in one thread, while being non-blocking. That does not mean that Twisted requires requests to be made in one thread only. It offers threading facilities to run the requests in separate threads without exposing the user to the underlying complexity. Twisted is ideal for Scrapy, as it revolves around making many requests, processing the results, and handling errors gracefully. This choice has made Scrapy very performant.

Scrapy offers utilities to control the amount of various performance-related settings. This is done by specifying parameters in the project settings file. These settings involve specifying download delays, timeout durations, cache sizes, whether and how often to retry a failed request, threadpool sizes etc. This way the user has full control over the Scrapy performance and it can be fine-tuned to meet their requirements.

Because Scrapy is able to handle a lot of requests, scrape the response pages and save the data, there is the risk of memory issues. One way this is handled, is by using the `queuelib` library [5]. It offers a way to create several types of queues on the disk and is optimised for speed. This prevents e.g. HTML pages that have yet to be scraped to overflow the memory, as it is written to disk.

As memory leaks are often the cause of bad performance, Scrapy has dedicated several utilities and documentation to debugging these leaks [6]. In the documentation there is also a page on how to measure the performance of a spider using the built-in benchmarking tool [7].

When making a lot of requests, especially to multiple websites, several external factors can limit performance. For example DNS servers or data throughput.

There are many more measures taken to ensure good performance and scalabity [8], [9]. For the sake of brevity, not all of them can be discussed.

7 Technical debt

Technical debt can be defined as [14] :

> a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution.

A static analysis, analysis on the use of SOLID principles, testing debt and the evolution of technical debt on Scrapy is described in this section.

Static analysis

To identify the technical debt we used Pylint which is a static analysis tool for Python. Pylint tests the source code for coding standard PEP 8, error detection related to imports and interfaces and detecting duplicated code.

The result of running Pylint is lots of errors and warnings. Most of the errors are due to decisions taken by the developers and some are mistakes or actual bad coding. The most common errors in the Scrapy code are related to docstrings, imports or PEP 8 compatibility as can be seen in Figure 6.

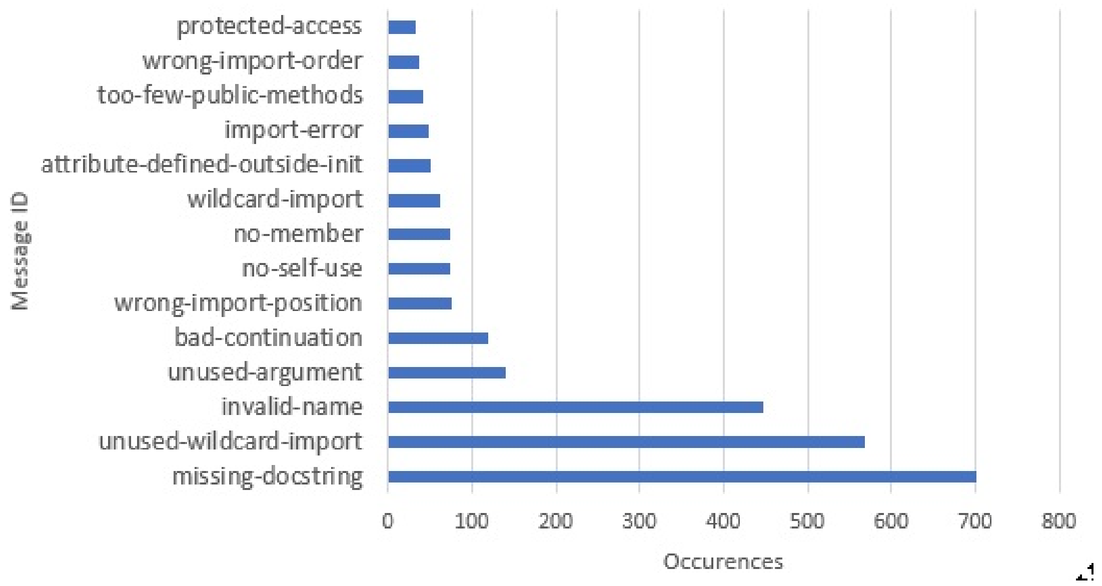


Figure 6: Results of Pylint

Scrapy does not use docstrings for modules which are described in the documentation. It is good to not duplicate the information, also known as the DRY (Don't Repeat Youself) principle [10], but this makes development harder since docstrings are integrated into most IDE's to help code faster (introspection). There are also missing docstrings where they are recommended, such as the 'utils' module which has 78 missing docstring messages from Pylint.

The results also contain a lot of import warnings, some due to convenience for the developers. By having a lot of imports in the `scrapy.__init__` file, Scrapy modules can be imported by using `import scrapy.Module` instead of `import scrapy.package.Class`. Other often occurring warnings in the Pylint report are PEP 8 violations related to code formatting and naming conventions, some lazy exception handling and a few 'FIXME's. These are errors with mostly low impact on the system and are not likely to cause any problems but it is always good to keep the code clean for easier future changes.

The Scrapy framework for item pipelines, DownloaderMiddleware and Spider middleware have methods that can be implemented by the user, but the methods are only documented in the documentation. This could be implemented as abstract classes to inform the user if they forgot to implement a function. The abstract classes also help the user to see which methods they can implement and the interface will make it more clear what type of middleware the class is. The hard part of designing the abstract classes is that all methods are optional and it has to be compatible with current code.

Scrapy´s modules and dependencies are a bit messy and from the dependency tables (Table 4 and Table 5) we can see that there are modules with a lot of dependencies. This leads to tight coupling and we can find classes in the utils modules with very few usages in the code, `ftp.py` , `mulipart.py` , `osssignal.py` which could be moved to other modules to reduce the dependencies in the utils module.

Discussions about technical debt

At the time of writing there are 9 TODO's and 4 FIXME's in the source code. In the Scrapy issues there are numerous discussions about fixing the technical debt. There is for example issue #2144 about making the Scrapy code PEP 8 compliant. However doing this would require a lot of code changes and all attempts have been rejected so far, as it resulted in too many merge conflicts.

There is also issue #8 that involves refactoring code. It has been open since 2011 but it appears that the developers got inspiration on how to fix it after the Django 1.10 release on august 2016.

Other issues involve refactoring as a solution, but are all aimed at improving functionality. Following this mindset, we also created an issue #2633 that will improve documentation and also cleans up the code, hence paying technical debt.

The number of issues that focusses solely on technical debt, in contrast with the number of developer notes in the code, indicates that a lot of technical debt issues are in the source code. This is not a good practice since they have the tendency to be overlooked and therefore will probably remain in the code for a long time.

Scrapy has been around since 2009, which is a long time for a framework. This also means that there have been a lot of changes and refactors to the code. Because of this, there are a lot of deprecated modules and files still present. The reason that they have not yet been removed is mostly for the sake of backward compatibility, according to the developers.

To give an example, the `contrib` module has been fully deprecated since version 1.0.0, which was released in 2015. In Februari 2017 (version 1.3.2) a discussion started (#2568) on finally removing the code. In this issue it was decided to document the deprecation policy to avoid deprecated code for living too long. This will in turn keep the code cleaner, hence paying technical debt.

SOLID principles

Scrapy does not often violate the SOLID principles [11], which is a mnemonic acronym for 'Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion'. These five principles or guidelines, when applied together, will make it more likely that programmers create a system that is easily maintainable.

The Scrapy code does not often use deep levels of abstractions, resulting in a low code complexity. On the other hand this lack of abstractions can create some confusion for developers. As has been explained before, the `item pipelines` , `DownloaderMiddlewares` , `SpiderMiddlewares` but also the `Extensions` can all be abstracted. Each of the classes in these modules implement common optional methods or functionality. These could be abstracted in a common interface. We contributed to the Scrapy project by implementing this change.

Non-intentional SOLID violations are hard to find as a lot of the principles are actively used, though not specifically discussed in the issues or pull-requests. Examples of where the principles are implemented can easily be found. The contracts which are offered for spiders make sure that the newly added spiders work correctly, can be compared with the Liskov Substitution Principle. Any spider following the contract can be replaced with any other spider as they all behave the same. The Single Responsibility Principle is largely adhered to with every scrapy functionality having its own class which becomes immediately clear from the folder structure. The modules in the DownloaderMiddlewares are all separated by different functionality (http authentication, compression, proxies, cookies etc.) which means that the Interface Segregation Principle with many client-specific interfaces is widely used.

Testing debt

Scrapy is overall a very well tested framework. The overall testing code coverage, according to Codecov, is 83.57% for version 1.3.3. In this section we will focus on the 16.43% of the code that is not covered by the tests.

Module / file	Coverage	Reason
`commands`	63%	The `Command` classes are not properly tested if they require a project.
`cmdline.py`	66%	Print functions aren't tested.
`shell.py`	68%	Interactive shell detection is untested (hard to test). Print functions aren't tested.
`contracts.__init__.py`	67%	Some bad weather cases are untested (easy to test). `Commands.add_pre_hook` is untested (easy to test).
`pipelines.files.py`	65%	Amazon S3 storage functionality is badly tested.
`extensions`	76%	A lot of untested low-level (memory) debugging utilities (very hard to test).
`mail.py`	76%	Code returns before the untested code if debug is on. (hard to test).

Table 6: Modules and files with low coverage

To get a good overview of where the testing debt hides, we can employ Codecov to see which modules and files have low coverage. In Table 6 we have compiled an overview of the modules and files with low coverage accompanied by the reasons for this.

What immediately becomes clear from Table 6 is that the command-line interface related code is generally poorly tested. The main reason for this is that it contains a proportional amount of functions dedicated to printing instructions. Although it is not hard to test print statements in Python, it could be speculated that the developers did not feel the need to test these as they are not very complex and by looking at the code, it is clear that they work as intended. Another reason for the low testing coverage of the `commands` module, is that the command-line command that requires a functional Scrapy project are mostly untested. These project-only commands [docs] can be a hassle to test because mocking an entire project, or creating a project that reflects all (or most) possible outcomes of running a command, is hard to do.

The `contracts` module has low coverage as well. The reason for this is rumours that the module might be moved and the code will probably change [12].

There is some Amazon S3 storage functionality that is untested. However the tests are present, but turned off on Travis because they require a real S3 storage. At the moment of writing, there is a discussion going in #1790 on how to run these tests in Travis as well.

Evolution of technical debt

Most issues are picked up quite fast and find their way into master in a relatively short time. Removing code which is deprecated, unused or misplaced is not one of the top priorities. The earliest available Codecov report on the Scrapy GitHub from 19-10-2015 shows a total coverage of 82.64%. At 2-4-2017 this has improved to a total coverage of 84.66%, showing a slight increase in 1.5 years.

There is no official policy which determines when a deprecated function gets deleted from the codebase. As mentioned in issue #2568, developers sometimes experience that coding is taking longer than needed or that mistakes are being made because of the existing technical debt. An example of this are almost-alike imports (e.g. `scrapy.dupefilter` and `scrapy.dupefilters`) of which one is deprecated but still shows up in the IDE autocomplete of the developer.

The same issue mentions the currently unwritten policy on removing deprecated code:

> It is OK to deprecate something in major release X and remove it in major release X+1 (given that they separated by at least half year or a year), but if the deprecated path doesn't hurt it can be kept for much longer.

- @Kmike on Feb 28, 2017

This general strategy can indeed be found when looking at earlier pull-requests, where backward compatibility is often preferred compared to code cleanups. This is also discussed in issue #1096 as well as the strategy for cleanups which happen just before new major releases. As there are not many major version releases this does not happen too often. There are only 23 pull-requests (of a total of 1290) related to cleanups, which started appearing from around 2013 increasing in activity as time goes on. The average time for these changes to make it to the master vary from a month to a year.

8 Conclusion

Scrapy is a widely used and ever-evolving scraping framework. It has a vast community of users and new features and bugfixes appear regularly.

In this chapter we have analysed Scrapy from different viewpoints and perspectives. We have found that due to the long lifetime of the project, the software architecture has become incoherent. This is gradually improving, although at a slow pace.

We are confident that Scrapy will continue to be the de-facto standard for scraping webpages using Python.

9 References

1. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. https://github.com/BruceDone/awesome-crawler
3. https://twistedmatrix.com
4. https://en.wikipedia.org/wiki/Event-driven_programming
5. https://pypi.python.org/pypi/queuelib
6. https://doc.scrapy.org/en/master/topics/leaks.html
7. https://doc.scrapy.org/en/master/topics/benchmarking.html
8. https://doc.scrapy.org/en/master/
9. https://github.com/scrapy/scrapy
10. https://en.wikipedia.org/wiki/Don't_repeat_yourself)
11. https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))
12. https://github.com/scrapy/scrapy/issues/1918
13. https://doc.scrapy.org/en/0.10.3/topics/architecture.html
14. https://en.wikipedia.org/wiki/Technical_debt

Syncthing: Open Source Continuous File Synchronisation

By Jayachithra Kumar, Lidia Fernandez, Robert Carosi, Sacheendra Talluri

Abstract

Syncthing is an application that enables users to synchronise files across different devices. The application is actively maintained and developed by a relatively small group of developers. The foci of the application are preventing data loss, security from attackers and ease of use. Through our analysis, we found that the simplicity of the architecture, consisting of three major layers, allows for easy addition of features while still maintaining the robustness of the core layer consisting of the synchronisation logic. In general, we observe that Syncthing is a well engineered project without glaring technical holes or major debt.

Table of Content

Chapter - Syncthing

Introduction

Syncthing is a software application used for synchronising files across devices. As its major purpose is reliable transport of information, we considered it to be a software worth studying. In our study, we document and analyse the engineering process of Syncthing. We hope it will provide the readers a resource to look at while building a software of similar nature. That is, software which deals with user data of utmost importance and is responsible for tasks critical for the safety of the data such as backups.

Our analysis consists primarily of three parts. The first part deals with the different views of the system. The views give an understanding of the socio-technical environment the system operates in. They also give an understanding of the basic principles the system is organised around. The second part gives a more technical description of the system. It consists of different models and a perspective which show the technical design of the system and the decisions that went into that design. The last section is about technical debt. Any software which has been around for sometime builds up technical debt. How the debt is managed is what makes this section interesting.

Views

Stakeholder View

Users use Syncthing to synchronise their files across different devices. We believe it has a significant amount of users. We extrapolate this from the data found here[1] and also from the number of stars on the Github repository. When last checked, Syncthing had 22919 daily active users with reporting enabled and 15106 stars.

Developers contribute code to the project. The most active developers to the project are @calmh and @AudriusButkevicius. @calmh is also the **founder**. Occasionally, other developers contribute big features but are not really active after that. The whole list of current developers can be found in this file[3].

@calmh and @AudriusButkevicius also have the **maintainer** status. So, they are responsible for the direction the software is going to evolve in. They would also be the de facto **assessors** and **integrators** as they are responsible for any pull request being merged and the overall quality. Specifically, the challenges faced by the **integrators** and how they assess the pull requests is covered in the `Contributions` section of the `Codeline Model`.

The **testers** category is an interesting category as the **developers** themselves are the testers in most cases. The maintainers don't accept code without any associated tests[4]. There are also a group of beta testers composed of users and developers. The list can be found here[5].

The **communicators** category primarily consists of the 2 maintainers and a user named @rumpelsepp. They maintain the documentation up-to-date. There are also several others[6] who have contributed to the documentation who can be grouped under this category.

The **support staff** consists of people who answer questions in the support section of the forum and on the IRC channel. It consists of the maintainers and a few others active on the forum[7]. Other members of the community, like users, help each other using the forum.

@calmh is a de facto **system administrator** for Syncthing instances that use the public discovery server. It is also possible to use Syncthing without using the public discovery server. In that case, whoever is hosting the discovery server used by that Syncthing instance becomes the **system administrator**. As Syncthing is a low level long running system service. All **users** also share the role of **system administrator** in some capacity.

Kastelo is the official corporate **sponsor** of Syncthing. It is a company founded by the maintainer of Syncthing, @calmh. It offers consultancy services and trainings related to Syncthing.

The Syncthing community provides public relay servers so that people behind firewalls can use Syncthing. These servers can be found here[8]. The operators of these servers are **suppliers** who supply the infrastructure required for the people behind firewalls to use Syncthing.

Syncthing is built using the Go programming language. It uses AngularJS for its default front-end and Jenkins for continuous integration. It also uses several major and minor libraries [9]. The communities and companies behind all these projects can be considered software **suppliers** in a loose sense.

There are also applications **dependent** on Syncthing for their own functionality. The developers and users of these applications depend on Syncthing to provide the required functionality. Examples include syncthing-inotify, developed by @Zillode and SyncTrayzor, developed by @canton7.

Syncthing has several **competitors**. Its open source competitors include Librevault and several rsync based solutions. A popular closed source but distributed file syncronisation service is BTSync/Resilio. Its most popular **competitors** would be Dropbox, Google Drive and others.

The influence various stakeholders have on the project can be seen in Figure 1.

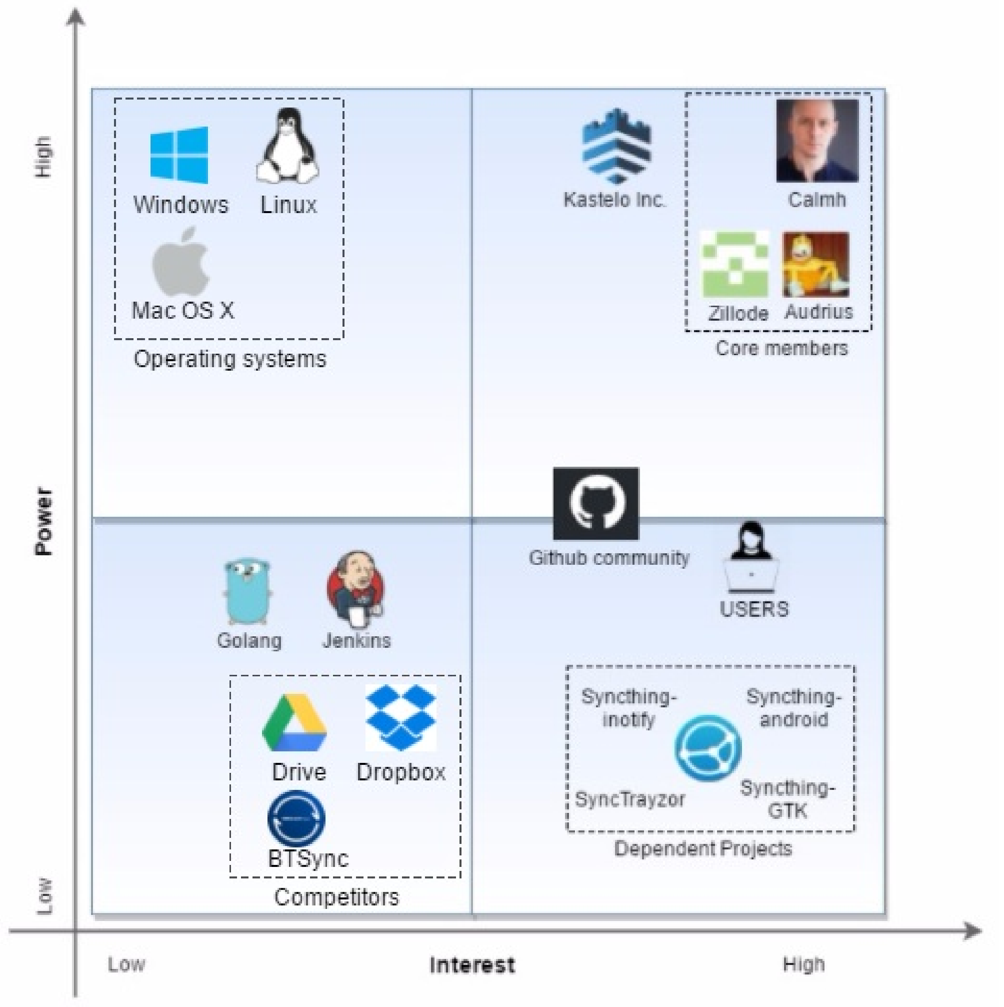


Figure 1: Power interest diagram

Context View

The context view describes the relationships, dependencies, and interactions between the system and its environment. For this purpose, we will determine the scope of Syncthing and we will analyse the external entities and services that interact with it. In Figure 3 we provide a visual overview of the different entities that we are going to describe in this section.

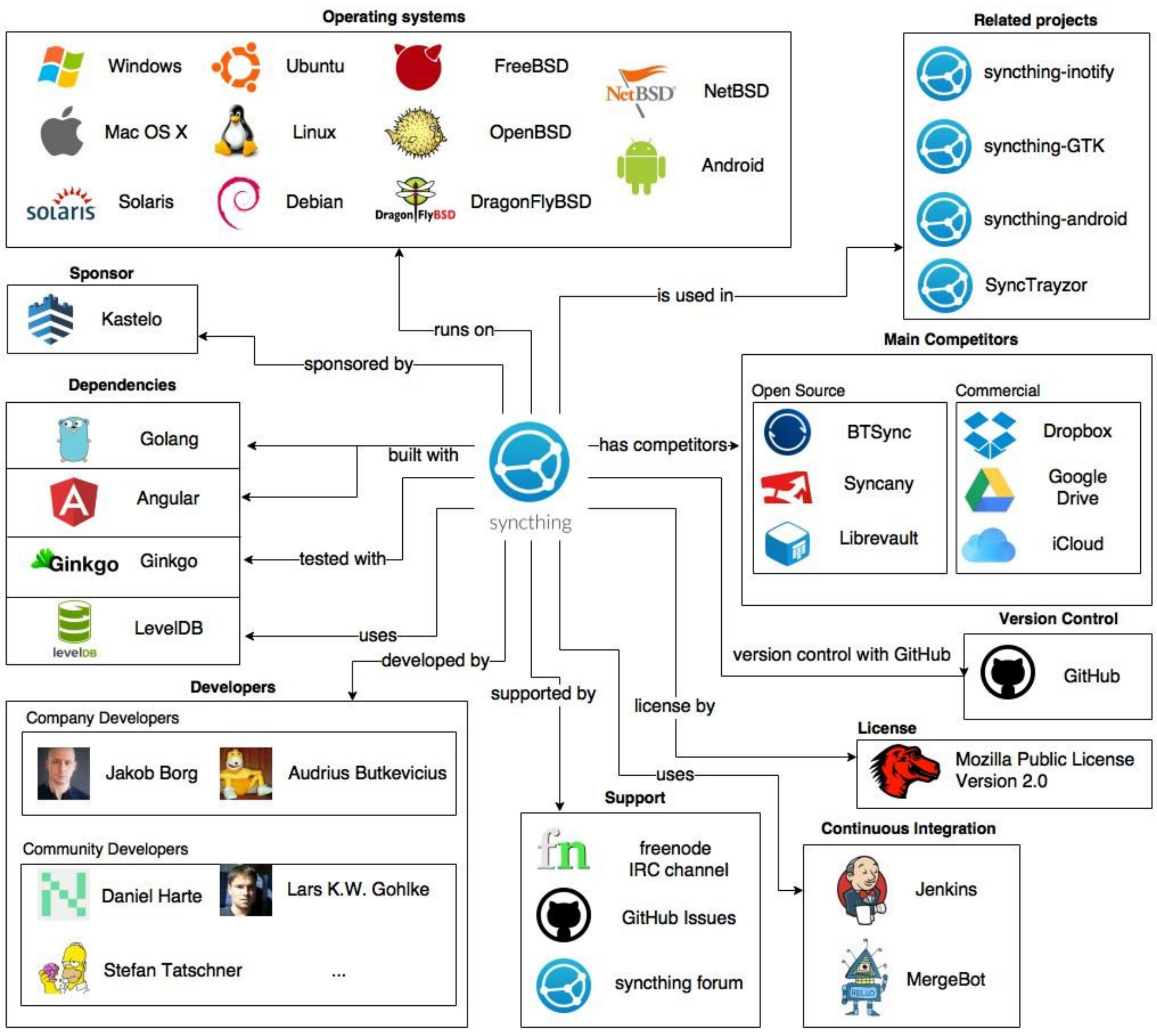


Figure 2: Visual representation of Syncthing's context view

There are multiple external entities that Syncthing interacts with. Analysing its dependencies, we find that Syncthing's core is written in Go, while its frontend is written in Angular. In addition, they have used Ginkgo as Go testing framework. As for the database, Syncthing uses LevelDB, which provides key-value storage. Syncthing also makes use of certain tools for Continuous Integration. Among them, the most important are Jenkins to perform test and MergeBot for automatic merging of Pull Requests. Syncthing community is mainly active in Github, where they organise the work and milestones and publish bugs and features. In addition, they actively use a forum to discuss the project further and a freenode IRC channel.

Information View

The information view tracks the flow of information through the system. In this view, the flow of data during different stages of operation of Syncthing is explained.

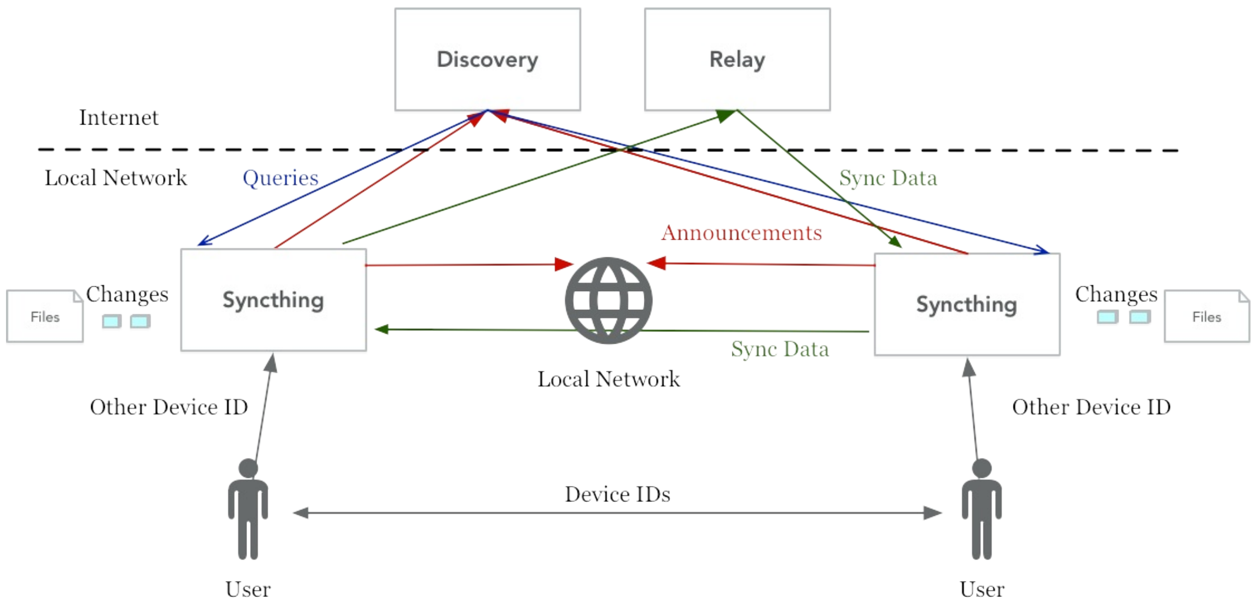


Figure 3: Information Flow diagram with 2 devices

Setup

When started for the first time on a device, Syncthing generates a Device ID. This is used to uniquely identify devices across the network. This network could be over the internet or a local network. It then broadcasts these IDs on the local network and also registers itself with a discovery server which is generally on the internet. This process is called announcing. Then it makes an index of all the files which are selected to be synchronized.

Adding a device

As Syncthing is a synchronization tool, adding another device to which files are to be synchronized is an important part of operation. This can be done when the user informs the Syncthing instances on each device of the Device ID of the other device. The Syncthing instances then contact the Discovery server to identify the location of the other device. This is called Querying. Location of the other device can also be obtained by listening for the announcements from that device if both are on the same local network. After obtaining the location of the other device, the Syncthing instances are ready to synchronize.

Synchronization

This stage is when the exchange of data takes place between the devices. The devices which are to be synchronized exchange the index of files they have. The index on each device is called the local model. The local models of the devices are exchanged and a global model is obtained at each device. This model is then used to synchronize files or updates to files using the Block Exchange Protocol. If possible, this takes place directly between the devices with the devices communicating and exchanging packets directly. But if the devices are not directly reachable (Possible if they are behind a NAT for example), Relay servers are used. When a relay server is in use, packets are first sent to the relay server which pushes them to the destination.

Models

Module Structure Model

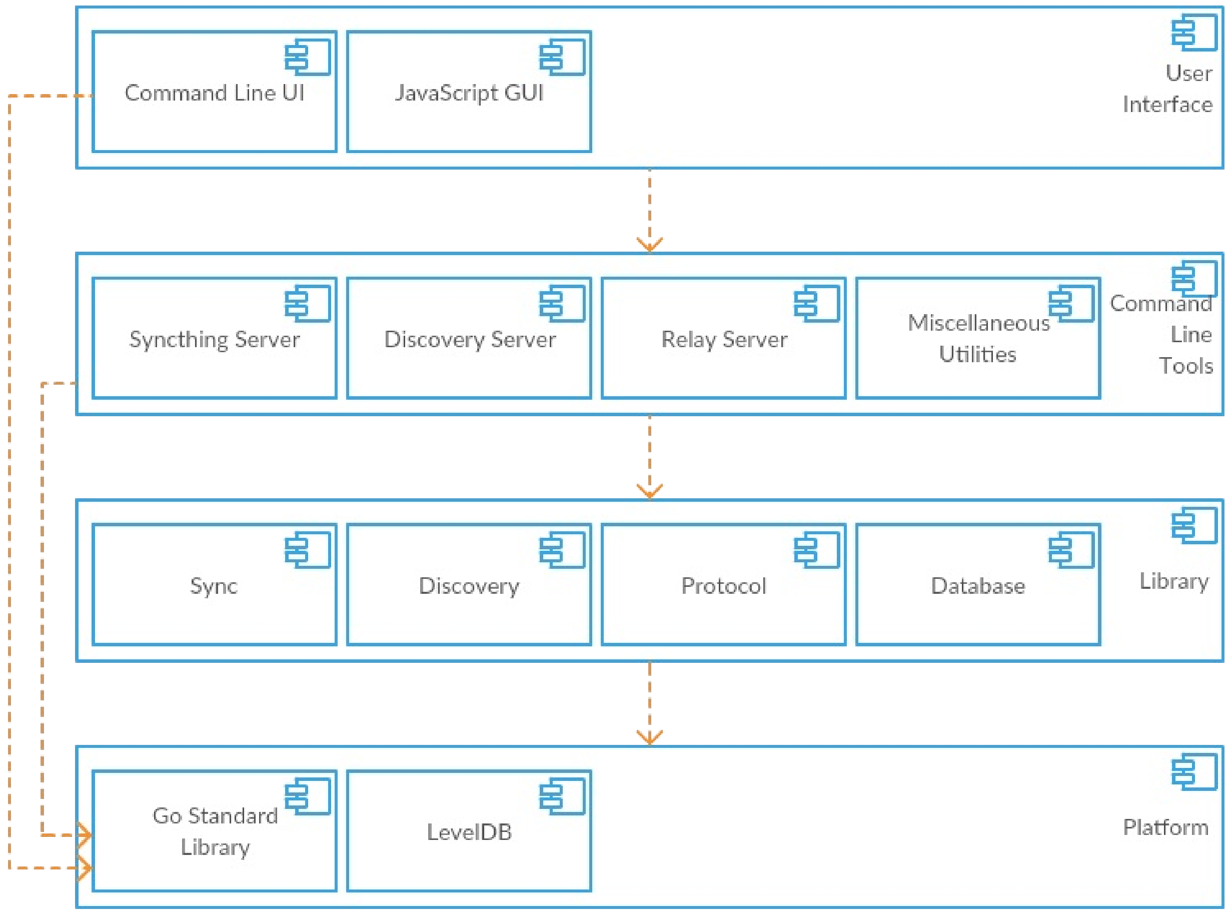


Figure 4: Syncthing's directory structure

The module structure model describes how the project is organised. Figure 4 is a high level UML component diagram of Syncthing. The project is organised into clearly identifiable layers with somewhat clear dependencies. Generally, the layer depicted above depends on the one depicted below it, but there are some dependencies which don't respect this boundary.

The outer boxes represent layers. The inner boxes represent the individual components that form those layers. The arrows represent that the component at the tail of the arrow depends on the component at the head of the arrow. The Platform corresponds to the Go standard library and other external dependencies. The Library is where the major logic of Syncthing like the Block Exchange Protocol, used for exchanging data related to file changes, is implemented. The Command Line Tools are a collection of tools the users actually use to interact with the Syncthing library. The GUI consists of the web based GUI and the command line based GUI.

The Syncthing server, discovery server, relay server and other miscellaneous components in the command line tools layer are the components users of Syncthing use to synchronise their files across devices. The Syncthing server is the one responsible for synchronising files across the user's devices in a peer to peer manner using the Block Exchange Protocol. The discovery server helps Syncthing servers find other Syncthing servers to synchronise with. The relay servers help Syncthing servers behind firewalls and NAT (Network Address Translation) enabled networks connect to each other. Other miscellaneous components are used to register a relay server with the Syncthing network, get information on files being synchronised by Syncthing and other purposes.

The sync, discovery, protocol and database components in the library layer contain the core logic of Syncthing. The sync component calculates hashes of blocks of files and finds which ones to synchronise. The discovery component uses discovery servers to discover other Syncthing servers and also contains logic for creating a discovery server. The protocol component implements the Block Exchange Protocol which takes care of the key exchange for authorisation and encryption between Syncthing servers and also communicating information about which files need to be synchronised. The database component is used for interacting with the LevelDB database so that Syncthing can store information about other Syncthing servers, synchronised and unsynchronised files, block hashes and user preferences.

The Go standard library, LevelDB and other miscellaneous libraries form the platform layer. These are used for encrypting network connections, calculating hashes, storing data and several other functions.

Common Design Model

The common design model will cover aspects of the software where common design approaches and common software components were used. As Syncthing is written in Go (also referred to as golang), all practices in Effective Go[11] are enforced.

Initialisation: It refers to the steps a component is required to take before becoming fully functional[12].

In Syncthing, all components are required to do the following during initialisation.

- Instantiate a logger object. This is done inside the `debug.go` which is present in every component.

- `main.go` files are used as entry points for command line applications. In these files, the init function is used to store the build date, architecture of the system and golang version to be used for debugging.
- In components with `main.go` files, the logger is initialised in the main function.

Termination and restart of operation: It refers to the conventions to be followed when the program terminates either properly or due to an error. It also includes the subsequent steps taken for recovery during a restart. In golang, concurrency is handled through lightweight threads called goroutines. Each goroutine has its own stack trace. When a component exits due to an error, care is taken to ensure that the stacktraces of all the running goroutines are printed. If an issue such as database corruption is detected during restart the user is notified instead of silently logging to a file.

Message Logging and Instrumentation: For the command line applications, the `log` package available in the golang standard library is used. For the main library, a custom logging package is implemented and this is imported and used by every package. The custom logger supports setting the name of the calling component during initialisation, printing it along with the error message. The go-metrics third party package is used for collecting performance metrics. StatHat is used for collecting metrics such as daily active users.

Internationalisation: It refers to the process of allowing the software to support artefacts for different locales. Multi language support was added to Syncthing in v9.0. Syncthing uses Transifex[17] for internationalisation. Transifex generates JSON formatted files containing the available strings in the languages specified by the person building Syncthing. The Syncthing GUI dynamically loads these translations and uses string interpolation to insert them in the right places. The language can be configured in the configuration UI.

Processing configuration parameters: In the Syncthing library, all configuration is passed through the universal configuration object. This object is passed to all components which can be configured and the components read what they require. They also subscribe to future changes in the object. In the command line application, configuration options are passed through command line flags using the `flags` package found in the golang standard library. Configuration can be passed to the library using JSON or XML config files. When configuration is changed using the GUI or command line tools, these files are also changed.

Database interaction: The database used by Syncthing to store local data is LevelDB. LevelDB unlike SQL databases is a low level storage engine with a key-value interface. Concurrency primitives and other facilities such as transactions are implemented in the `db` component. So, any interaction with the database should go through the `db` component.

Pull Requests and Issues: Pull requests and issues follow specific formats defined in the files `PULL_REQUEST_TEMPLATE.md` and `ISSUE_TEMPLATE.md` . When a person either makes a pull request or opens an issue, GitHub displays these templates in the description to be filled[13].

Codeline Model

Codeline Model is used to keep an order when it comes to the organization of the system code. In order to describe Syncthing's Codeline Model, we will provide an overview of the source code structure and the contribution process, based in the information given in [1].

Source code structure

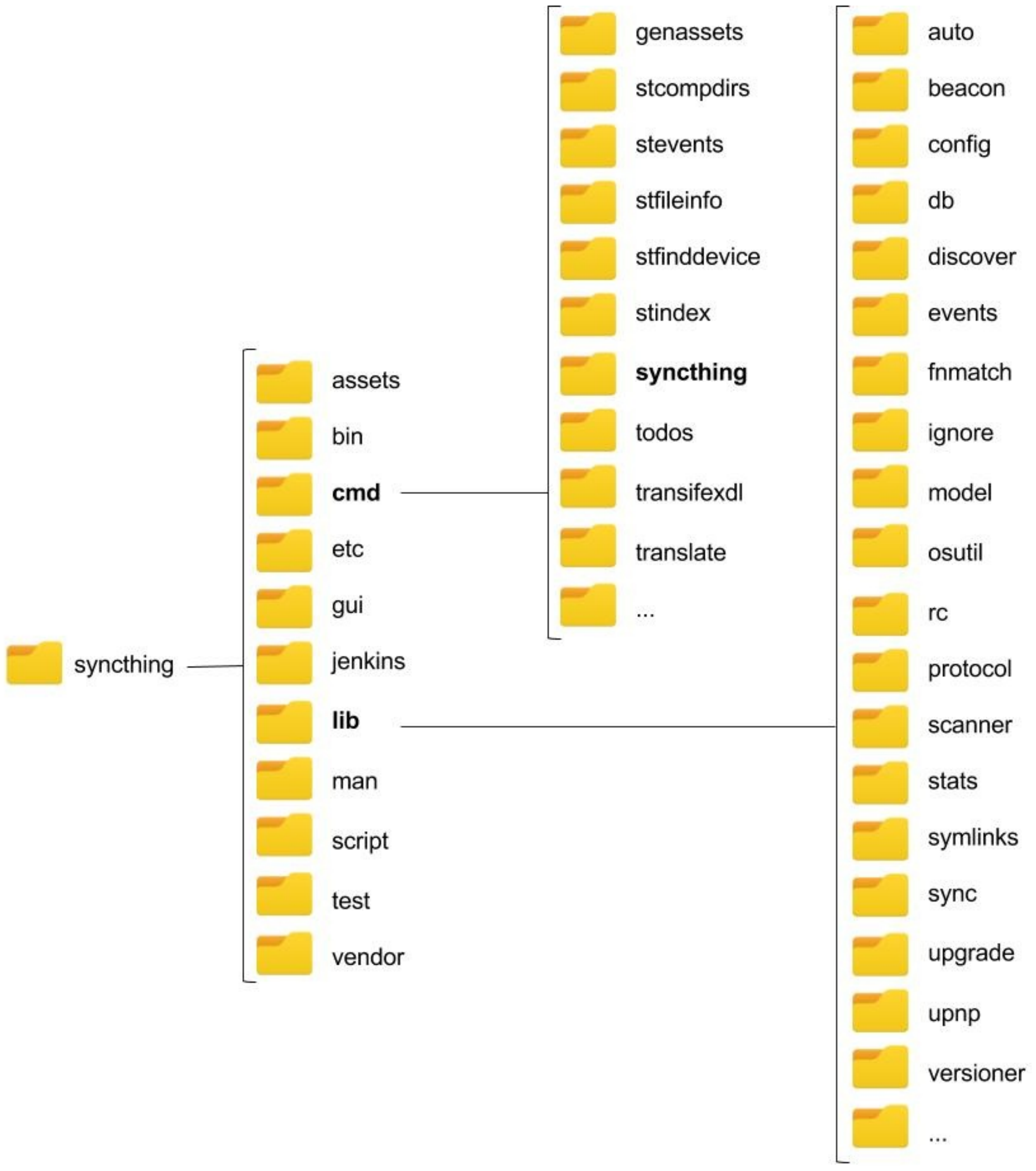


Figure 5: Syncthing's directory structure

In the source repository[16] we can find a tree of various packages and directories. The actual code lives in the `cmd/syncthing` . `lib` directories -contains all packages that make up the parts of syncthing. The web GUI lives in `gui` . A detailed description of the structure can be found in [15].

Contributions

In order to contribute to the project, developers should submit a pull request in the GitHub project. This pull request can belong to three different categories[15]- Trivial, Minor or Major- following semantic versioning[2]. Depending on these categories, the pull request will follow different requisites before being accepted:

- Trivial: These may be merged without review by any member of the core team.
- Minor: It can be merged on approval by any other developer on the core or maintainers team.
- Major: It must be reviewed by a member of the maintainers team.

When tests are passed and the pull request is approved, it will be committed into the main code. After a commit, the next step is the launch of a release, which are again classified into three types15]:

- A new patch release is made each Sunday, although serious bugs, that would crash the client or corrupt data, cause an immediate patch release.
- Minor releases are made when new functionality is ready for release. This happens approximately once every few weeks.
- A new major release happens when a whole product is ready. At the time of writing this book, it has not yet happened: the project hasn't reached yet the 1.0.0 version.

During the whole contribution process, the project uses Github issues to track bugs, feature requests or any other requirements needed [15]. Some issues are assigned to milestones, which are associated with future releases.

Usability Perspective

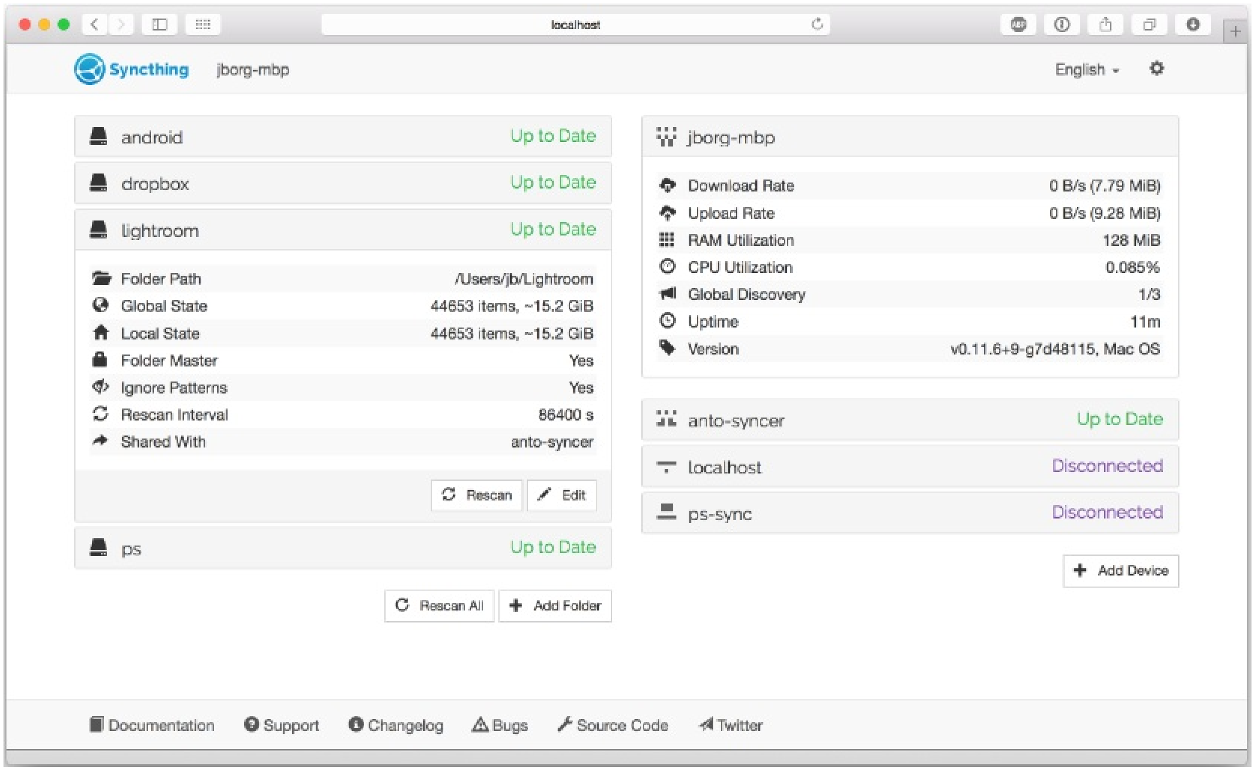
The usability perspective aims to comprehend the ease with which people who interact with the system can work effectively[14]. This perspective addresses a wide range of loosely connected concerns such as the usability of User Interface, process flow around the system, information quality and architecture of the system. Usability perspective focuses on the end-users of the system, but also addresses the concerns of any others who interact with it directly or indirectly, such as developers, maintainers and support panel. In case of Syncthing, the success of the system depends on the effectiveness with which files can be shared and synchronised between devices on local network or between remote devices over the internet. By analysing the system usability we get an impression of what a Syncthing instance looks like and how it facilitates high usability. In this section we analyse the usability of Syncthing by identifying the users, touch points and the interaction between both.

Identifying Users

As the purpose of Syncthing is quite general, the user base of Syncthing cannot be narrowed down to a specific group of people. However, Syncthing has a large user base. From the latest information collected from Syncthing usage data we can see that Syncthing currently has 22919 users. Apart from regular users, Syncthing is also used by developers and maintainers for testing and providing support.

Identifying Touch points

Touch points are defined as places where a user may interact with the system. A Syncthing instance can be generally accessed through a web interface as shown in Figure 6. Since Syncthing has such a broad and diverse level of user base, Syncthing's user interface is kept simple to use and easy to navigate through.



android	Up to Date
dropbox	Up to Date
lightroom	Up to Date
Folder Path	/Users/jb/Lightroom
Global State	44653 items, ~15.2 GiB
Local State	44653 items, ~15.2 GiB
Folder Master	Yes
Ignore Patterns	Yes
Rescan Interval	86400 s
Shared With	anto-syncer
ps	Up to Date

jborg-mbp	
Download Rate	0 B/s (7.79 MiB)
Upload Rate	0 B/s (9.28 MiB)
RAM Utilization	128 MiB
CPU Utilization	0.085%
Global Discovery	1/3
Uptime	11m
Version	v0.11.6+9-g7d48115, Mac OS
anto-syncer	Up to Date
localhost	Disconnected
ps-sync	Disconnected

Figure 6: Syncthing home screen

Syncthing developers divide the web UI into two main groups: folder view and device view.

Folder View

This is the left side of the interface shown in Figure 6, that shows the ID and the current state of all configured folders. Clicking on the folder name causes the section to expand to show more detailed folder information like its folder path and the devices that the folder is shared with. It also shows two buttons **Rescan** - for forcing a rescan, and **Edit** - for editing the configuration. Furthermore, a folder can be in any one of the following states:

- **Unknown** - while GUI is loading,
- **Unshared** - when you have not shared this folder,
- **Stopped** - when the folder has experienced an error,
- **Scanning** - while Syncthing is looking in the folder for local changes,
- **Up to Date** - when the folder is in sync with the rest of the cluster,
- **Syncing** - when the device is downloading changes from network.

Among these folder details you can see the current "Global State" and "Local State" summaries as well as the amount of "Out of Sync" data if the folder state is not up to date.

Global State This indicates how much data the fully up to date folder contains. This is basically the sum of the newest versions of all files from all connected devices.

Local State This shows how much data the folder actually contains right now. This can be either more or less than the global state, if the folder is currently synchronising with other devices.

Out of Sync This shows how much data needs to be synchronized from other devices. This is basically the sum of all out of sync files - if you already have parts of such a file, or an older version of the file, less data than this will need to be transferred over the network.

Device View

This is the right side of the interface shown in Figure 6, that shows the overall state of all configured devices. The local device is always at the top with the remote devices in alphabetical order below. For each device you see its current state and when expanded, more detailed information. In the detailed information it shows the "Download Rate" and "Upload Rate". These transfer states are from the perspective of a local device, even those shown for remote devices. The rates for the local devices are the sum of those for the remote devices.

Apart from the different UI groups, since Syncthing is an open source platform that is used throughout the world, it should be able to provide support in different languages. Currently Syncthing provides translations in 35 different languages and developers are working to add more languages to the list. The preferred language can be chosen from the drop-down menu on the top right corner of the window.

Technical Debt

Identifying Technical Debt

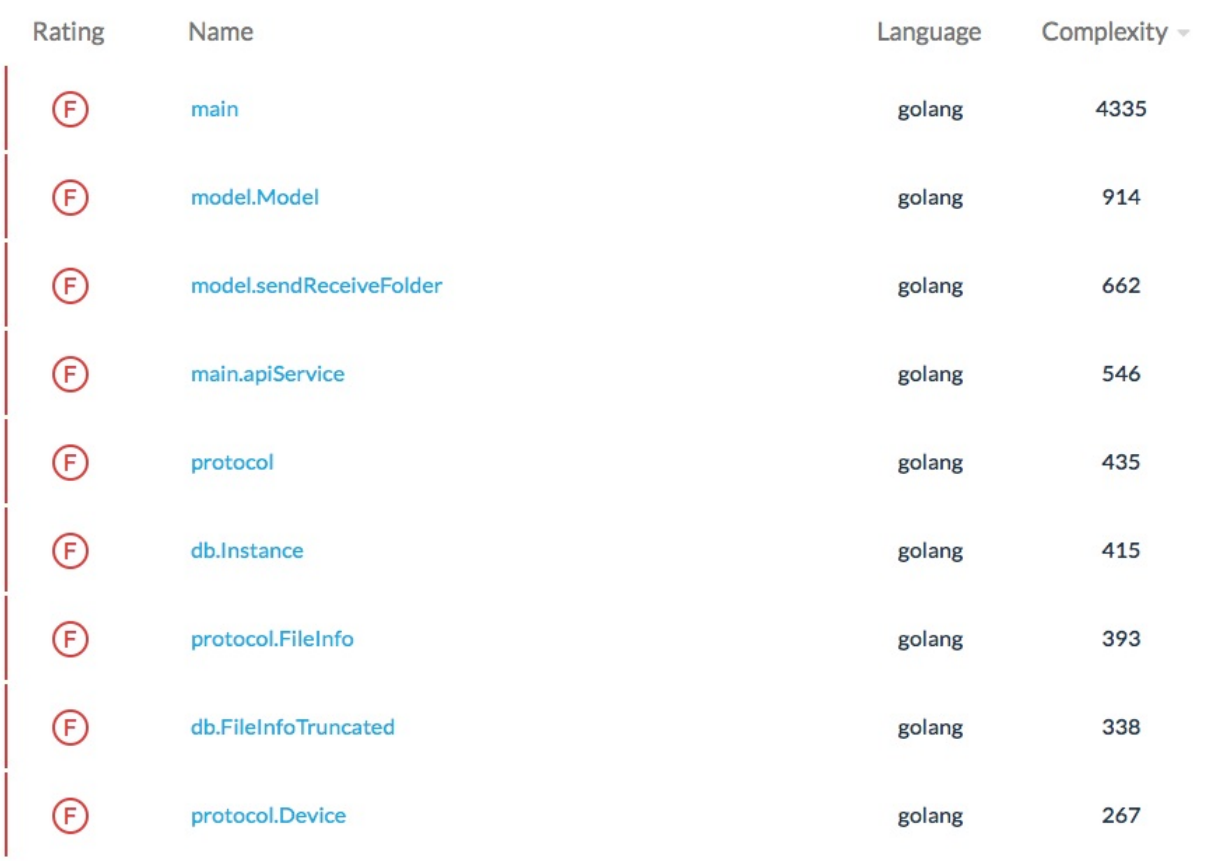
In this section we will provide an analysis of the technical debt of Syncthing. For this purpose, we will use an automatic tool for code quality analysis. We will describe in detail the most relevant issues and we will also indicate which parts of the project present high-priority technical debt that should be tackled.

Automated code quality analysis - Codebeat

Codebeat is a tool for automatic code review. It gathers the result of code analysis into a single, real-time report that gives all project stakeholders the information required to improve code quality. Codebeat analysis is very thorough and it can be consulted at [https://codebeat.co/projects/github-com-syncthing-syncthing]. We won't include a complete description of the report in this document, but we will analyze some of the most representative results that may be of interest to assess the technical debt.

Codebeat organizes its reports in three sections: Complexity, Code Issues and Duplication.

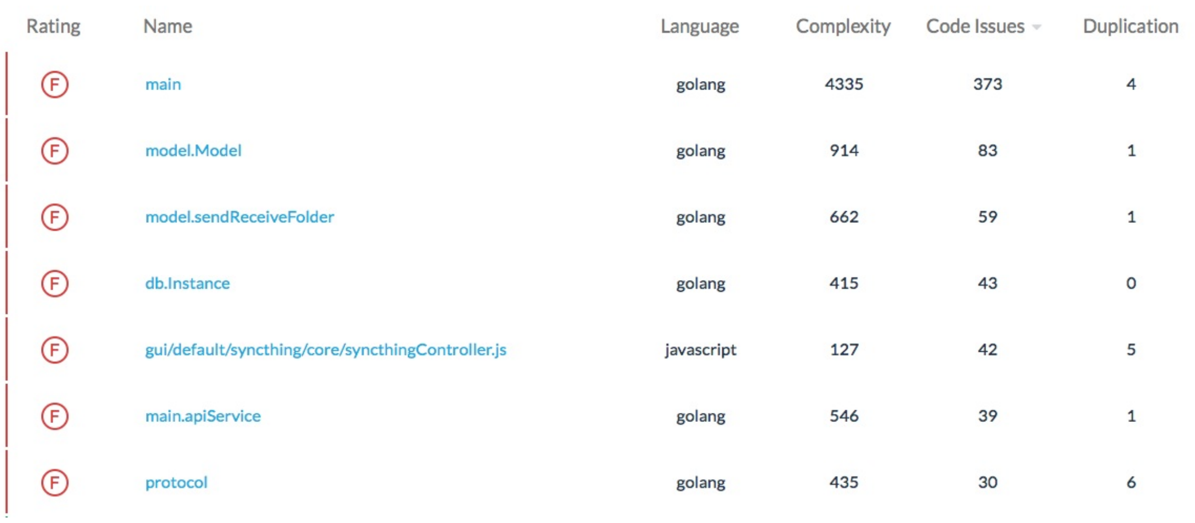
- **Complexity**: In Codebeat, high complexity indicates part of code that contains too much logic and should be broken down into smaller pieces. It can also indicate that the few existing functions are each too busy and they need to be individually refactored. Namespaces with scores over 250 are considered to be high complexity. In Figure 7, we can observe that several namespaces have a high complexity, some of them even ten times over the minimum threshold. All of this issues should be urgently addressed.



Rating	Name	Language	Complexity
F	main	golang	4335
F	model.Model	golang	914
F	model.sendReceiveFolder	golang	662
F	main.apiService	golang	546
F	protocol	golang	435
F	db.Instance	golang	415
F	protocol.FileInfo	golang	393
F	db.FileInfoTruncated	golang	338
F	protocol.Device	golang	267

Figure 7: Fragment of namespaces ordered by complexity

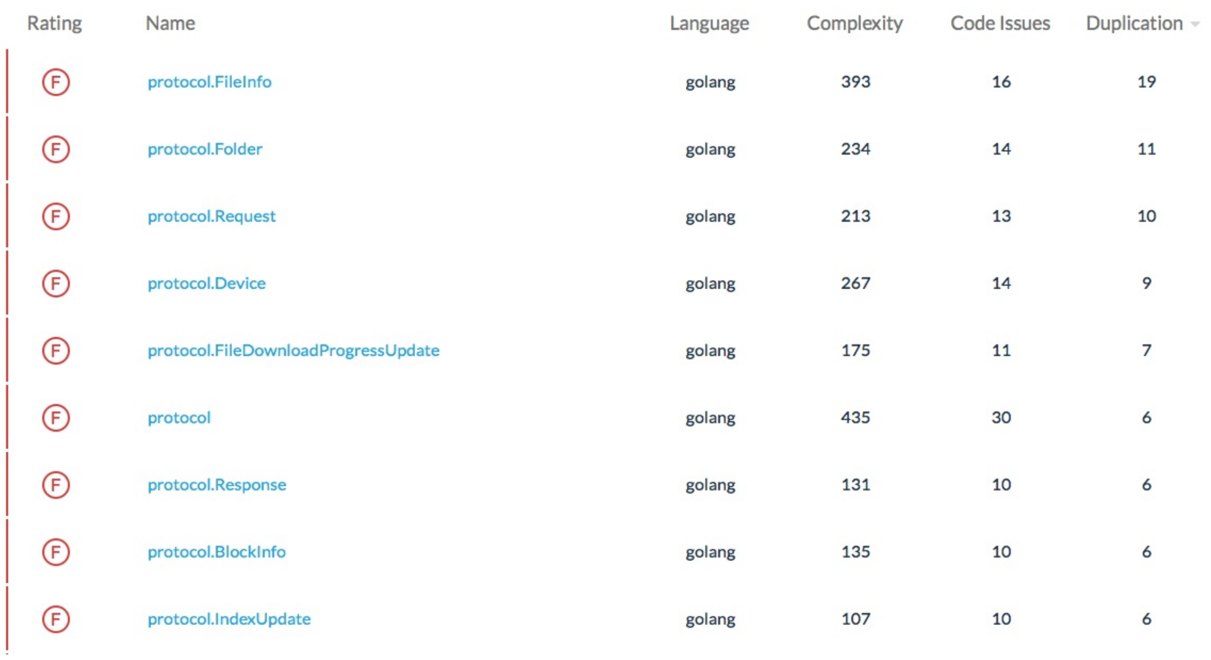
- **Code Issues**: In Codebeat, code issues analysis contains pieces of code that present several issues that can be improved. In this case, the number does not represent a score: it represents the number of issues found in the code. These issues can be related to too many lines of codes, too high number of functions, too many instance variables or too much block nesting. However, this metric does not represent necessarily a problem that needs to be addressed: it only indicates pieces of code that are not ideal, and that may need to be analysed. In Figure 8, we can see that `main` package and `model` library present several issues. These pieces of code would certainly need a revision to estimate how important are these issues and if they indeed pose a problem.



Rating	Name	Language	Complexity	Code Issues	Duplication
F	main	golang	4335	373	4
F	model.Model	golang	914	83	1
F	model.sendReceiveFolder	golang	662	59	1
F	db.Instance	golang	415	43	0
F	gui/default/syncthing/core/syncthingController.js	javascript	127	42	5
F	main.apiService	golang	546	39	1
F	protocol	golang	435	30	6

Figure 8: Fragment of namespaces ordered by code issues

- **Duplication**: In Codebeat, this metric gives suggestions of duplicated pieces of code that should be redesigned to avoid this issue. Although we can see several cases in Figure 9, if we dive deeper in the Codebeat full analysis, we can see that none of the detected duplication is considered critical: they are just warnings. These issues are most likely small duplications difficult to avoid, and as so they are not urgent issues that should be tackled immediately.



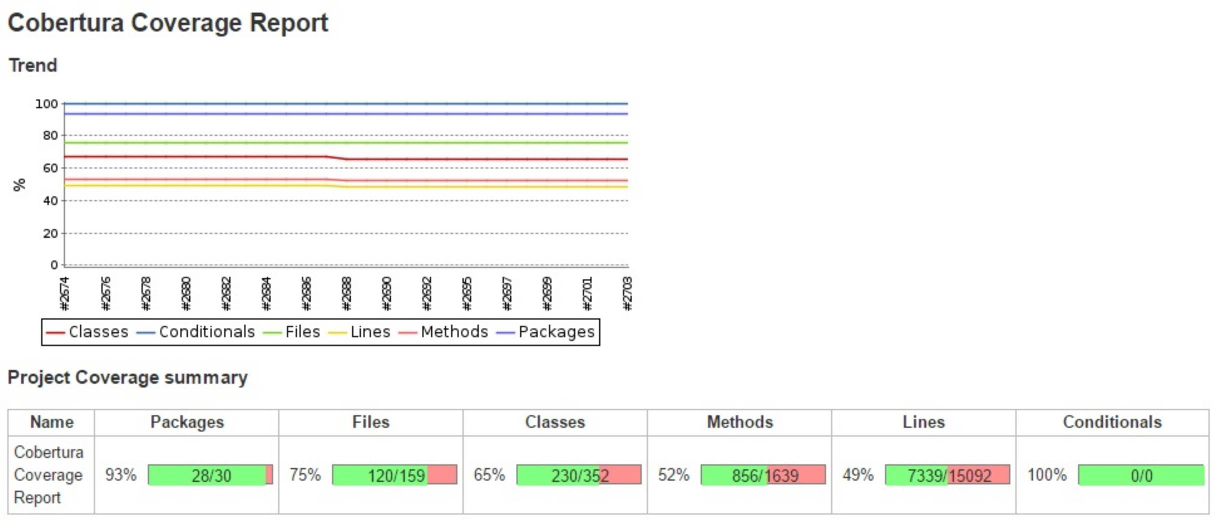
Rating	Name	Language	Complexity	Code Issues	Duplication
F	protocol.FileInfo	golang	393	16	19
F	protocol.Folder	golang	234	14	11
F	protocol.Request	golang	213	13	10
F	protocol.Device	golang	267	14	9
F	protocol.FileDownloadProgressUpdate	golang	175	11	7
F	protocol	golang	435	30	6
F	protocol.Response	golang	131	10	6
F	protocol.BlockInfo	golang	135	10	6
F	protocol.IndexUpdate	golang	107	10	6

Figure 9: Fragment of namespaces ordered by duplication level

Identifying Testing Debt

Testing debt measures the extent to which an application is tested properly, in order to ensure that the application keeps functioning even after changes are made.

Syncthing requires every contributor to include tests for both Minor commits - which includes adding a simple new feature, bugfix or refactoring, and Major commits - that include adding new complex feature, large refactorings or changing the underlying architecture of (parts of) the system. These tests can be run using the "go run build.go test" command. When it comes to code coverage, Syncthing maintains an extensive coverage report for each build. Syncthing uses cobertura, an open source code coverage tool and a plug-in of Jenkins that generates code coverage metrics to record and display. The extensive code coverage results for Syncthing can be found here. These metrics are generated not only for the latest build but the coverage results of the previous builds are also kept track of, and are used to verify how these results evolve over time. Figure 10 shows these code coverage results for each package, class, and files, and as we can see the package-wise coverage results for Syncthing remain a constant 93% throughout several builds. However, by checking the coverage results for objects, we notice that there is a minor drop in coverage after build 2688. The current coverage is 65% and hence it is safe to conclude that there is still some room for improvement.

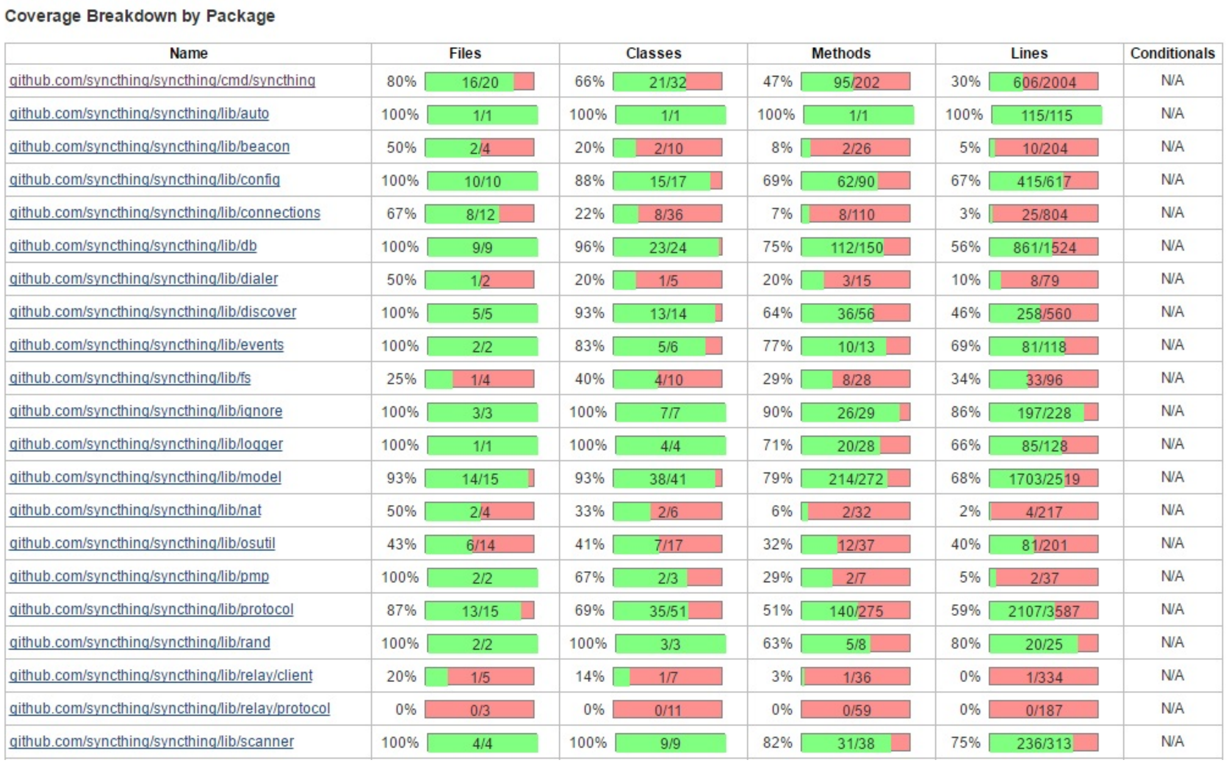


Project Coverage summary

Name	Packages		Files		Classes		Methods		Lines		Conditionals	
Cobertura Coverage Report	93%	28/30	75%	120/159	65%	230/352	52%	856/1639	49%	7339/15092	100%	0/0

Figure 10: Code coverage results

Furthermore Jenkins provides the code coverage results for each package in the application as shown in Figure 11. As we can see, several packages under 'lib' directory have a 100% coverage. However, there are also few packages like 'lib/relay/protocol' with 0% code coverage and this is because the test files in these directories are empty. Writing tests for these directories would increase the overall code coverage of the application to a great extent.

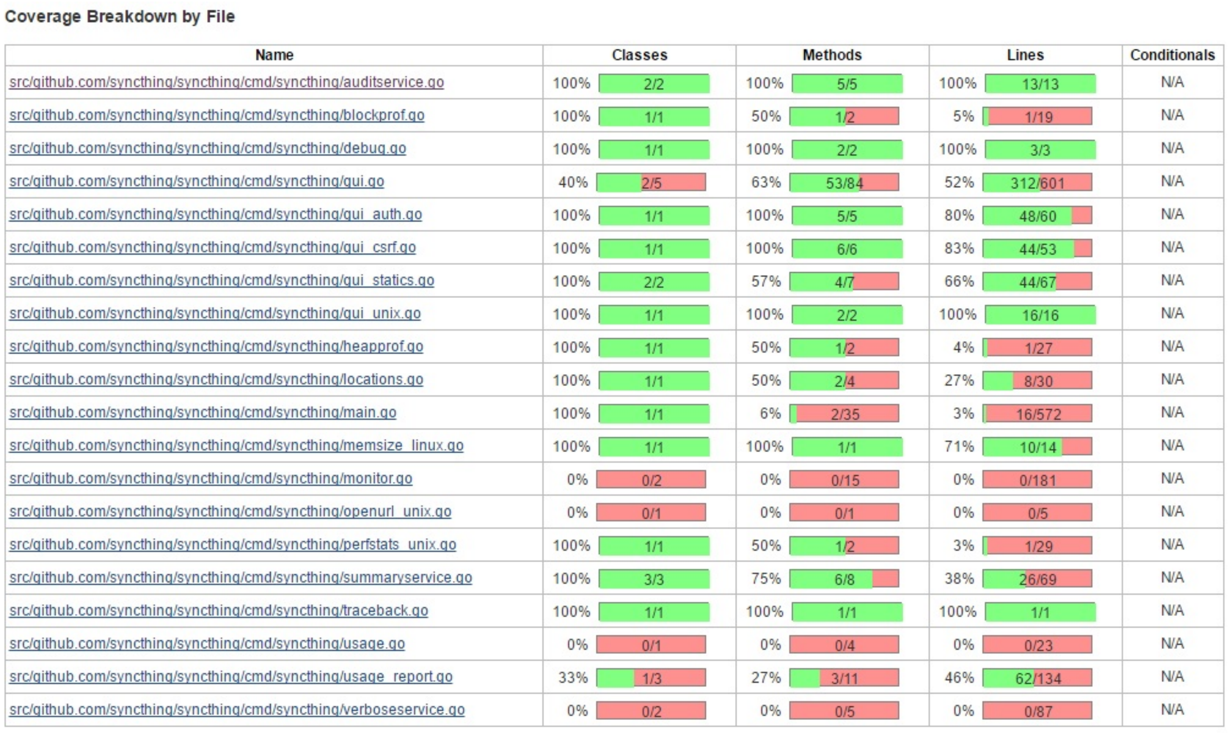


Coverage Breakdown by Package

Name	Files	Classes	Methods	Lines	Conditionals
github.com/syncthing/syncthing/cmd/syncthing	80% 16/20	66% 21/32	47% 95/202	30% 606/2004	N/A
github.com/syncthing/syncthing/lib/auto	100% 1/1	100% 1/1	100% 1/1	100% 115/115	N/A
github.com/syncthing/syncthing/lib/beacon	50% 2/4	20% 2/10	8% 2/26	5% 10/204	N/A
github.com/syncthing/syncthing/lib/config	100% 10/10	88% 15/17	69% 62/90	67% 415/617	N/A
github.com/syncthing/syncthing/lib/connections	67% 8/12	22% 8/36	7% 8/110	3% 25/804	N/A
github.com/syncthing/syncthing/lib/db	100% 9/9	96% 23/24	75% 112/150	56% 861/1524	N/A
github.com/syncthing/syncthing/lib/dialer	50% 1/2	20% 1/5	20% 3/15	10% 8/79	N/A
github.com/syncthing/syncthing/lib/discover	100% 5/5	93% 13/14	64% 36/56	46% 258/560	N/A
github.com/syncthing/syncthing/lib/events	100% 2/2	83% 5/6	77% 10/13	69% 81/118	N/A
github.com/syncthing/syncthing/lib/fs	25% 1/4	40% 4/10	29% 8/28	34% 33/96	N/A
github.com/syncthing/syncthing/lib/ignore	100% 3/3	100% 7/7	90% 26/29	86% 197/228	N/A
github.com/syncthing/syncthing/lib/logger	100% 1/1	100% 4/4	71% 20/28	66% 85/128	N/A
github.com/syncthing/syncthing/lib/model	93% 14/15	93% 38/41	79% 214/272	68% 1703/2519	N/A
github.com/syncthing/syncthing/lib/nat	50% 2/4	33% 2/6	6% 2/32	2% 4/217	N/A
github.com/syncthing/syncthing/lib/osutil	43% 6/14	41% 7/17	32% 12/37	40% 81/201	N/A
github.com/syncthing/syncthing/lib/pmp	100% 2/2	67% 2/3	29% 2/7	5% 2/37	N/A
github.com/syncthing/syncthing/lib/protocol	87% 13/15	69% 35/51	51% 140/275	59% 2107/3587	N/A
github.com/syncthing/syncthing/lib/rand	100% 2/2	100% 3/3	63% 5/8	80% 20/25	N/A
github.com/syncthing/syncthing/lib/relay/client	20% 1/5	14% 1/7	3% 1/36	0% 1/334	N/A
github.com/syncthing/syncthing/lib/relay/protocol	0% 0/3	0% 0/11	0% 0/59	0% 0/187	N/A
github.com/syncthing/syncthing/lib/scanner	100% 4/4	100% 9/9	82% 31/38	75% 236/313	N/A

Figure 11: Package-wise coverage results

By drilling down further, we were able to analyse the code coverage results for each file in a package and each class in a file. When we get to this level, Jenkins displays both the overall coverage statistics for the class and also highlights the lines that were covered in green, and those that weren't in red. For example, in 'cmd/syncthing' package, certain files like 'gui.go' have much less code coverage as shown in Figure 12 and thus reduces the overall coverage of the package.



Coverage Breakdown by File

Name	Classes	Methods	Lines	Conditionals
src/github.com/syncthing/syncthing/cmd/syncthing/auditservice.go	100% 2/2	100% 5/5	100% 13/13	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/blockprof.go	100% 1/1	50% 1/2	5% 1/19	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/debug.go	100% 1/1	100% 2/2	100% 3/3	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/gui.go	40% 2/5	63% 53/84	52% 312/601	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/gui_auth.go	100% 1/1	100% 5/5	80% 48/60	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/gui_csrf.go	100% 1/1	100% 6/6	83% 44/53	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/gui_statics.go	100% 2/2	57% 4/7	66% 44/67	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/gui_unix.go	100% 1/1	100% 2/2	100% 16/16	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/heapprof.go	100% 1/1	50% 1/2	4% 1/27	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/locations.go	100% 1/1	50% 2/4	27% 8/30	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/main.go	100% 1/1	6% 2/35	3% 16/572	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/memsize_linux.go	100% 1/1	100% 1/1	71% 10/14	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/monitor.go	0% 0/2	0% 0/15	0% 0/181	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/openurl_unix.go	0% 0/1	0% 0/1	0% 0/5	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/perfstats_unix.go	100% 1/1	50% 1/2	3% 1/29	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/summaryservice.go	100% 3/3	75% 6/8	38% 26/69	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/traceback.go	100% 1/1	100% 1/1	100% 1/1	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/usage.go	0% 0/1	0% 0/4	0% 0/23	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/usage_report.go	33% 1/3	27% 3/11	46% 62/134	N/A
src/github.com/syncthing/syncthing/cmd/syncthing/verboseservice.go	0% 0/2	0% 0/5	0% 0/87	N/A

Figure 12: Coverage results for files

By studying the lines denoted by the red highlights, we can see that methods dealing with Database connections are not covered properly and by fixing this, a potential technical debt could be avoided.

Suggestions to reduce the project debt

Technical debt is quite low in Syncthing. In the collaboration guidelines it is specified that, before merging a Pull Request, the proposed code should pass with 0 errors several linters regarding style and syntax, so the project does not present these kind of issues. Also, they have been very careful with their interfaces, and the duplicates are not high priority issues since they usually have strong reasons to repeat some code. The technical debt in this project lays in its complexity: high cyclomatic complexity, too many lines of codes, too many functions or too deep block nesting. Our suggestion will be to redesign the most complex packages (such as `main` , `protocol` library or `model` library) in order to reduce its complexity and improve its performance.

As for testing debt, from the analysis it is evident that even though the project on a whole has a very good code coverage, there is still some room for improvement. However, in general all source files are tested and a number of automatic tests are run on the code when a pull request is triggered and hence there is no major testing debt here. Another potential testing debt that is evident from the analysis is that the framework consists of a large number of css files which are relatively harder to test. But one could easily automate the testing of certain UI features like proper scaling of GUI elements when resizing a webpage, verification of navigations etc. Finally, we also noticed that apart from testing Syncthing at a unit level, the developers have also included few integration tests. However, these tests are mostly aimed at System Integration testing (SIT). The other aspects of the system like security and performance are neither tested nor scheduled for later testing, thereby posing another potential testing debt.

Evolution of Technical Debt

In order to research the evolution of technical debt within the Syncthing project, we will look at how the code evolved over time. The releases along with the changelog will help us understand what happened when and why. Finally, we will look at several issues and pull requests that pay technical debt and discuss them in more detail.

The first release of the Syncthing project was v0.1.0 in December 2013. In the first months, the release cycle was extremely short, sometimes releasing twice in a single day. Four months later, in March 2014, after over 30 releases v0.7.0 was released. The code at the time, as with many new projects was changing a lot. The general architecture was constantly evolving, requiring short release cycles to keep up. Many of the changes were to the documentation in order to keep it in sync with the code itself.

Towards the end of 2014 the code started to stabilize with release cycles slowing down to about once a week. As the codebase grew more stable and many of the basic features had been implemented, it took more and more time to make a meaningful contribution. Contributions became more structured as more contributors joined the project.

Platforms

Since v0.9.6 the `build.sh` script was replaced by `build.go` script for better cross platform support. In the beginning this caused a lot of cross platform issues because no matter how hard they tried to abstract away the operating system, there were differences that turned out problematic. A good example is the allowed characters in filenames which are different on Windows and Linux. This led the maintainers to drop support for Linux specific filenames in favour of portability.

External projects making use of syncthing such as syncthing-inotify, syncthing-gtk, and syncthing-android started cropping up in early 2014 and complicated matters. Changes made to the core now needed to properly reflect in the dependent packages. This was a source of much frustration as it took cross platform to a new level. Users expected these implementations to work even on Android TV. These projects were deliberately placed in separate GitHub projects, to separate concerns of the core project from different frontend implementations.

Releases

Looking at the different releases we can see many bugs are introduced when a protocol version is upgraded. Aside from backwards incompatibility, the protocol is often unstable until it is tested by the public in different scenarios, configurations, operating systems etc. Furthermore, v0.13.3, v0.13.4, and v0.13.5 are interesting releases as they followed shortly after the major release v0.13.0 and were labelled as 'bug-fix release'. They fixed about 10 bugs that were introduced and discovered only after the major release. The distributed nature of the Syncthing project makes it prone to concurrency bugs that are hard to detect by developers.

v0.13.6 was the first release that explicitly mentioned cleaning up code. This so called 'cleanup-release' was first introduced in the section about discussing technical debt. It shows Syncthing's efforts to keep the codebase clean and prevent accumulating interest on technical debt.

Some of the releases are labelled as bug fixes or security fixes, but it is unclear if these problems were introduced as a result of accumulated technical debt. In a complicated distributed system, it is always difficult to cover all possible scenarios, inevitably exposing attack vectors to malicious users. Having an active and engaged community helps fixing these issues as soon as they are detected.

Conclusion

In this chapter we have analysed Synchting by first using different views and models, and finally by looking at the technical debt within the project. We have seen how Syncthing is able to synchronize files between devices, while giving guarantees regarding data loss, security, and ease of use. The organization of the code in layers allows for separation of concerns, and provides a decoupled architecture. This, along with good code quality and collaboration practices results in a healthy codebase, with minimal technical debt. Syncthing is an open source project with a strong core team that provides a suitable alternative to proprietary synchronization technologies.

References

1. Syncthing User Data. https://data.syncthing.net/
2. Semantic versioning. http://semver.org
3. Syncthing Authors. https://github.com/syncthing/syncthing/blob/master/AUTHORS
4. @calmh's comment on tests. https://github.com/syncthing/syncthing/pull/3780#issuecomment-268509136
5. Syncthing Beta Testers. https://forum.syncthing.net/badges/131/beta-tester
6. Syncthing Scribes. https://forum.syncthing.net/badges/121/scribe
7. Syncthing Helpful Members. https://forum.syncthing.net/badges/126/very-helpful
8. Syncthing Relays. http://relays.syncthing.net/
9. Syncthing Manifest. https://github.com/syncthing/syncthing/blob/master/vendor/manifest
10. Syncthing. The Syncthing Goals. https://github.com/syncthing/syncthing/blob/master/GOALS.md.
11. Effective Go. https://golang.org/doc/effective_go.html
12. Separation of Initialization and Construction. Stack Overflow. http://softwareengineering.stackexchange.com/questions/206086/separation-of-construction-and-initialization.
13. Issue Templates. https://help.github.com/articles/creating-an-issue-template-for-your-repository/
14. Nick Rozanski and Eoin Woods. 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.

15. Syncthing Docs. https://docs.syncthing.net/.
16. Syncthing. Source Code. https://github.com/syncthing/syncthing.
17. Transifex. https://www.transifex.com/syncthing/syncthing

Telegram-Web

Kilian Grashoff, Bart Heemskerk, Baran Usta, Michiel Vonk

Delft University of Technology, 2017

Abstract

Telegram-Web is a client-side web-application, which uses the Telegram API to enable instant messaging functionality in web browsers. Its sole maintainer is Igor Zhukov, who started the project as a hobby. The application is built using AngularJS, UI bootstrap and CryptoJS. The project contains a lot of technical debt. One form of technical debt is the lack of a modular structure. Also, most of the code is not covered by testing and almost no documentation is present. A plan is proposed to pay off this technical debt, which will make contributing to Telegram-Web easier.

Introduction

Instant messaging applications offer real-time transmission of text over the internet. They allow for effective communication, because a recipient can almost instantly see a received message. Because of their text-based nature they can also be used in many settings. Telegram is a free, cloud-based instant messaging application with a focus on security and speed. It offers a well defined public API to enable developers to implement their own systems. There are official Telegram clients for various mobile and desktop platforms, and an official web client. All of Telegram's official clients are open source projects.

Telegram-Web (a.k.a Webogram) is the official web client of Telegram. It is hosted on web.telegram.org and also packaged as Chrome and Firefox extensions. Telegram-Web started as a hobby of its creator, Igor Zhukov, and was an unofficial client at first. Later, it became the official web-client for Telegram. In this chapter, Telegram-Web's architecture is analyzed. First its features and stakeholders are discussed. Then its architecture is discussed by means of a context, development, and deployment view. The deployment view is relevant, because the deployment of Telegram-Web is somewhat unconventional. Telegram-Web is further investigated from the internationalization perspective. This perspective is interesting because Telegram-Web supports multiple languages, and a strict procedure is used to guarantee the quality of translations. Finally technical debt in the project is analyzed. Telegram-Web suffers from significant technical debt, so a plan is proposed to pay off this debt.

Stakeholder Analysis

The stakeholders involved in the development of Telegram-Web were analysed in several ways. First, ten recent issues and ten recent pull requests were investigated. Pull requests and issues that had extensive discussion were selected for analysis. Pull requests that contained major changes to the project were also selected. This selection gave information on relevant stakeholders, and how they interact. Secondly, the GitHub contribution statistics were analysed. This second analysis helped to make sure that no relevant stakeholders were missed in the first analysis. The involved stakeholders were categorized as described by Rozanski and Woods. [13]

- **maintainer** and **assessor**: *Igor Zhukov* started the project, and is its sole maintainer. This can be concluded from the fact that he is the only one that merges pull requests and closes issues on GitHub. He also assumes various other roles in Telegram-Web's development (see below). Since he is the sole person who controls what is approved to merge into Telegram-Web, he is regarded as assessor as well.
- **acquirer**: *Telegram Messenger LLP* was founded by Pavel Durov, Nikolai Durov and Axel Neff. [15] Telegram Messenger LLP develops the core Telegram service and its API. Although Telegram-Web is the official web client for Telegram, Igor Zhukov's relationship to Telegram Messenger LLP is unclear. The reason for this is that Telegram Messenger LLP is highly secretive, after its founders left Russia for political reasons. [12] However, Igor Zhukov used to work for the same company as the Telegram founders (VKontakte). [10] For that reason, is assumed that he is either employed by Telegram Messenger LLP, or has close ties to it. Therefore, Telegram Messenger LLP, represented by CEO Pavel Durov, is regarded as acquirer.
- **system administrator** and **suppliers**: due to Telegram's secretive nature, it is unclear who administrates the deployment of Telegram-Web, therefore the system administrator is unknown. It is also unclear who supplies the hardware that runs this deployment, so the **suppliers** are unknown too.
- **communicators**: the analysis of GitHub issues showed that *@stek29* and *@Ryuno-Ki*

reply frequently to issues, therefore acting as communicators. Markus Ra is the press contact for Telegram Messenger LLP, so he acts as communicator as well. The people in the *Telegram Support Initiative* are the final group of communicators. They are volunteers who answer questions by other users using Telegram itself.
- **developers**: the analysis of pull requests and the GitHub contributors graph showed that *@stek29* and *@Ryuno-Ki* are the only people who have recently contributed significant changes, so they are the main active developers besides Igor Zhukov.
- **production engineer** and **tester**: *@Ryuno-Ki* set up a framework for testing and created some initial tests for Telegram-Web, so he is regarded as having both these roles.
- **users**: every user of Telegram-Web is considered a stakeholder. Telegram-Web can be used in four ways:

    1. Chrome app;
    2. Firefox app;
    3. hosted online (http://web.telegram.org and Github pages on https://zhukov.github.io/webogram);
    4. self-hosted.

The following additional types of stakeholders were also identified. The explanation for each type describes why these groups are relevant stakeholders in addition to the stakeholders described above.

- **candidate developers**: *submitters of pull requests yet to be merged or rejected* indicate interest in a feature to the developers and maintainer. Despite their work not being merged, they influence the development of Telegram-Web. This influence is evident in the analyses for pull requests #330 and #1306.
- **reviewers**: reviewers actively influence the development of Telegram-Web by reviewing the work of candidate developers. *@zhukov* acts as reviewer, since he reviews the works by all contributors before deciding if he will merge their work. *@stek29* and *@Ryuno-Ki* also voluntarily review pull requests on GitHub.
- **translators**: *volunteer translation teams*, led by *Markus Ra*, influence Telegram-Web by making the translations used by Telegram. This process will be explained in-depth in the internationalization perspective.
- **maintainers of derivative works**: these maintainers have an interest in the development of Telegram-Web, since they can pull in any work that happens to Telegram-Web into their own work. Since Telegram-Web has many forks, the GitHub network graph functionality could not be used to find derivative works. A script was written, which calculates the number of diverging commits for each repository on GitHub. Two major derivative works were found:
    - hippopogram by @I-hate-farms with his own added features like markdown support. [9]

    - TWebogram by @rubenlagus is a version of Telegram-Web with additional functionality to help with the Telegram Support Initiative. [14]
- **competitors**: competitors are interested in the development of Telegram-Web since they have a competing service. *WhatsApp* and *Facebook messenger* are the two chat clients with the most users worldwide, both are owned by Facebook, Inc. [11] They have a web-version that competes directly with Telegram-Web. *IRC* and *Signal* compete based on specific features that are advertised both for them and Telegram. IRC is known for having a large ecosystem of bots, which Telegram has too. Signal is known for its privacy features, Telegram advertises this also.

People contacted

Igor Zhukov was contacted, since he is the person with the most influence and knowledge about Telegram-Web. Questions were asked about what the best way was to make small contributions to Telegram-Web and how he came to start Telegram-Web. He replied that any contributions would be helpful, but that he did not have suggestions besides the issues on GitHub. He confirmed that he started Telegram-Web as a hobby.

While submitting pull requests with tests (#1355 and #1362), a request to review was sent to @Ryuno-Ki, since he set up the Telegram-Web testing framework. He made some helpful suggestions, which were incorporated in our pull requests.

Context View

This section will discuss the relationships between Telegram-Web and external entities. Telegram itself will be treated as a black box.

System Scope

Telegram-Web is an application built against the Telegram API to allow mobile and desktop users to use Telegram without installing an application. On some platforms (Firefox OS and Chrome OS) this application can even be installed since there are no native apps available on those platforms. The scope of this chapter is limited to Telegram-Web, so the Telegram API is considered an external entity.

History

Messaging has been around even before the Internet by using postal and other services. When the internet came around there were multiple ways to communicate, some looked like sending a letter to each other (email). There were also message boards and public chat channels in which (like IRC) even had private messaging.

Telegram itself is more comparable to either MSN messenger, or more recent mobile applications such as Whatsapp or Facebook Messenger. What these applications offer is a service to communicate with other persons, either alone or in group-chats. Telegram tries to differentiate from this service by providing more secrecy and the ability to create bots, these bots are programs which allow the user to play a game and or provide integration with third party services. [6]

Telegram-Web has mainly been developed by @zhukov. It started as a hobby and since 13-11-2014 it is adopted as the Official web client of Telegram. [8] Over time other contributors tried to help out @zhukov by issuing a pull request with their contribution. As can be seen in table 1 @zhukov is still the lead developer of this application.

Time period	Time period (exact)	Contributors
first week	5 - 12 Jan	only @zhukov
first month	5 Jan - 5 Feb	help from: @paulmillr for setting up the repo and @imangani for a small MIME fix.
first year	5 Jan 2014 - 5 Jan 2015	22 contributors which totaled to 58 commits vs 158 commits by @zhukov
second year	6 Jan 2015 - 6 Jan 2016	25 commits by contributors, 423 commits by @zhukov
remainder	7 Jan 2016 - 31 Mar 2017	21 commits by other contributors and 254 commits by @zhukov

Table 1: commits done to Telegram-Web over time

Context Diagram

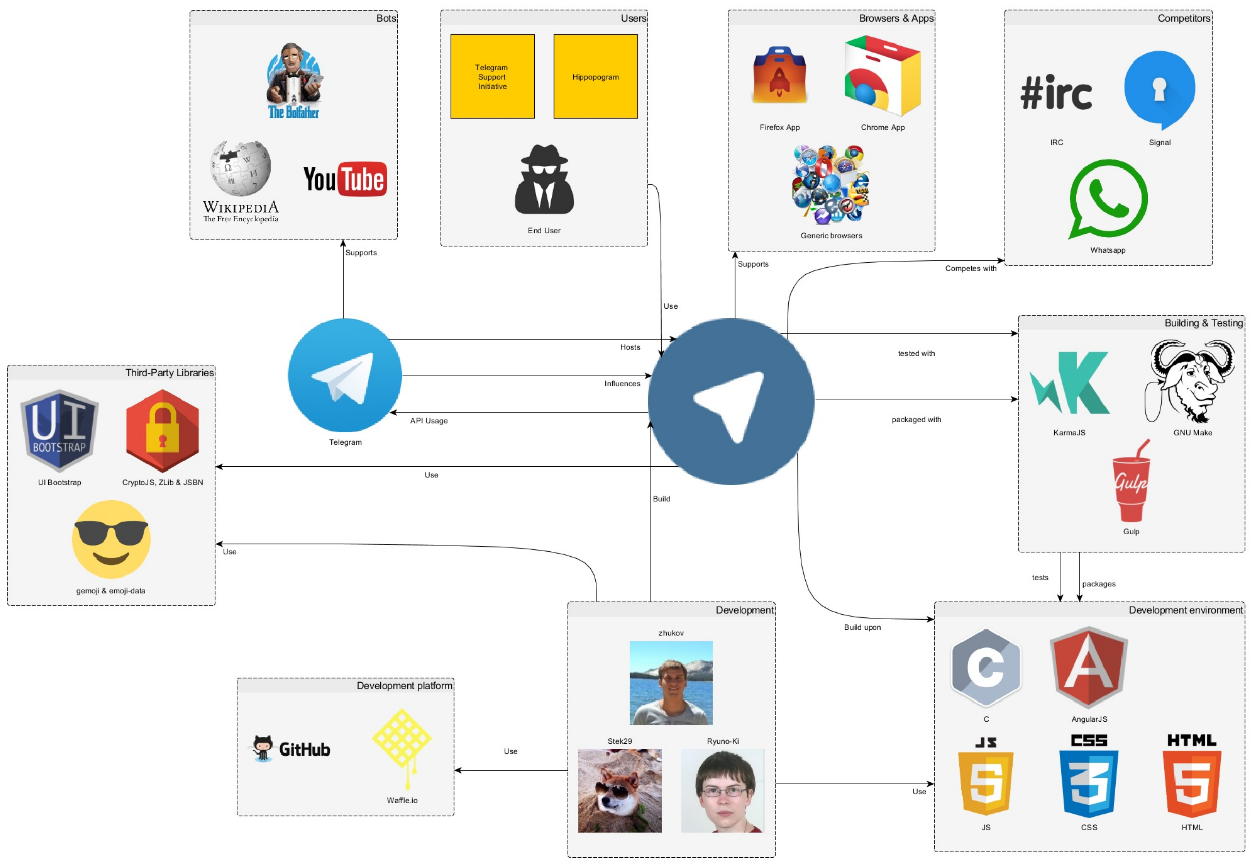


Figure 1: Context Diagram

Context Model

The context-diagram as seen in figure 1 is an overview of the components connected to Telegram-Web. The following subsections will start at the bottom of the diagram, then review the top part and finally the interconnection and reason why Telegram itself is also included.

Development

At the bottom five groups are visible. Together they represent the development of Telegram-Web:

- **third-party libraries:** the third party libraries which can be found at the Telegram-Web GitHub page;
- **development:** the main developers selected from the pull request, issue and fork analysis, as described in the stakeholder analysis;
- **development environment:** the languages used in the project, AngularJS is the framework upon which the whole application is built and thus classified as part of the development environment.
- **development platform:** to aid the development process, a central code and issue location is needed. In this case GitHub is used in combination with Waffle.io to show the

current issue status.

- **building & testing:** some tools are used to build Telegram-Web, by creating a standardized build process using make and gulp no steps can be forgotten

Users and Competitors

Excluding the Bot group the top parts consist of:

- **users:** which consist of the end-users and two special groups which run their own fork: TSI with some additional branding and Hippopogram a derivation with some additional features;
- **browsers & apps:** which shows the cross-browser version and the packaged apps for both Chrome and Firefox;
- **competitors:** which show widely used chat apps with comparable functionality.

Telegram API

The Telegram-Web application cannot exist without the Telegram API, since Telegram-Web is only a client that needs the data stored behind the Telegram API. The external services and bots accessed through the Telegram API are a big influence on the development of Telegram-Web. One example of this is the **Youtube bot**. For usability a thumbnail returned from the bot is embedded into the chat. Telegram-Web has functionality to handle this.

Development View

This section describes the architecture of Telegram-Web. The development view helps to support the development process. Its concerns are common design approaches, module structure, and codeline organization. Each of these concerns will be discussed in this section. [13]

Common design model

Common design approaches and design constraints are helpful to keep the system coherent. Each developer has to follow the AngularJS patterns that are used in the project. Additionally, security is an important concern of Telegram. Therefore, Telegram-Web has to adhere to strict constraints while interacting with the server. [18,19,20]

AngularJS design patterns

Telegram-Web is designed as a Single Page Application and uses the AngularJS 1 framework. AngularJS was specifically developed to make building single page web applications easy, which is probably why it was used for Telegram-Web. It's based on the Model-View-Controller design pattern, which structures an application in three parts:

- **model:** the data (also called state) of the application;
- **view:** the interface of the application, which shows the data in the model. The view is defined in HTML-based Template files;
- **controller:** component that updates the model when the user makes a change to the view. This is the place where business logic such as validation of the user input happens.

There are some additional types of components in AngularJS:

- **directives:** updates part of the view when the model changes;
- **filters:** functions that format data for display to the user;
- **services:** reusable business logic that is independent of views.

Figure 2 shows the (possible) dependencies between these different types of components. An arrow `A->B` indicates that components of type `A` can depend on components of type `B`.

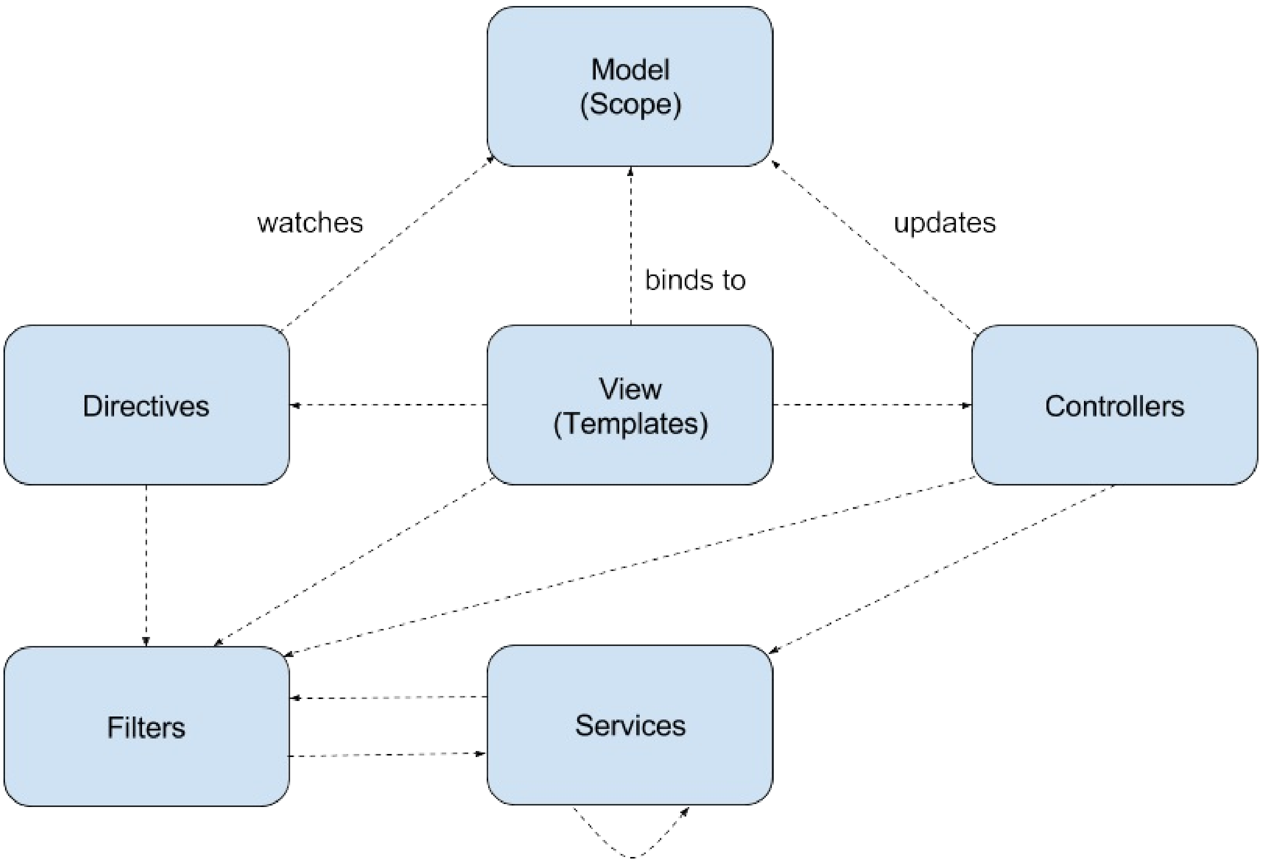


Figure 2: the model of different components of AngularJS

Module structure model

The module structure model defines the organization of the system's source code. It specifies the modules into which the individual source files are collected and the dependencies among these modules. [13]

Figure 3 shows the Telegram-Web modules, as defined using AngularJS modules and explicit dependencies between them. Injected dependencies will be discussed later.

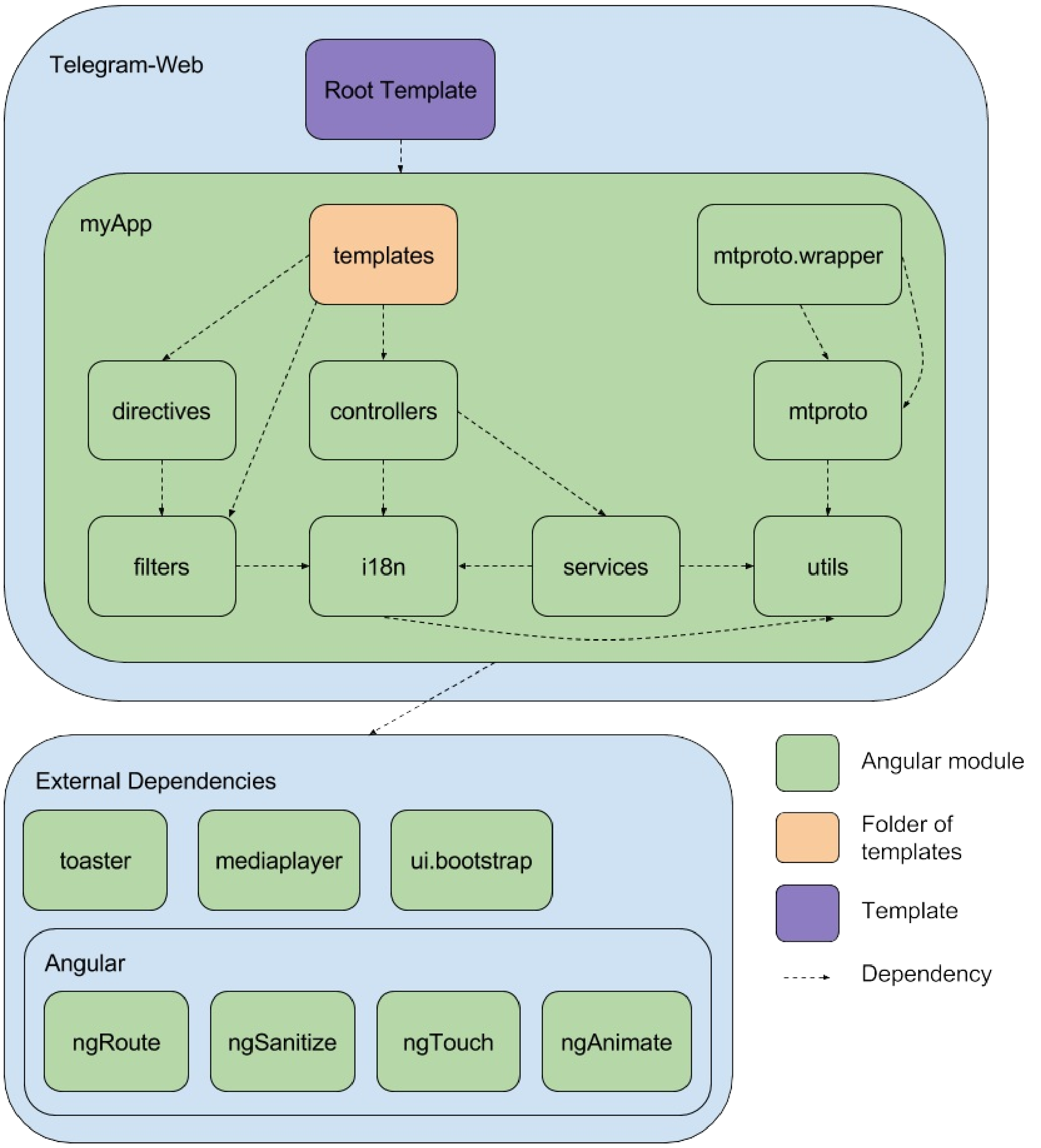


Figure 3: Telegram-Web module structure

The main module of the Telegram-Web is called myApp, and depends explicitly on all modules that are part of it, as well as all external modules. The only functionality that the myApp module itself provides, is to specify the controllers that AngularJS injects for three different URL's: `/`, `/login`, and `/im`. Those three controllers in turn trigger loading of the templates that contain the needed components for the page to be displayed.

Internal Modules

Each of the component types (controllers, directives, filters, services, and templates) function as described in the section on the AngularJS component structure. The recommended way to use modules in AngularJS is to make a module for every individual feature and reusable component in the application. [2] However, Telegram-Web only has a single module per component type except for the following separate modules:

- `i18n` is the internationalization module. It provides functionality to show the interface in one of the languages that Telegram-Web supports.
- `utils` provides several functionalities. It mainly consists of services to write, read and transfer data to local storage and over the web.
- `mtproto` implements the low-level functionality of the MTProto protocol that Telegram-Web uses to connect to the Telegram-API.
- `mtproto.wrapper` uses the functionality provided by MTProto to provide high-level functionality that is used in other modules.

External modules

The myApp module depends on external modules, that can be used by any of the modules within myApp. Telegram-Web depends on the following external modules:

- `Toaster` provides a way to show notifications on the webpage itself;
- `Mediaplayer` provides an easy way to play media from inside AngularJS applications, by wrapping the <audio> and <video> HTML tags in a directive;
- `ui.bootstrap` implements the Bootstrap user interface elements in AngularJS directives;
- finally `ngRoute`, `ngSanitize`, `ngTouch` and `ngAnimate` are modules for specific functionalities that AngularJS provides.

Injected dependencies

The explicitly defined dependencies of AngularJS modules only determine the order in which AngularJS constructs different components. These components are free to specify arguments that they need outside of their module, which are injected by AngularJS. This means that the dependencies as listed in figure 3 are far from complete, in fact almost all modules in Telegram-Web reference most of the other modules. Figure 4 is an overview of injected dependencies between components in internal modules.

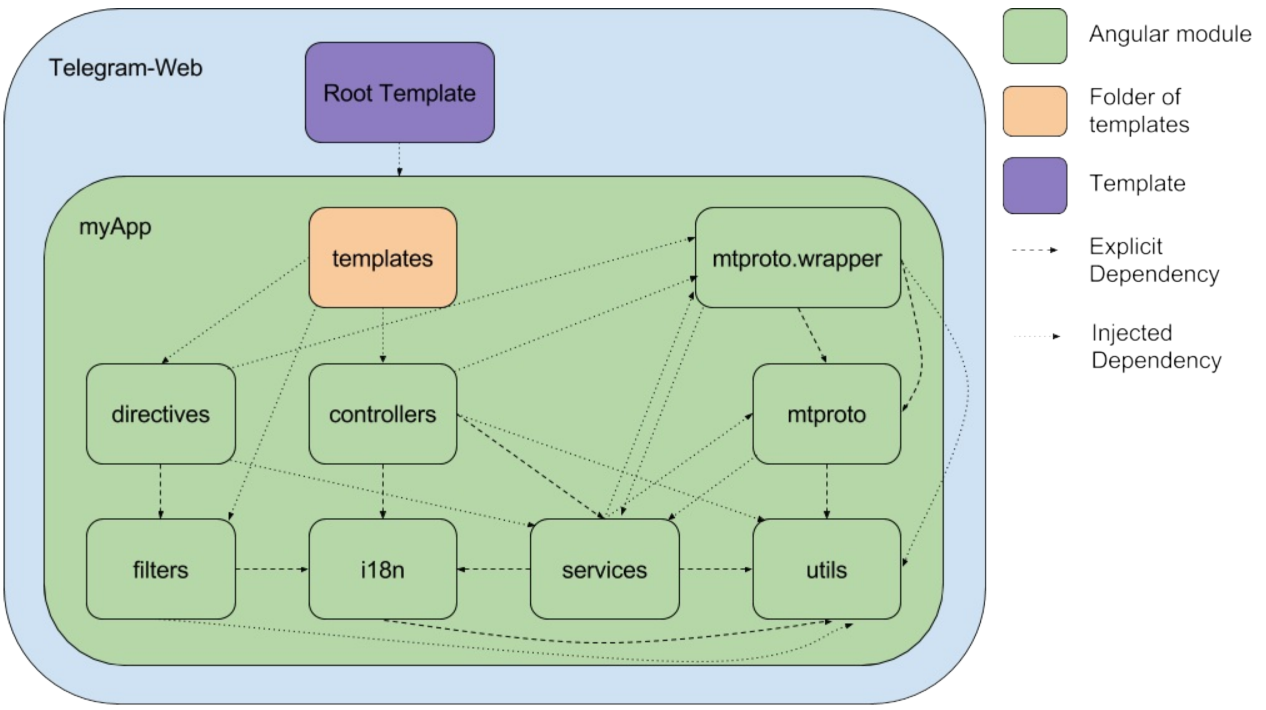


Figure 4: Telegram-Web module structure with injected dependencies

Layering rules

Because the dependencies between modules in Telegram-Web are unmanaged there is no clear grouping of modules into layers possible. Since there are no layers, there are also no layering rules.

Codeline model

The codeline model specifies the organization of files and the interaction with external tools. First, the directory structure and configuration files are specified. Then, the tools that are used for building and testing are documented. Finally, the strategy for source code management is explained.

Directory structure and Configuration

The root folder contains the project files such as:

- `package.json` , which gives information about the package/application;
- `gulpfile.js` , which allows building a distributable version;
- `LICENSE` , `README.md` , and `CONTRIBUTING.md` , which give repository information and

explain how someone can contribute.
- `Dockerfile`, specifies how the application runs a Docker container.

The folder structure then boils down to:

- `.tx` Transifex configuration;
- `app/`, application files such as a manifest and the start file: `index.html`;
- `app/fonts`, Google fonts;
- `app/img`, project image files;
- `app/js/lib`, mainly non AngularJS files;
- `app/js`, mainly AngularJS files;
- `app/less`, LESS Stylesheet;
- `app/nacl`, *Native Client* C files, which compile to a `pexe` file;
- `app/partials`, HTML views;
- `app/vendor`, third-party libraries;
- `scripts`, external scripts for downloading Google fonts, country name translation and emojis;
- `test/unit`, unit tests.

All configuration parameters and constants must be put in the `config.js` file. When forking Telegram-Web an `API_id` and `API_hash` must be retrieved at https://my.telegram.org.

Building and testing

The Telegram-Web project is based upon JavaScript and AngularJS. The package manager `yarn` is used to download all dependencies. Recently the test runner `KarmaJS` and test framework `Jasmine` were added. Using the configuration in `karma.conf.js` the tests can be run by executing `gulp karma-single` for a single run or `gulp karma-tdd` for continuous testing. In continuous testing mode `KarmaJS` watches for file changes and then reruns the test.

The building process is based on `Gulp` and has a list of tasks, specified in `gulpfile.js`. These tasks are named according to the goal of the task (e.g. copy-images). There is also a `Makefile` which can be used to call several *Gulp* and *Transifex* tasks together to perform common actions. For example, using the command `make package` will execute gulp to build the software, remove unnecessary files, and package it as a zip file.

Source management

The Telegram-Web repository is located on GitHub. The main developer (@zhukov) uses one branch for software development and one for the GitHub pages hosted version.

When users want to contribute it is recommended to open an issue for the proposed changes before starting development. This helps to prevent unnecessary effort. Telegram-Web uses the forking workflow. [5] In the forking workflow the user forks the repository and makes their changes in a separate branch. When they have completed their work, they can request the code to be merged into the main repository by issuing a pull request. After a pull request is issued, Igor Zhukov reviews it. He decides whether to merge the contribution, request changes to it, or reject it.

Deployment view

Althought the name of Telegram-Web would suggest it's a web application most of the processing power is needed at the Telegram API, which is out of scope for this chapter. Therefore only the packaging and distribution of the client-side application will be discussed.

Packaged Apps

All versions of Telegram-Web can be treated as a packaged app, through a market place or from the Webclient. The package is the same for different platforms, although they look at different files to execute the application:

- `webogram.appcache` : hosted version of Telegram-Web;
- `manifest.webapp` : Firefox Packaged App;
- `manifest.json` : Chrome Packaged App.

The `webogram.appcache` is a list of files which are downloaded to the users browser cache. The other two are comparable to each other, the Firefox version has more options (custom permission messages and activity references).

Dependencies

In the packaged manifests permissions are given to give the user control over the app. The Chrome/Firefox app manifest asks permission on installing, while the hosted version asks permission whenever needed. The manifest contains the following permissions:

- `notifications` , `push` , and `desktop-notification` : these permissions are needed to get notified on events like receiving a message;
- `webview` and `fullscreen` : the webview is needed to "embed guest content", since the app is a package webpage, and must be able to go fullscreen to give the user a full app experience;
- `{"fileSystem": ["write"]}` , `storage` , `unlimitedStorage` , and `device-storage:{music,pictures,sdcard,videos}` : to store received images;

- `contacts` : to start a chat with a known contact.

Most of these permissions are soft dependencies. For example: a user can only send or receive a picture if they access to their storage. The same holds for contacts. In addition, if a user denies push access he/she cannot receive notifications.

Throughput needed

Since the packaged apps are served through either the Chrome or Firefox Marketplace the only infrastructure needed, is for the hosted variant. The webcache consists of the files specified in the webogram.appcache. These files are downloaded from the server (if caching is allowed) when a user starts the app for the first time or when the application is updated. All packaged versions are the same and consist of 5,65 MB of data. When removing the manifests and files not specified in the `webogram.appcache` 5,44 MB is used, divided over 84 files.

Two better known hosted versions of Telegram-Web are:

- Telegram-Web
- Github Pages - Zhukov

For the Github pages the NGINX webserver is used with a Content Delivery Network (CDN) to distribute load across the globe. The Telegram-Web headers show that the webserver used is NGINX 1.6.2. a server which, depending on hardware and configuration, can handle over 500.000 requests/sec. [1] It is more likely that the server runs out of bandwith before it reaches its requests threshold. Using the network bandwidth available and dividing it by the size of a package, a rough number of users simultaneous updating is calculated. The results of this calculation are shown in table 2.

Speed	Updates / second	Updates / Hour
1Gb/s	23	82800
10Gb/s	235	846000

Table 2: number of updates possible per network bandwith

It is hard to judge whether the current infrastructure is sufficient to allow all users to update in a timely fassion. No one besides the hosting party knows what the current demand is during updates. Using a CDN like the GitHub pages does mitigate this problem.

Internationalization Perspective

Rozanski and Woods [13] describe the internationalization perspective as "the internationalization perspective ensures the system’s independence from any particular language, country, or cultural group". The Telegram app is used by milions of users all around the world. That's why the internationalization perspective played a significant role designing the system. This section discusses how the internationalization perspective affected the system.

Internationalization of Project

As stated earlier, the project is hosted by GitHub and a number of developers contributed who are from all around the world such as Argentina, India, Taiwan and many more. [7] To enable people across the world to contribute easily the language used in the code, in the discussions under issues, and in the PR's is English.

Localization

Telegram Messenger LLP aims to serve all the users in the most convenient way for them. This is why they aim to provide the service in their own language. Currently Telegram-Web has six different languages support. The default language of the app is English. However, when the app is visited and a language has not been set before the browsers default language is checked. If languages is supported, the app is configured and shown in this language. Besides these supported languages some language translations are under development. For the translation Transifex, which will be discussed in detail below, is used. Telegram-Web also supports different accents in some languages. That is why the necessary characters that are used in those accents are defined manually in 'config.js' file as 'Config.LatinizeMap' variable.

Transifex

Transifex is a third party service that provides a collaborative localization platform. It is used for each of the client projects of Telegram. [21] This service has been used for translations since Telegram-Web became the official web app for Telegram. The project is managed by both Igor Zhukov and the Telegram Head of Support, Markus Ra. In order to add support for a new language in official Telegram-Web, it has to be approved and translated completely. There are currently seven languages that are being translated by volunteers selected by Telegram Support Team. These languages are for example: Turkish, Korean and Chinese.

Translation Guideline

As a policy Telegram relies on volunteers for many services to keep the Telegram free. [22] Translation is one of these services. The Telegram Messenger LLP expects each client's translation to follow similar principles to provide a pleasant service to the users. Translations must be:

- consistent: should be consistent in all platforms;
- natural: instead of translating word by word, it should reflect the culture;
- default: similar to the language that other popular apps uses;
- beautiful: should look like the app is built in that region;
- looks good: words must not cause any problem with UI.

Technical Debt

Technical debt is the difference between how a project is run currently and how it should be run in an optimal way. [17] In this section the technical debt of the Telegram-Web project will be dissected and discussed. The debt will be discussed in three forms: architectural debt, testing debt, and documentation debt. The evolution of the technical debt will be examined and a solution/working method is proposed to pay off the debt.

Architectural Debt

Architectural debt occurs when the architecture of a software system is not optimized, making development more difficult. The Module Structure Model shows that Telegram-Web suffers from architectural debt. This is because most of the code is not split up into modules. Instead Telegram-Web contains only a few very large modules. Examples are the files `controllers.js` (2796 LOC) and `directives.js` (1927 LOC)[22], both part of the MyApp module. This is not how the AngularJS module system is meant to be used. The system's intention is that every component that implements a feature is made into an individual module. [2]

Comparing the current master to the oldest version of the Telegram-Web source on GitHub shows that most of the module structure remained unchanged. Because the project was a lot smaller back then, this was not as much of a problem as it is now.

Splitting the large modules into smaller modules would not fix the dependency management, because AngularJS does not impose restrictions on module connectivity. Migrating to Angular 2 would solve a part of that issue because it is more strict and offers more tools on module management. [3]

SOLID Violations

Both the Single-responsibility principle and the Open-closed principle are violated due to the lack of module structure. [16] The lack of module structure leads to a lack of consideration of dependencies, causing many components to depend on many others. Because there are no modules with well-defined interfaces, all code is open to modification. It is also not possible to depend on the functionality of a single module, meaning that the code is closed to extension.

Because Telegram-Web is not programmed using object-oriented principles, the Liskov substitution principle, Interface segregation principle and Dependency inversion principle do not apply. [16]

Testing Debt

Code needs to be tested in order to assure the quality and code needs to conform to a style in order to improve the readability of the code. Whenever code is not tested or does not conform to a code style it creates testing debt. The testing debt of Telegram-Web will be examined and discussed below.

Code Testing

Libraries for testing JavaScript code are present since pull request #1293. Because this pull request is merged recently, it lacks proper coverage by unit testing or any other testing as can be seen in table 3.

File	% Stmts	% Branch	% Funcs	% Lines
js/	7.51	0.83	1.98	7.52
controllers.js	7.09	1.31	6.47	7.12
directives.js	4.65	0	0.46	4.66
filters.js	9.7	16.67	17.65	9.7
services.js	7.93	0.12	0.77	7.93
templates.js	100	100	100	100

Table 3: Code coverage since latest addition of tests. templates.js is generated by gulp and is fully tested via testing directives.js

Table 3 shows that the project doesn't have any UI or End-to-End testing, because no testing framework for these kind of tests are added to the project. It is argued in PR #1293 that most of the code is tied to UI. Testing UI would enable developers to check if UI changes still conform to certain guidelines, like the one that may be enforced on the main developer by Telegram. The exact guidelines are unclear since none were found. However, based on issue #1357, it is more likely that style is based on the main developers best judgement.

Code Style

@zhukov added StandardJS to the project on the 28th of June, 2016, saying "*for now it doesn't pass well, but that's a start*". However, it is not strictly required that contributions do not add StandardJS errors to the code. The result is a large amount of errors in several files: 1145 errors in ten non-library files and 24414 errors in library files. Most of the later errors are found in app/js/lib/config.js: 23506 errors, most of which are spacing errors and the usage of double quotation marks for strings.

Documentation Debt

Documentation is a vital part of software development. It increases the quality of the software by helping developers to follow the same rules. This also increases the maintainability of the software.

There is no documentation that describes the architecture of Telegram-Web. There is also very little technical documentation with information about the code. There is some info about localization. Another existing document which can be considered technical documentation is the projects GitHub page.

However, there is no documentation associated with the source code. For example there is no document on the data structures used or explanation of the complex functions. AngularJS provides an example of how JSDoc can be used for documentation. [4] However, the Telegram-Web code contains no documentation comments, so no documentation can be generated using these tools.

Evolution of Technical Debt

For checking the evolution of technical debt the CodeClimate CLI tool is run on branches over a period of time, starting at commit 160 and increasing by 160, as can be seen in table 3. These are saved as JSON files which can then be processed by any common scripting language, in this case PHP.

Commit Hash	By	Date	Message
8743a5d	*Bart Heemskerk*	2017-02-21 17:06:13 +0100	Removed npm libraries from Code Climate Testing
5cf067a	*Igor Zhukov*	2016-06-13 15:23:25 +0300	Added tooltip
2e994c1	*Igor Zhukov*	2015-11-29 21:12:40 +0300	Updated changelog
3c55bed	*Igor Zhukov*	2015-07-10 19:36:24 +0300	Improved mobile UX
a317a5b	*Igor Zhukov*	2015-03-19 02:44:49 +0300	Bump to 0.4.1
17e79e7	*Igor Zhukov*	2014-12-28 20:54:42 +0100	disabled own fonts
6653fe7	*Igor Zhukov*	2014-10-27 20:31:00 +0300	Fixed $timeout
441e7ef	*Igor Zhukov*	2014-09-15 14:46:11 +0400	Fixed search messages bugs
b996717	*Igor Zhukov*	2014-06-09 20:45:32 +0400	Test cache manifest update
b9b7c11	*Igor Zhukov*	2014-03-19 15:25:53 +0400	Update Makefile to support gulp publish
e999975	*Igor Zhukov*	2014-01-23 17:49:24 +0400	File download improvements

Table 3: Evaluated commits

CodeClimate

CodeClimate is a code analysis tool which has a lot of different configuration and so-called engines. Each engine can test the code for a different aspect. Based on the number of issues the engines finds in a code project, that project will get a code between 0 and 4, with 4 meaning that the project has no issues.

CodeClimate was configured to run three engines over each commit:

- **CSS-Lint**: which validates css;
- **ES-Lint**: which is a pluggable Javascript linter;
- **Fixme**: which checks the comments for words indicating a 'to do' like *TODO*,*BUG*,*FIXME* or *XXX*.

Graph

The result of running CodeClimate on the selected ten commits can be found in figure 5. On the horizontal axis of this graph the number of commits is shown instead of time. This is to take into account that there were more active and less active periods in the development period.

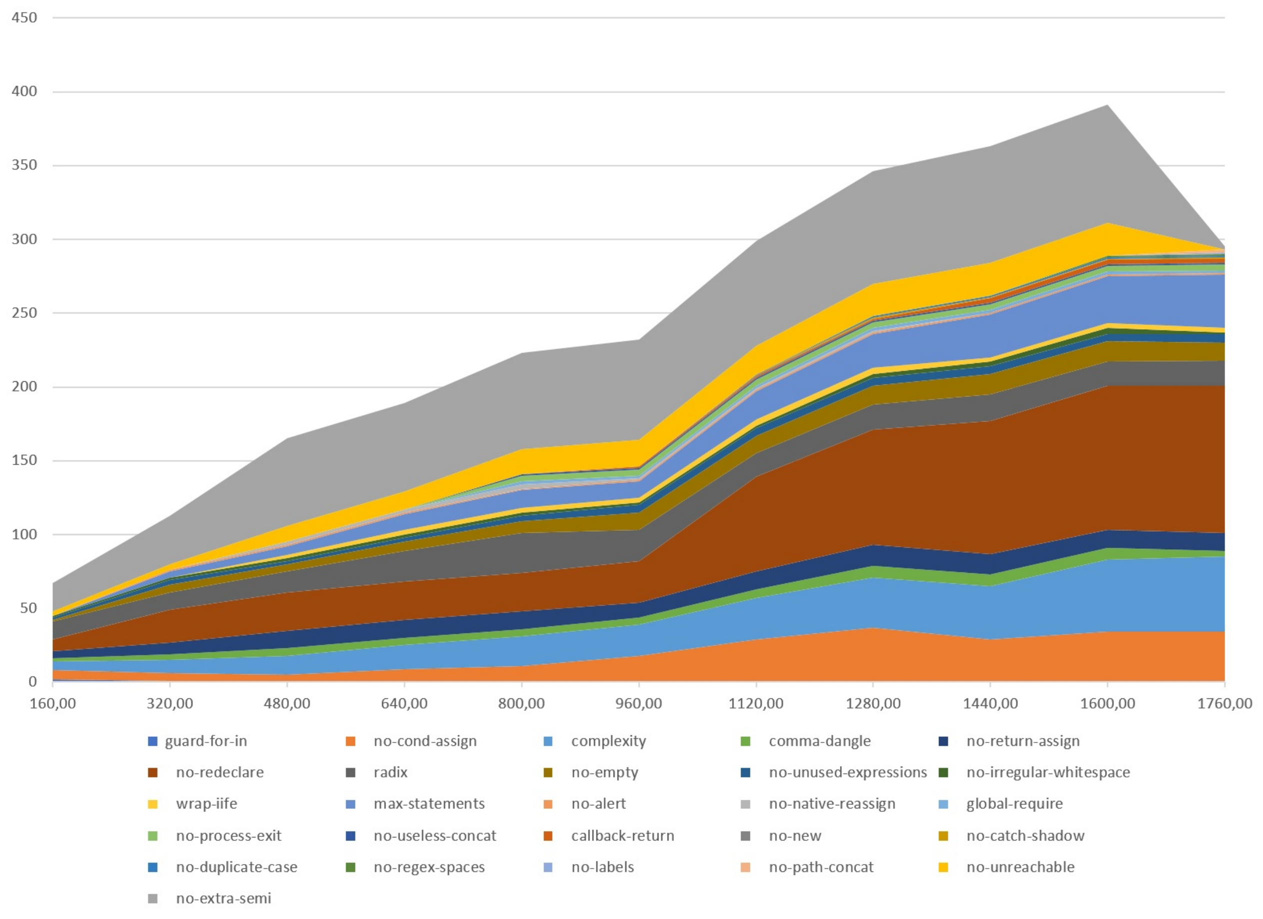


Figure 5: Evolution over time, sorted by ES-Lint and CSS-Lint errors

Proposal for paying off technical debt

In order to pay off the technical debt discussed in this section, the following steps are proposed:

1. **Improve test coverage**: as it stands there is not enough test coverage to guarantee that the functionality will remain the same when drastically changing the architecture. This step has been initiated by some contributions, which have added seven tests for the controllers.
2. **Split larger files into dedicated, smaller files**: the project consists of a few, large files, each file representing one module of the system (like controllers.js and directives.js). By splitting these modules into smaller modules, where each module has its own file, the project will be more comprehensible. Each file can also get its own dedicated

documentation.
3. **Refactor Code smells**: CodeClimate has found several code smells, like duplication, over-complex functions and functions with too many statements. Refactoring the code to eliminate these smells makes the code more maintainable.
4. **Migrate to Angular 2**: Angular 2 has better support for managing the architecture and dependencies of the modules than AngularJS. Migrating to Angular 2 will improve maintainability, because it has stronger reinforcement of architectural principles (usage of modules).
5. **Add continuous intergration**: at this point a CI service should be linked to the project, to assure no new technical debt will be introduced.

Conclusion

Telegram-Web was studied in this chapter by analysing its architecture through several views and perspectives. The development view shows that Telegram-Web lacks a good module structure. Further analysis of technical debt revealed that a lack of documentation and testing are also issues for this project. These issues undermine the development process by slowing down developers who want to add features or fix bugs. They also discourage new developers that want to contribute to the project. The cause of this debt can be attributed to the evolution of Telegram-Web, which started as an unofficial client. Because the end-user does not notice the debt directly and due to practical constraints, fixing it is a low priority for the maintainer. However, paying some of this debt would help Telegram-Web, by making it significantly easier for new developers to contribute to the project. A plan is proposed to help start this process. Several contributions have been made to implement the first steps.

References

1. 500,000 Requests/Sec – Modern HTTP Servers Are Fast, https://lowlatencyweb.wordpress.com/2012/03/20/500000-requestssec-modern-http-servers-are-fast/. Accessed 19-03-2017.
2. AngularJS Developer Guide: Modules - Recommended Setup, https://docs.angularjs.org/guide/module#recommended-setup. Accessed 06-03-2017.
3. AngularJS to Angular Quick Reference, https://angular.io/docs/ts/latest/cookbook/a1-a2-quick-reference.html. Accessed 29-03-2017
4. AngularJS: Writing documentation, https://github.com/angular/angular.js/wiki/Writing-AngularJS-Documentation. Revision 21
5. Atlassian: Forking Model, https://www.atlassian.com/git/tutorials/comparing-workflows#forking-workflow. Accessed 06-03-2017.

6. Bots: An introduction for developers, https://core.telegram.org/bots Accessed 02-04-2017
7. Contributors list, https://github.com/zhukov/webogram/graphs/contributors. Accessed 01-04-2017.
8. GitHub Compare README.md, https://github.com/zhukov/webogram/commit/05a0ddca0e070216933df81cbaa35c42f8237bee Accessed 31-03-2017
9. I-hate-farms/hippopogram GitHub repository, https://github.com/I-hate-farms/hippopogram. Accessed 27-02-2017.
10. LinkedIn Profile Igor Zhukov, https://www.linkedin.com/in/igorzhukov/. Accessed 27-02-2017.
11. Most popular mobile messaging apps worldwide as of January 2017, https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/. Accessed 27-02-2017.
12. Once Celebrated in Russia, the Programmer Pavel Durov Chooses Exile, https://www.nytimes.com/2014/12/03/technology/once-celebrated-in-russia-programmer-pavel-durov-chooses-exile.html. New York Times, 2014. Accessed 27-02-2017.
13. Rozanski,Nick and Woods,Eoin. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
14. rubenlagus/TWebogram GitHub repository, https://github.com/rubenlagus/webogram. Accessed 27-02-2017.
15. Russia's Zuckerberg launches Telegram, a new instant messenger service, http://www.reuters.com/article/idUS74722569420130830. Reuters, 2013. Accessed 27-02-2017.
16. SOLID principles, https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design. Accessed 29-03-2017
17. Technical Debt, https://github.com/delftswa2017/course-info-2017/blob/master/slides/04-technical-debt.pdf. Accessed 29-03-2017
18. Telegram API: authentication guide, https://core.telegram.org/api/auth. Accessed 06-03-2017.
19. Telegram API: security guideline, https://core.telegram.org/mtproto/security_guidelines. Accessed 2017-03-06.
20. Telegram API: MTProto protocol, https://core.telegram.org/mtproto/. Accessed 06-03-2017.
21. Telegram Translation Repositories at Transifex, https://www.transifex.com/telegram/public/. Accessed 01-04-2017.
22. Telegram Translation Guideline, https://core.telegram.org/translating_telegram. Accessed 01-04-2017.
23. Test coverage of Telegram-Web,

https://codeclimate.com/github/bartist/webogram/coverage. Accessed Build #7

24. Used 3rd party libraries by Telegram-Web, https://github.com/zhukov/webogram/blob/master/app/vendor/README.md. Accessed 06-03-2017.

Visual Studio Code

By Daan Schipper, Julian Faber, Rick Proost and Wim Spaargaren.
Delft University of Technology

Visual Studio Code is a lightweight open source text editor developed under Microsoft and can be contributed to through the GitHub repository `vscode`. Extra functionality for Visual Studio Code is provided by means of extensions, which can also be developed by third party developers. This chapter gives an overview of several software architectural views and perspectives, and describes the structure of the project. First the stakeholders of Visual Studio Code are discussed and the results of an interview with one of those stakeholders is shown. After this, the people, system and other external entities with which Visual Studio Code interacts is detailed. Next, the software development process is described and the functional elements of Visual Studio Code are examined. Moreover, the performance of Visual Studio code is analysed, and lastly the included features of Visual Studio Code are discussed.

Table of contents

Introduction

This chapter will give an overview on the architecture, structure, workflow and testing in the development environment of Visual Studio Code. Visual Studio Code is a new type of tool that combines the simplicity of a code editor with what developers need for their core edit-build-debug cycle. Visual Studio Code provides comprehensive editing and debugging support, an extensibility model, and lightweight integration with existing tools. Visual Studio Code is updated monthly with new features and bug fixes. It can be downloaded for Windows, Mac and Linux on the Visual Studio Code's website. The `vscode` repository is where developers can contribute by adding issues or making pull-requests. Visual Studio Code is developed and made open-source by Microsoft, because there was no lightweight alternative provided by Microsoft for their more complex, fuller featured IDE, Visual Studio. Visual Studio Code therefore comes without compilers to stay lightweight, but can still utilize them by installing extensions. In this chapter, the Visual Studio Code environment is described in several different perspectives.

Features

This section gives an overview of the different features of Visual Studio Code. In Figure 1 an overview of the different features of Visual Studio Code can be found.

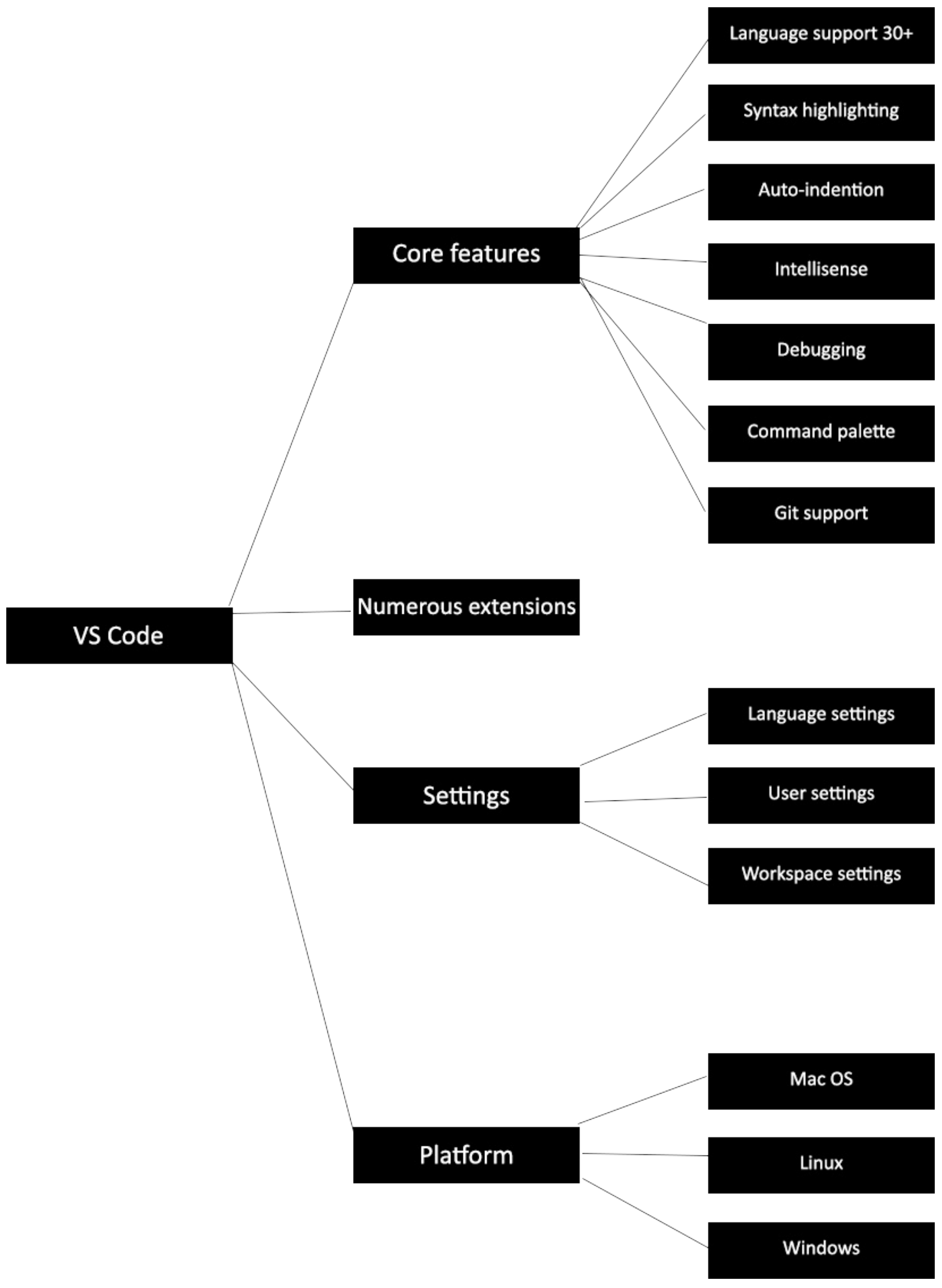
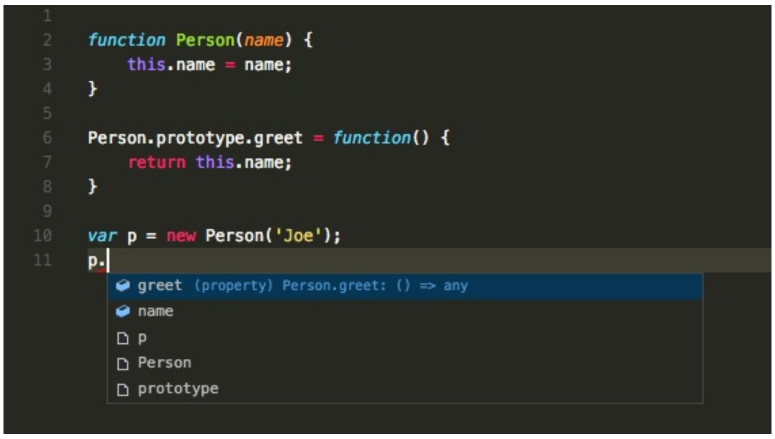


Figure 1: Visual Studio Code feature overview.

Out of the box, over 30 different programming languages are supported. For these languages Visual Studio Code provides syntax highlighting, auto-indention and code completion by using IntelliSense. IntelliSense includes a variety of code editing features such as code completion, parameter info, quick info and member lists which can be seen in Figure 2.



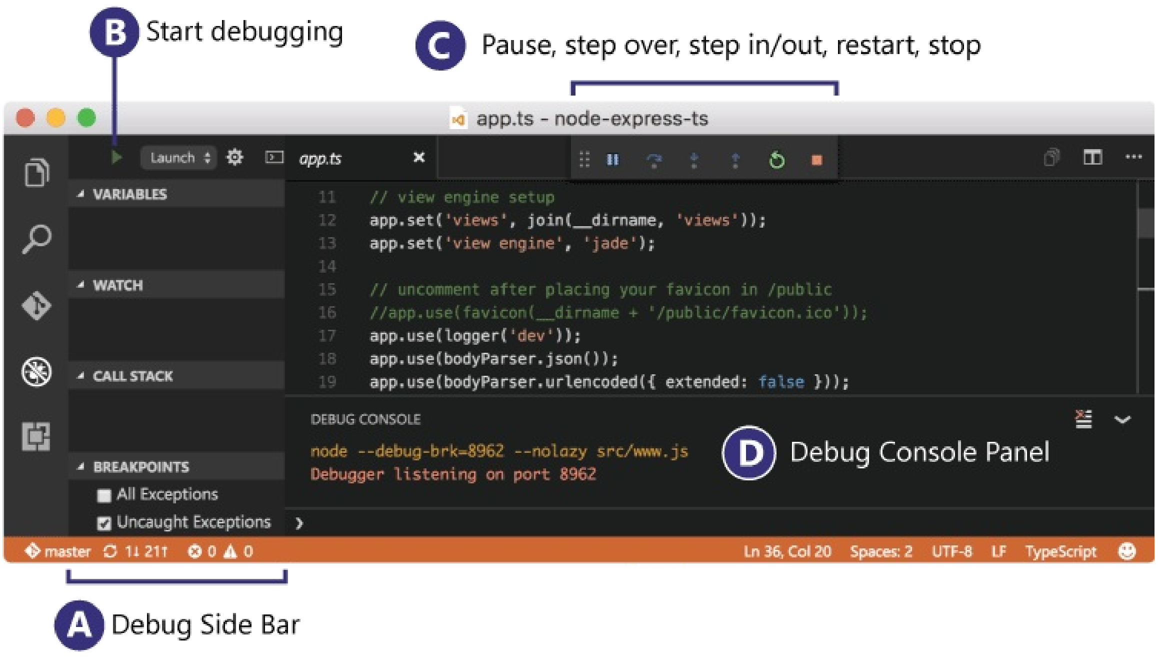
```
function Person(name) {
    this.name = name;
}

Person.prototype.greet = function() {
    return this.name;
}

var p = new Person('Joe');
p.
```

Figure 2: IntelliSense auto completion in Visual Studio Code (source).

Visual Studio Code also provides an interactive debugging tool, which makes users able to step through source code and inspect variables for example. The debug screen can be seen in Figure 3.



```
// view engine setup
app.set('views', join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

```
node --debug-brk=8962 --nolazy src/www.js
Debugger listening on port 8962
```

Figure 3: Debug support in Visual Studio Code (source).

Visual Studio Code also has integrated Git support, for the most common git commands. In Figure 4 an overview of the git support is shown.

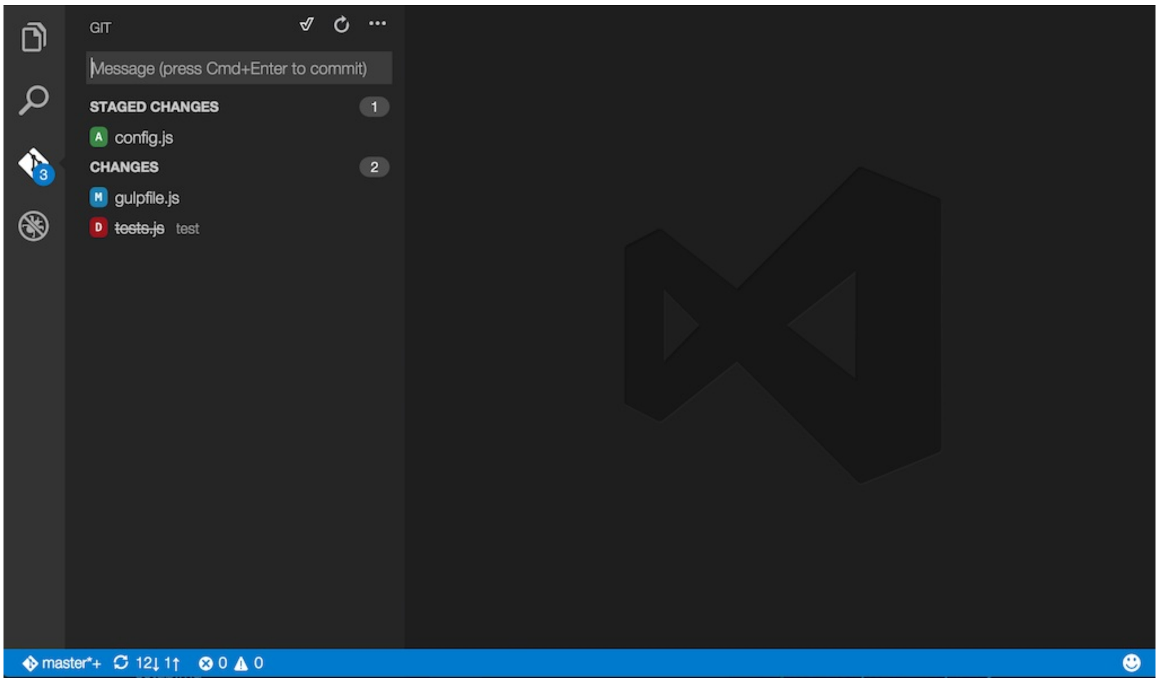


Figure 4: Visual Studio Code git integration (source).

Third party developers can also write extensions for Visual Studio Code to provide extra functionality to the core program. These extensions can be installed by users through the Extensions Marketplace.

Stakeholders

Stakeholders are the people, groups or entities with an interest in an organization. Visual Studio Code is a large open-source project in which many stakeholders are involved. Rozanski and Woods [1] classify multiply stakeholders, the most important for Visual Studio Code are described below.

Developers

Developers are defined as the stakeholders who construct and deploy the system from specifications (or lead the team that do this). The development teams located in Zürich and Redmond are the main developers of Visual Studio Code. These are the developers which can be assigned to issues and merge pull requests on the Microsoft Visual Studio Code GitHub repository. The developers of Visual Studio Code try to process the issues on GitHub, which improves the program with new features or bug fixes.

Users

Users are defined as the stakeholders who define the system's functionality and ultimately make use of it. Users of Visual Studio Code are software developers. The interest of users in Visual Studio Code is that it is a lightweight tool, and is free to use. Visual Studio Code provides the simplicity of a code editor with what developers need for their core edit-build-cycle. Users can influence the system by requesting new features and reporting bug fixes on GitHub.

Support staff

Support staff is defined as the stakeholders who provide support to the users for the product or system when it is running. Visual Studio Code provides support to users in four different ways. Coding questions can be asked on Stack Overflow. Bug reports and new feature requests can be done via issues on GitHub. For other feedback Twitter is used and it is also an option to send an e-mail with comments or questions. The different types of support can be accessed through the following links:

- Stack Overflow
- GitHub
- Twitter
- opencode@microsoft.com

Integrators

In this section the Visual Studio Code workflow is described. It explains how newly written code gets placed in production, whether it is written by a Visual Studio Code developer, or a contributor. It also gives insight on how large software projects can handle overhead of having a lot of developers and contributors.

Integrators are the ones that decide whether new code, or changes to old code, can be merged with the actual code in production. The tasks and challenges which these integrators have, are therefore analyzing pull requests and review them by making comments or even suggestions. When code is approved by the integrator, or multiple integrators, it can be merged to production code. Often the integrators also have a lot of involvement in the planning and road map, so they prioritize issues at hand, but also discuss new features. In short, integrators will deal with importance, as well as feasibility of issues, coding standards, such as commenting and naming conventions, and make sure code is in line with software architecture or discusses refactoring if needed. Also, large features that influence the architecture of the code base, or user interface, are only assigned to the development team.

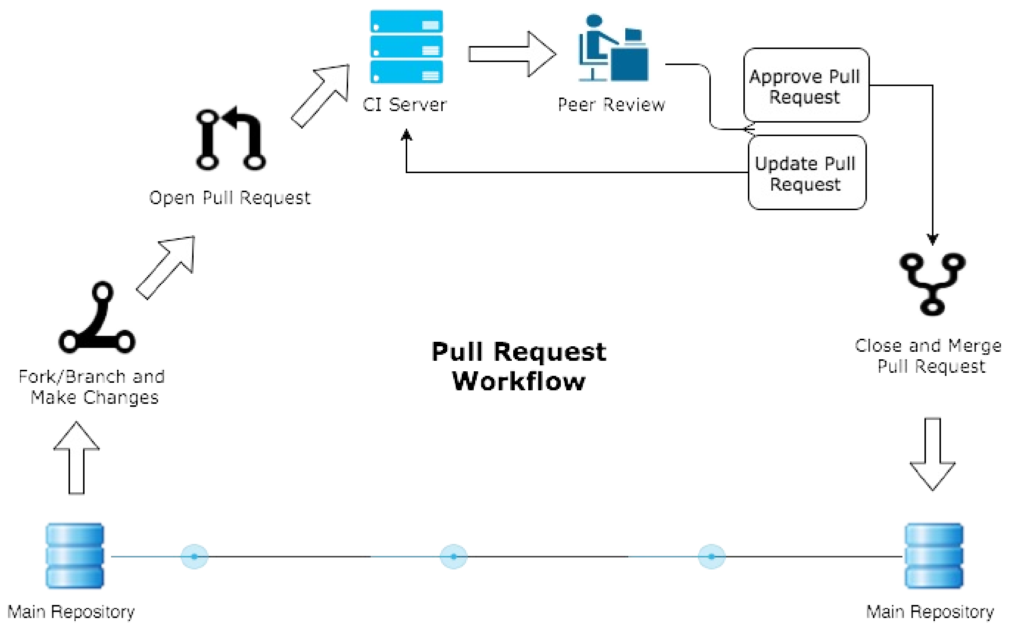


Figure 5: Pull-request workflow (source).

Figure 5 depicts the actual workflow of adding newly developed code to the project.

Integrators Visual Studio Code

It is hard to really determine every integrator in the Visual Studio Code team, because everyone in the team can be assigned as reviewer. Although everyone of them can review, not everyone of them can integrate or in other words merge. The most interesting part found during the pull request analysis is that there are different integrators for different parts of the system. The Visual Studio Code team also automated the assigning process of integrators to pull requests by having a bot analyze the history of the changed files. The integrators then spend 2-3 hours a day to give advise or merge pull requests, told by @egamma in the interview. Coding guideline and continuous integration are also used in the process of reviewing, these are discussed further in the chapter.

Feasibility

Integrators of Visual Studio Code accept small changes from the community fairly quick. The integrators also warn contributors when they are in over their heads, or just may need to invest a lot of their time if they continue on the selected issue. The bigger features and tasks are at request in Visual Studio Code, and normally only distributed to the Visual Studio Code team. The integrators at Visual Studio Code mostly have an idea on how a solution must be to fit in the architecture, so the integrators will also deny workarounds and tell you upfront when another bigger solution is expected.

Context view

The context view of a system defines the relationships, dependencies, and interactions between the system and its environment. The environment in the context are the people, the systems, and external entities with which the system interacts [1]. The context view diagram in Figure 6 shows the identified entities and in what way they relate to Visual Studio Code and each other.

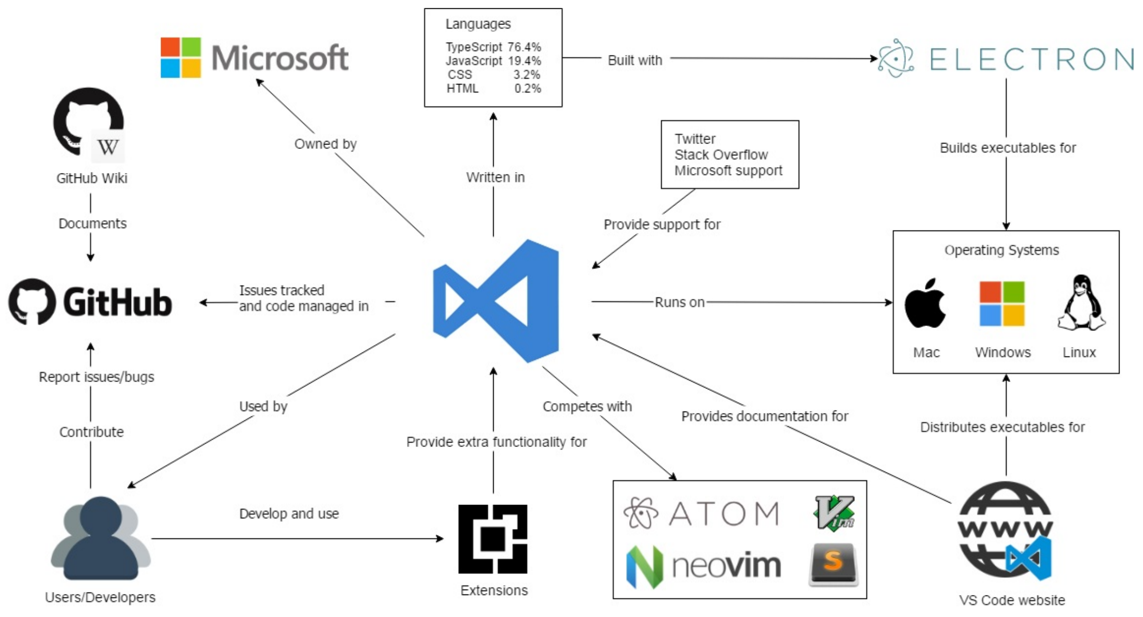


Figure 6: Context view diagram

The project is built using Electron, which takes TypeScript, JavaScript, CSS, and HTML and builds it into executables for the different operating systems. The website for Visual Studio Code then distributes these built executables for each of the operating systems. Next to distribution the website also gives documentation such as the API, docs, updates on new versions, and a blog.

Another important element in the context of Visual Studio Code are extensions. These extensions are generally developed by external developers to give extra functionality to Visual Studio Code and to personalize the experience for each user. An example is an extension that provides supplementary programming language support, like syntax highlighting and debugging.

GitHub is used to manage the code and track issues provided by users and developers, which can then be solved through contribution. The GitHub repository also includes a wiki which describes among other things how the project is structured, how you can contribute, and links to resources.

The main stakeholders included in this diagram are the acquirer Microsoft, the developers, and the competitors. The users and developers are treated as the same entity, because they perform similar roles in the context of Visual Studio Code. Some of the competitors that are identified are Atom, Sublime, Vim, and Neovim, but many other similar editors are available.

These are competitors because all of them strive to be lightweight text-editors and provide extra functionality besides just editing text. An IDE like IntelliJ, for example, is not considered a competitor, since it is more complex in use and not as lightweight.

Development view

This section describes the architecture that support the software development process, the way the code is structured and the standardization of design and testing.

Module organization

Visual Studio Code consists of a layered and modular core that can be extended using extensions as seen in Figure 7. Extensions are run in a separate process referred to as the extension host, which utilises the extension API.

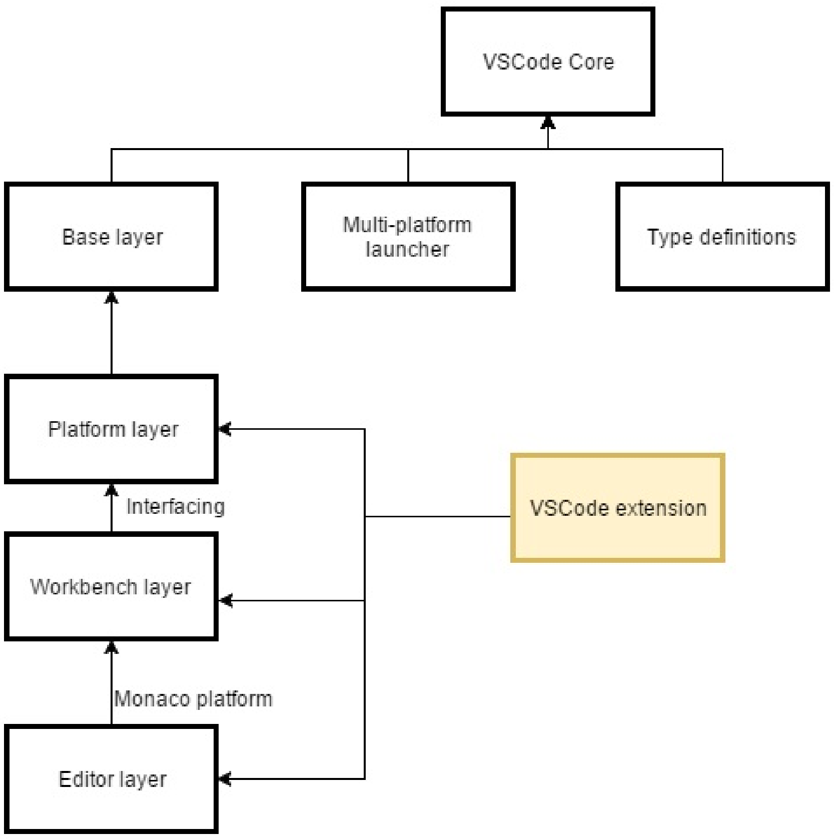


Figure 7: Module organization

Target environments

The core of Visual Studio Code is fully implemented in TypeScript. Inside each layer Visual Studio Code is organized by the target runtime environment. These layers will be explained in detail in the next section. This ensures that only the runtime specific APIs are used. In the Visual Studio Code project, the following target environments have been distinguished:

- *common*: Source code that only requires basic JavaScript APIs and run in all the other target environments.
- *browser*: Source code that requires the browser APIs like access to the DOM.
- *node*: Source code that requires Nodejs APIs
- *electron-browser*: Source code that requires the Electron renderer-process APIs
- *electron-main*: Source code that requires the Electron main-process APIs

The code is organized this way because of the API-like architecture the Visual Studio Code team applied. The layered way of splitting the project means that the base does not depend on anything else then the base. The API-like architecture means, that configurations/services/extensions can inject almost all kind of data that Visual Studio Code relies on. Outside developers can thus make modifications to Visual Studio Code without learning the whole system. Developing new extensions also becomes more attractive, since a service or lots of data can easily be injected. The editor layer of Visual Studio Code, is also a stand-alone editor, called Monaco Editor, and therefore a very modular part of the project.

Codeline organization

The core of Visual Studio Code is partitioned into the following layers:

- *base*: Provides general utilities and user interface building blocks
- *platform*: Defines service injection support and the base services for Visual Studio Code
- *editor*: The "Monaco" editor is available as a separate downloadable component
- *workbench*: Hosts the "Monaco" editor and provides the framework for "viewlets" like the Explorer, Status Bar, or Menu Bar, leveraging Electron to implement the Visual Studio Code desktop application. Next the layers are detailed and the functionality of each is explained.

The base layer

The code present is the base layer is used throughout Visual Studio Code. The common environment provides amongst others the structure to handle errors in Visual Studio Code, processes the events and declares the URI and UUID. In the common environment the code ranges widely from simple functions to reduce code duplication, such as returning a hash value for an object, to complex code which handles asynchronous processes. Also, the structure to handle errors is declared in the common environment, as well as event handling and declaration of URIs and UUIDs.

In the node environment functionalities are present that read the configuration files, handle the checksum for encryption, character encoding and decoding, directory and file manipulation and network handling. The imported modules are related to operating system-related utility methods, such as reading and writing of files, on the system on which Visual Studio Code runs on. Network modules are also imported in this module to interact with the web.

Besides the standard environments `browser`, `common` and `node`, the base layer also contains the `parts` and `worker` environments.

The parts environment consists of three parts. The first part is `ipc`, which stands for inter-process communication. As the name suggest, this handles the communication between processes in Visual Studio Code. Next is the `quickopen` part, this handles the functionality of the quick open file menu. Lastly is the part called `tree`, which is the base of the implementation of the DOM tree of Visual Studio Code. This includes the data the tree contains and the frequently used code to render the tree.

The final environment is the worker environment. A worker is a script that runs in the background, independently of other scripts without affecting the performance of the page. In this environment the implementation of a worker is present as well as a worker factory.

The platform layer

The platform layer defines service injection support and the base services for Visual Studio Code. The Visual Studio Code project is organized around services of which most are defined in the platform layer. Services is able to reach the clients via constructor injection.

A service definition consists of two parts: the interface of a service, and a service identifier. The latter is required because TypeScript doesn't use nominal but structural typing. A service identifier is a decoration and should have the same name as the service interface.

The platform layer builds upon the base layer, it creates instances and registers services for almost all things you see and do not see in Visual Studio Code. Commands through the developer pallette, keybindings, clipboard are all handled by the platform layer, but also the visible parts, like searchbar, markers are all handled here. The workbench layer in its turn builds upon the platform layer, which initializes much more details, like CSS, that are not handled in the platform.

Extensions are all instantiated and registered through the platform layer, which give the extensions a lot of power over the Visual Studio Code system.

The editor layer

The editor layer is the part of the system that handles the functionality and displaying of the Monaco Editor. It handles everything from syntax highlighting for different languages to user input like copying, pasting and selecting text.

Also services are defined in the editor layer, which can be used by the controllers to fetch certain data. One of these services is TextMate, which interprets grammar files for the use of text highlighting.

The final part of the editor layer is contributions. Contributions are separate components that extend the functionality of the editor. The functionalities are for example hiding and unhiding (block)comments, code indentation, and the usage of links. The contributions have styling elements included where applicable (e.g. link highlighting), and in some cases their own tests.

The workbench layer

The workbench layer hosts the Monaco Editor and provides the framework for viewlets. These viewlets are for example the explorer, status bar and menu nar and make use of the Electron framework.

The common environment has one component, namely the editor. The different editor components implement the `baseTextEditorModel` from the `editor` module. Furthermore there are components like `panel` , `options` and `viewlet` . All of these components provide interfaces for common environment of the workbench.

The electron-browser environment implements the actual GUI of the workbench, which is based on the Electron framework. An overview of this component is shown in Figure 8. The `main` component fires up the workbench. First the `shell` component is called. The `shell` component contains the different components from which the actual workbench is build. The shell makes use of five components.

- `crashReporter` : handles the workbench in case of a crash.
- `nodeCachedDataManager` : saves the settings of the workbench.
- `command` : handles the different keybindings which can be use in the workbench.
- `extensionHost` : handles the different extensions installed in the workbench.
- `actions` : handles all sorts of actions which can be done in the workbench, such as zooming in/out, switching from window and opening a new window.

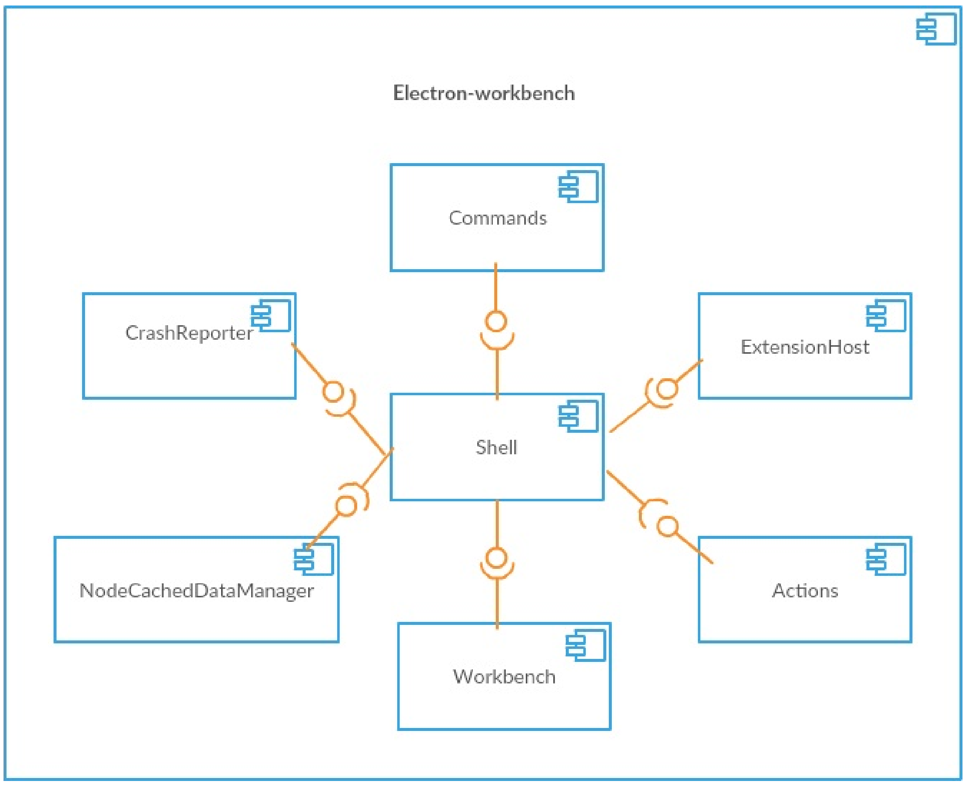


Figure 8: Workbench electron-browser component

The `parts` environment contains all components which together compose the Visual Studio Code workbench. There are at the moment of writing twenty eight different parts. Examples of parts are `git` , `search` and `output` .

A set of rules have been defined which each part must obey:

- There cannot be any dependency from outside vs/workbench/parts into vs/workbench/parts.
- Every part should expose its internal API from a single file (e.g. vs/workbench/parts/search/common/search.ts).
- A part is allowed to depend on the internal API of another part (e.g. the git part may depend on vs/workbench/parts/search/common/search.ts).
- A part should never reach into the internals of another part (internal is anything inside a part that is not in the single common API file).

Standardization

Visual Studio Code is a code editor used by over four million active users that welcomes developers to contribute. Without standardization of code and testing, contributions are not properly developed and the code base becomes a disaster. Contributors can submit bug reports, suggest new feature, build extensions, comment on new ideas, or submit pull requests.

Visual Studio Code has a wiki where developers can find information about the code base and instructions on how to build Visual Studio Code from the source. A detailed explanation on how you can contribute is also listed in the wiki. Coding guidelines are also listed on the wiki which define how code should be written in order to keep every file readable and maintainable.

Visual Studio Code uses tools, called linters, to enforce the coding guidelines. These tools have configuration files and are set-up in the git root of Visual Studio Code. By installing these linters as extensions in Visual Studio Code, developers get notified by visualized errors and other kind-of messages in the editor. The linters perform static analysis over written code, following the rules in the specified config files.

Furthermore, Visual Studio Code is tested with the use of the JavaScript testing framework Mocha. Also a smoke test is performed before each release. This Smoke test is carried out to make sure all major functionality works as intended.

Lastly, Visual Studio Code uses Travis CI and Appveyor for continuous integration on GitHub. Travis CI is used for testing if the build runs on Linux and Mac OS. AppVeyor runs the build test on Windows.

Technical debt

Technical debt, also known as design debt or code debt, is a concept in programming that reflects the extra development work that arises from wrong implementations. When code that is easy to implement in the short run is used instead of applying the best overall solution, adjustments need to be made in the long run. The overall best solution stems from the architecture used and followed for the system being developed, so identifying technical debt requires knowledge of the used architecture and implementation of existing code. Technical debt can also arise from different usage of multiple different syntaxes in one programming language. This causes readability of code to decrease and maintaining code gets more difficult.

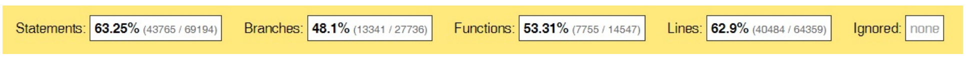
Identifying technical debt

As mentioned in the standardization section, Visual Studio Code make use of linters. Rules for naming conventions, type casting, and code styles are written here to ensure that contributors to Visual Studio Code do not create an increase in technical debt. Whenever the contributor does not install these linters, they get notified by the pre-commit checks which Visual Studio Code offers. If these checks fail, it means there is some sort of technical debt which must be fixed before being able to commit and push the desired contribution.

In the development view the layers of Visual Studio Code were identified. One reason for the layered structure was the low dependency in the layers. MaDGe is a tool for checking dependency between Javascript files and can be used to test the previously mentioned low dependency. The conclusion can be drawn after running the tool that no technical debt can be found by inspecting dependency here.

Identifying testing debt

At the moment of writing, 2832 unit tests are written for Visual Studio Code. The unit tests can be easily run through the command line, which takes about fourteen seconds to carry out. Visual Studio Code provides a way to generate a coverage report, as described on their wiki page. This coverage report generates an HTML website which include test coverage of every component. In Figure 9 the overall code coverage result is shown.



Statements: 63.25% (43765 / 69194) Branches: 48.1% (13341 / 27736) Functions: 53.31% (7755 / 14547) Lines: 62.9% (40484 / 64359) Ignored: none

Figure 9: The overall code coverage of Visual Studio Code.

The generated report divides test quality in three different categories: `bad`, `medium` and `good`. At the moment of writing, the overall code coverage of Visual Studio Code has a medium code coverage.

This testing debt can be improved by simply adding more tests. This could be done by just hiring extra developers whose primary task is to expand the amount of tests. Also various testing tools are available to automate this process.

A possibly better way, would be to let contributors test. Letting contributors improve the testing debt which can be done as follow. Test cases could be added as issues on GitHub for contributors to implement, since this is not done currently. Another possibility is to demand unit test for each contribution. This way code which is added by contributors outside of the Visual Studio Code core development team is automatically tested.

Discussion about technical

The discussion of technical debt is mostly present on GitHub. A label 'debt' exists to mark related issues and in the monthly iteration plans developers can be assigned to work on an issue related to technical debt. On GitHub the monthly iteration plan is presented, of which some tasks are categorized as technical debt.

Also TODO mentions are present in the source code, of which most are assigned to a specific developers. This ensures that the problem is not left as is.

Functional view

The Functional view of a system defines the architectural elements that deliver the system’s functionality [1]. The view shows the key functional elements, the external interfaces, and the internal structure of the system.

Functional capabilities

Functional capabilities define what the system is required to do and what it is not required to do [1]. Since Visual Studio Code strives to be a lightweight code editor, the main functionalities that it needs to have coincide with that. Table 1 shows the core functionalities required of Visual Studio Code and describes what their responsibilities are.

Functionality	Description
Editor	The main component of the system is the editor itself, which Visual Studio Code calls the Monaco editor. Its main responsibilities are showing the text from files, syntax highlighting, and the editing of the text in the file.
Search	The search functionality enables the user to search for occurrences in files. It also gives the functionality to replace these occurrences.
Explorer	The file explorer displays the folder structure of an opened folder and shows the contained files. Next to this it also has the functionality to change between editors and show the open editors.
Debugging	The debugging component allows for debugging applications written in Visual Studio Code. Using breakpoints and a debugging environment, the code of the program being debugged can be analyzed.
Marketplace	The marketplace provides a way to install new extensions and provide information on these extensions.
Use extensions	The system enables the use of external extensions to change how certain parts of Visual Studio Code work. It makes changes by being a connection between the extensions and the program.
Git	The git component allows the user to make use of git version control in Visual Studio Code itself.

Table 1: Functional capabilities

For a text editor it is clear what it is not required to do, for example IDE's can compile the written code and a text editor should not. Since Visual Studio Code is more than just a text editor there are functionalities that are not part of its responsibilities. As an example, it should not directly implement the functionalities of external extensions, since this responsibility lies with the extension itself. From the interview it is know that Visual Studio

Code tries to be inbetween text editors and IDE's, such that it takes useful functionalities from the IDE but still keeping the lightweight aspect of editors. This means that it should not require functionalities to make it a full-blown IDE.

External interfaces

The external interfaces provided by Visual Studio Code mainly concern functionality to make extension development possible. Among other things the functionality concerns making changes to the editor, such that syntax highlighting for different languages can be achieved. Next to that it also allows specific debuggers for languages to be made. There are too many interfaces to completely list them in this report, so for a full list of available external interfaces refer to the API. As an example of what is provided, table 2 shows a few of the accessable interfaces.

Namespace	Description
window	The window namespace deals with the current window of the editor, it shows messages to user and keeps track of open editors.
extensions	Provides the ability to get extension by their id and can then activate them, this way extensions can make use of other extensions.
CommentRule	Holds the properties of how line and block comments for a language work.
TextDocument	Represents a text document, such as a source file. It holds properties for the filename, the uri of the file, and the number of lines. It also provides functionality for getting the text from the document and saving the file.
TextEditor	Represents the text editor that is attached to a document, it holds the selected text and it can perform edits and decorate text.

Table 2: Examples of external interfaces

Internal structure

In the section on the development view the different environments of Visual Studio Code are described, namely the base, code, editor, platform, and workbench environments. Each of the different functionalities belong to different environments, which are as follows:

- The search, explorer, debugging, marketplace, and git functionalities are a part of the workspace environment.
- The use of extensions is a part of the platform environment.
- The entire editor funcitonality is the editor environment.

Performance and scalability

Performance and scalability are important factors in big software projects. These factors should be kept in mind from the start, while performance regression in an early level might not be noticeable, but when scaling to many users, or to large files, in the case of Visual Studio Code, the system should still be working properly.

Desired quality

The desired quality of Visual Studio Code is to be a lightweight code editor, which supports developers in their core edit-build-debug cycle. To satisfy this, Visual Studio Code should perform well in both real and perceived performance. To effectuate this, Visual Studio Code tries to let the user experience be lightweight as well. And to keep up the performance of Visual Studio Code, feature owners need to agree with any architectural impact a change may make.

Applicability

Since Visual Studio Code is lightweight the requirements to run Visual Studio Code are that it should run on recent hardware. It is recommended to use a processor of 1.6GHz or faster and at least 1 GB of ram. Though the performance of Visual Studio Code itself is tested and known, there are elements from which the performance is unknown. Namely the extensions which can be installed from the marketplace, since these extensions are developed by third party developers.

Concerns

One of the biggest concerns for Visual Studio Code is the response time. It's important that users do not have to wait considerably long for opening files, since this reduces the time they are able to develop software. This response time can be combined with the peak load behavior when large files are opened. This causes peak load behavior, since the whole file needs to be loaded.
To reduce these concerns Visual Studio Code tries to tackle the predictability concern, by providing stable releases every month.

Activities

To visualize performance of Visual Studio Code, practical testing is needed. It is difficult to determine good practical tests, while Visual Studio Code can be used in many ways. This is why practical tests gives the developer insight of performance on specific scenarios and not project wide performance.

The next practical tests are performed for performance measurement, all on the same hardware run three times:

- Start-up time of VS Code
- File-open time while VS Code is already open
- RAM usage

For the practical testing a small script was written to generate a text file with one million lines.

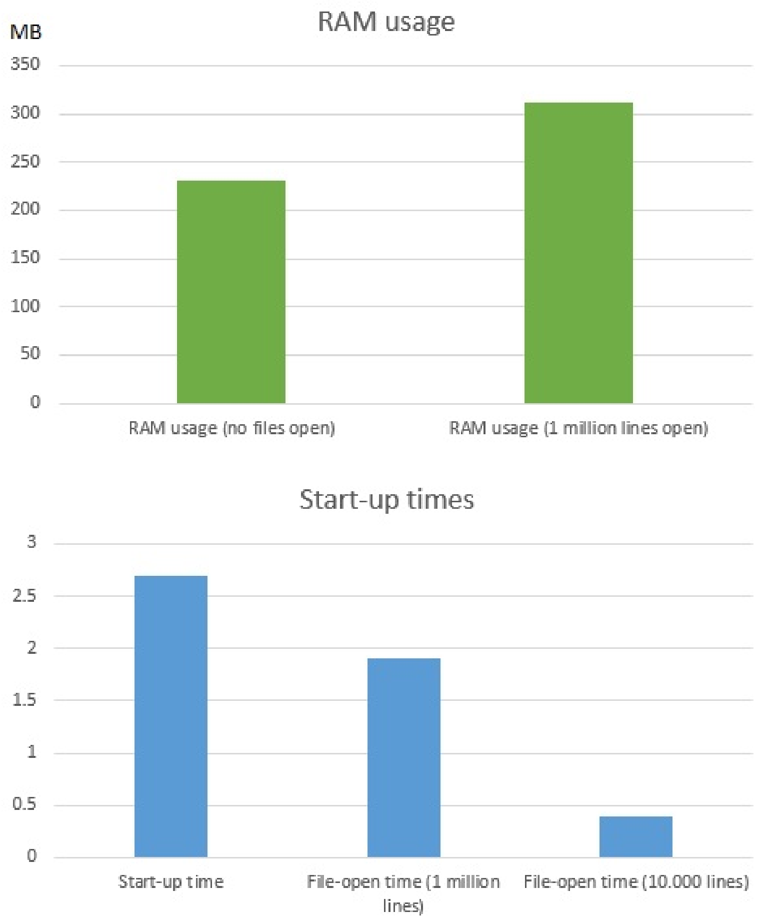


Figure 10: RAM usage in MB and start-up times in seconds

Figure 10 depicts that the usage of RAM does not increase rapidly when opening big files. A file with one million lines does not give that much of an impact on the memory of Visual Studio Code. For opening the file however there is a significant increase in loading time, from 0,39 seconds to 1.90. Because developers open and edit many files in a project, this can be an important metric for the Visual Studio Code development team. Since improving these loading times ensures that the system does not slow down its users. Visual Studio Code does not allow opening a file with more than ten million lines, displaying the message

`The file will not be displayed in the editor because it is very large`. Opening the one million lines file did not seem to have such an impact on the resources, so this is probably precaution against automatic opening of project files.

Tactics

Another way Visual Studio Code tries to optimize processing is by spending a entire week after a release to try and optimize the implementation. It may occur that some parts of the code have been rushed in order to ship it with the release. Processing is not really prioritized, but since Visual Studio Code depends on certain frameworks such as IntelliSense keeps working until files are too large. To minimize the use of shared resources, modules are divided in different layers such as base and common. Finally Visual Studio Code uses asynchronous processing in the form of a worker. As discussed in the development view workers can be used to run desired processes in the background which do not affect performance of the current page of Visual Studio Code.

Conclusion

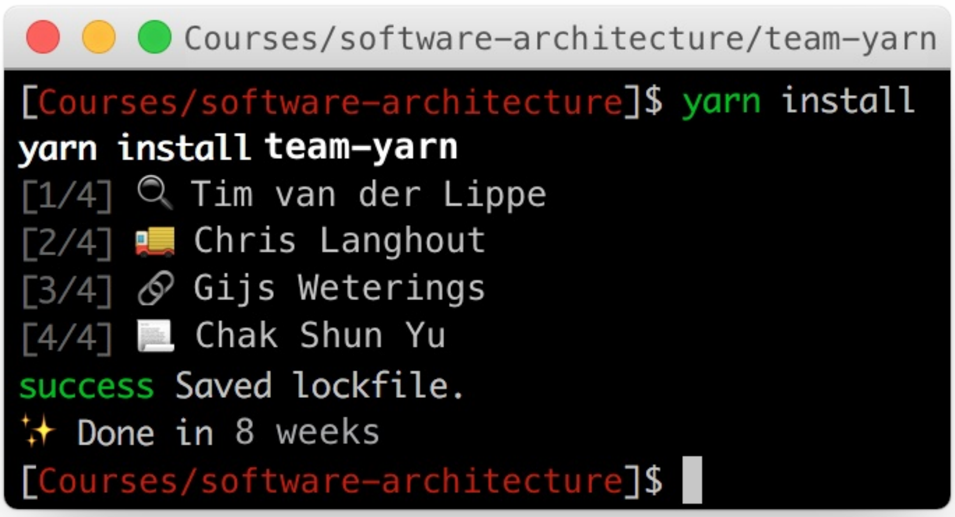
The goal of this chapter was researching the open-source project Visual Studio Code and presenting a detailed description on the software architecture, aswell as the factors it depends on to be a success. These factors are determined in the development view, performance perspective, technical debt and by interviewing Erich Gamma. In the chapter is concluded that the architecture is divided in different layers and that continuous integration tools and linters are used for maintaining code quality. Visual Studio Code development is driven by the community, prioritization of features comes from issue tracking and every week a feature is implemented and optimized by the internal development team. The chapter indicates the importance of a structured workflow that can ever be improved so that code quality can be maintained, even with many developers contributing. There is still room for improvement in technical debt, like TODO's which are forgotten and test coverage what is lacking behind. Most of the development environment however is taken good care of what is shown in the chapter. Visual Studio Code now has over four million users and delivers a new version every month.

References

1. Nick Rozanski and Eoin Woods. 2011. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.

Yarn

By Tim van der Lippe, Chris Langhout, Gijs Weterings and Chak Shun Yu.



```
Courses/software-architecture/team-yarn
[Courses/software-architecture]$ yarn install
yarn install team-yarn
[1/4] 🔍 Tim van der Lippe
[2/4] 🚚 Chris Langhout
[3/4] 🔗 Gijs Weterings
[4/4] 📃 Chak Shun Yu
success Saved lockfile.
✨ Done in 8 weeks
[Courses/software-architecture]$ 
```

Delft University of Technology

Abstract

Yarn, an alternative client for the existing npm registry, went open-source at October 11th 2016. Yarn started as an internal project at Facebook. Now Google, Exponent and Tilde are collaborating. For this chapter, the architecture of Yarn was analyzed in detail. Furthermore, technical debt has been identified and solutions were proposed. Some of these solutions were implemented and provided to the development team as contributions. The Yarn project looks very promising, and is already used by a lot of people. More and more projects are choosing Yarn over npm, and this will only increase as bugs are resolved and additional features roll out.

Introduction

Over the past years, the growth of JavaScript (an implementation of ECMAScript) has exploded. It is used in almost every segment of programming, from front-end to the world of embedded devices.

To be able to manage all components, libraries and frameworks written for all these applications, Isaac Z. Schlueter started work on the npm package manager and npm registry. Like JavaScript itself, the growth of the registry seems to have no ceiling. In Figure 1, the growth of the npm registry is displayed.

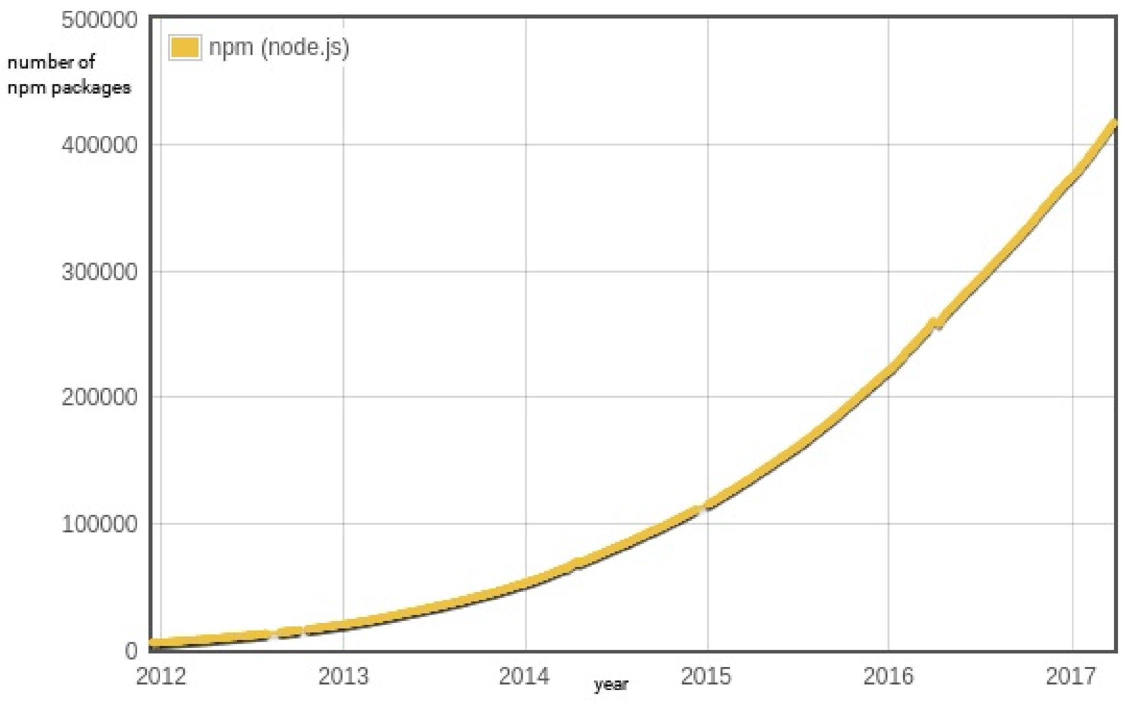


Figure 1. Diagram showing the growth of the npm registry. Source: modulecounts.com

But, the explosive growth of this ecosystem also brings on challenges. Facebook in particular had issues dealing with the growth of npm modules in their React.js project [2]. This was due to the npm client's way of handling sub-dependencies, and the non-determinism of their install algorithm.

To solve these issues, Sebastian McKenzie(@kittens), a Facebook employee, started work on Yarn on Jan 23rd 2016. Soon after, Exponent, Google, and Tilde joined to collaborate on the project, discussing features, strategies and architecture. Ten months later, the first official release of Yarn was public.

While the big issues with the npm client were resolved by Yarn, the speed of development and the ever-changing ecosystem for which Yarn is built resulted in new challenges. Managing technical debt, dealing with open source development and keeping up with the demands of users are the biggest challenges for Yarn, in order to stay successful. In this chapter, that process is analyzed, and meaningful contributions are provided and described.

Stakeholder Analysis

As described before, one of the main reasons that Yarn was created, and thus exists, is Facebook and their product React.js. They are two of the most important parties involved with Yarn, also known as stakeholders. A stakeholder analysis was conducted to identify all the stakeholders of Yarn, according to the categories described in Rozanski and Woods (2012) [1]. This section is dedicated to elaborating on this analysis. The most significant and important stakeholders will be provided, together with a description of their role in the Yarn project.

Acquirers

The incentive of creating Yarn started from Facebook, as they ran into problems with consistency, security, and performance with npm [2]. Yarn was launched by Facebook [2] and most members of the core development team are employees of Facebook.

Assessors

The complete Yarn project team is in a sense an assessor of Yarn, as all developers oversee the project and try to make sure that it conforms to standards and quality. This is reinforced by their Code of Conduct, which states: "[They are responsible] for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior" [3]. Three individuals stood out from this group: @bestander, @daniel15 and @wycats.

- **@bestander** is an employee of Facebook. He, at the time of writing, is the second highest contributor of the repository. But more importantly, he takes an active role in the GitHub repository by overseeing and starting discussions and reviewing almost all the (recent) issues and PRs. He is also very active on the RFC GitHub, a place dedicated for the Yarn project team to discuss about important aspects of Yarn. The RFC process is elaborated upon in Release Management.
- **@daniel15** is also part of Facebook. He is, like @bestander, also highly active on the GitHub repository. He does a lot of explaining in issues and pull requests, helping external contributors with their improvements.
- **@wycats** is not an employee of Facebook. He was approached due to his experience with application-level package managers (Bundler for Ruby and Cargo for Rust) and now part of the Yarn project team [4]. Although his contributions in the form of lines of code are relatively low, his main importance comes from elsewhere. He is actively involved on the RFC repository for Yarn, and plays an important role in the discussion and decisions on features and the overall design of Yarn.

Communicators

Communicators are described as parties which "provide training for support staff, developers, maintainers, and so on, and create manuals for the users and administrators of the product or system" [1]. To identify communicators, the history of various documents in the project have been looked into. Examples are the `CONTRIBUTING.md`, a guide on how to correctly contribute to the Yarn project, the `README.md`, and the `CODE_OF_CONDUCT.md`. In the history of these files, we have searched for people who have made significant contributions to these files with the purpose of guiding or helping other users. Using this method, @kittens, @cpojer and @thejameskyle were identified as communicators. Not surprisingly, they are all members of the Yarn project team.

Next to these people, there are also all the contributors to the GitHub repository of the website of Yarn. Their work is of significant value to other stakeholders, as the website contains startup guides, installation guides, manuals and more information about Yarn.

Competitors

Before Yarn, two package managers for web development ecosystem existed. At first Bower was seen as the early alternative to npm, then Yarn was created as an improvement over npm. Following the announcement of Yarn [2], Bower suggested its users to use Yarn instead of Bower despite the dropped support of Bower in Yarn [5]. That left npm and Yarn as competitors of each other on the field. Specifically, the npm client is the competitor of Yarn, as npm is also a supplier of Yarn due to its connection with the npm registry.

Developers

For the developer stakeholders, the decision was made to split the category and define additional sub-types of developers. The sub-types and their respective criteria are defined as follows: *Core developers*, who have write access to the repository and contribute frequently, *external contributors*, who do not have write access to the repository, and offer pull requests from forks of the repository.

- The *core developers* of Yarn are @kittens, @bestander, @daniel15, and @cpojer. These are the parties that started development before Yarn became open-source. Not surprisingly, they are all part of the Yarn project team. @kittens is by a significant difference in commits the top contributor to the Yarn project, as well as the creator of Yarn.
- Other developers that contributed significantly to the repository of Yarn, but are not part of the Yarn project team, are *external contributors*. Currently, 232 contributors are listed on GitHub. Together, they form the developers of the *Yarn community*.

Suppliers

Suppliers are parties which "build and/or supply the hardware, software, or infrastructure on which the system will run" [1]. For Yarn, only npm was identified as a supplier. This relation results from the reliance of Yarn on the npm registry [2]. Although Yarn is created to replace the existing workflow of the npm client, thus being a competitor, it emphasizes the compatibility with the npm registry [2]. As this registry is built, maintained and provided by npm, npm is classified as a supplier.

Users

Besides Facebook, important users of Yarn are Exponent, Google and Tilde. They were heavily involved in the design and creation of Yarn [2]. The Polymer team is another important user. Polymer was approached by Facebook, asking what requirements they had for using Yarn [6]. Other than Polymer, the Yarn team reached out to Ember, Angular and React.js to make sure that Yarn would be a good fit for projects that are using these frameworks [4]. Last, but not least, there is the Yarn community. Part of them, the developers, were already mentioned. Not all of the community are developers, but every person in the community is a user of Yarn. They define what they expect of Yarn, mainly through GitHub issues, and ultimately form and make use of Yarn.

Context View

The context view (Figure 2) shows the interaction between Yarn and its environment. The most important stakeholders are represented in the view. The user base of Yarn is slowly growing as people and projects are trying it out as a substitute for npm. Yarn lists Bundler, Cargo and npm as inspiration in the readme of their GitHub page [7]. Bundler is a manager for Ruby application's gem dependencies. Cargo is the package manager for Rust and npm is the preceding JavaScript package manager.

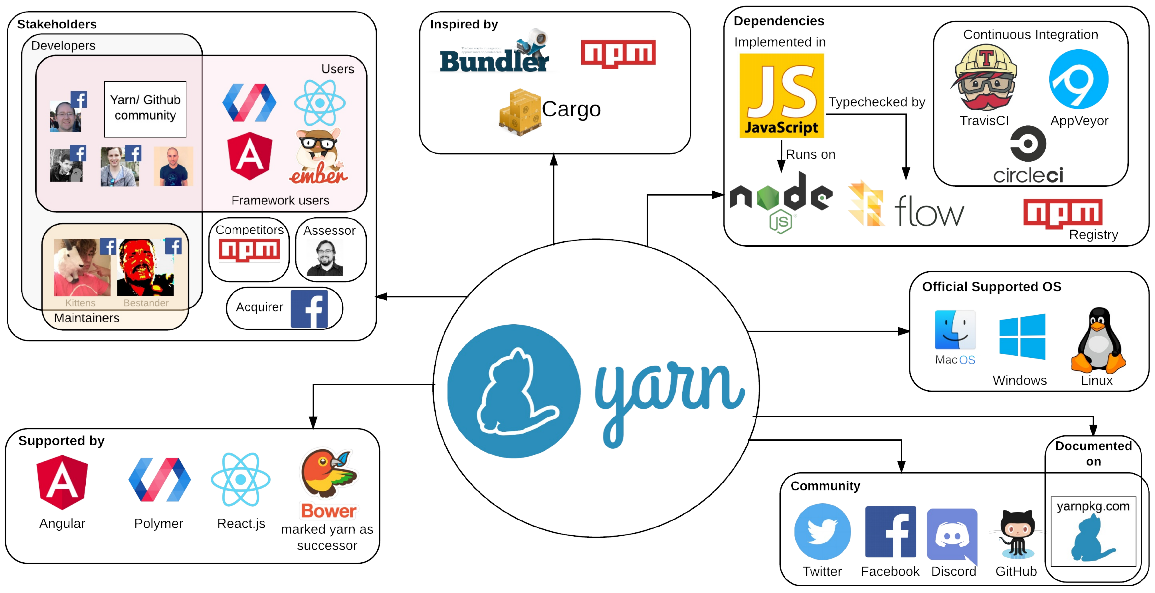


Figure 2. Diagram displaying the context of Yarn.

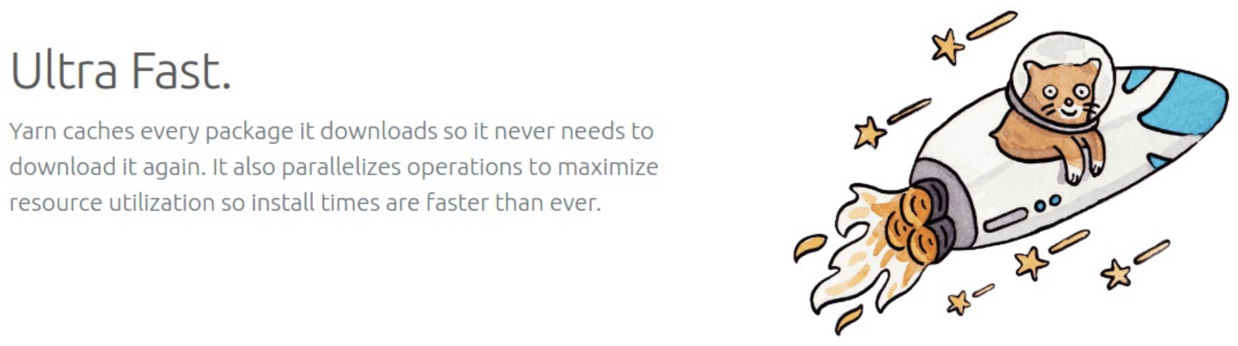
Yarn is implemented in JavaScript that runs on Node.js. The packages a user can install are retrieved from the npm registry. Flow.js is used to typecheck the code.

For continuous integration (CI), Travis CI, CircleCI, AppVeyor and Jenkins are used. These three services are not only used for continuous integration, they are also functioning as separate integration tests to check compatibility. The community of Yarn exists on Twitter, Facebook, Discord, GitHub and on their own site (where the documentation also exists).

Official support of the Yarn project is given by numerous projects, these include Angular, Polymer, React.js and Bower. Bower even went as far as mentioning Yarn as their successor [5].

Performance Perspective

Performance is one of the key features of Yarn, prominently displayed on the homepage (see Figure 3). In this section, the architectural decisions made to ensure high performance are listed.



Ultra Fast.

Yarn caches every package it downloads so it never needs to download it again. It also parallelizes operations to maximize resource utilization so install times are faster than ever.

Figure 3. Performance advertised as one of the key features of Yarn.

One of the primary usecases for Yarn is the performance feature compared to the npm. In the initial blogpost of Yarn [2], the developers point out that by parallelizing operations the installation of dependencies is significantly faster via Yarn compared to npm. The most known usecase is React Native. Concrete benchmarks are published on the Yarn website for which different configurations of Yarn are compared against the npm client. These benchmarks show that the installation process of Yarn is an order of magnitude faster.

Architectural Support for Performance

The parallelization of Yarn's installation process occurs in three different phases [2]:

- Resolution of dependencies
- Fetching dependencies
- Linking(/Copying) dependencies

These three different phases correspond to the package structure, as described in the architecture. For each dependency, the first step is to resolve the package in the registry. This step will include discovery of its sub-dependencies. First, all resolutions are done in parallel. Then, if there are any sub-dependencies, they are asynchronously added to the set of dependencies. Finally, once all dependencies are resolved, the resolution phase is finished.

Every resolved dependency is saved and written to a `yarn.lock` lockfile. The lockfile of Yarn locks down a resolved version, which means that consecutive installations do not require resolution. Memory and storage is therefore sacrificed in favor of preventing network requests to resolve dependencies.

Once all dependencies are resolved, or read from the lockfile, the content of each dependency is fetched. Yarn maintains a global cache on the users machine, in which one installation per version exists for every package. In other words, there will be one instance of `A@1.0` and once instance of `A@2.0` in the global cache. When a dependency is fetched, the fetcher first checks this global cache. If a package already exists there, then no network request is triggered. Else, a network request is spawned and the content of the dependency is inserted in the global cache. Finally, the dependency is copied into the local `node_modules/` . This architectural approach ensures that re-installation of common packages does not unnecessarily delay the install process.

Every step for each dependency is fully asynchronous in the Yarn architecture, which allows full parallelization of every installation step.

Architecture

The architecture of a system is dependent on the processes and workflows of the development team, as well as the project itself. To illustrate this, this section first explains the high-level package structure, then identifies key design patterns. Next, an analysis of the testing practices of the development team is discussed. Finally, the Yarn release process is analyzed.

Package Structure

Yarn has several distinct functional elements performing various functions. The aim of this section is to identify these elements, and describe their responsibilities and interaction. For this, the top-level packages from the folder with the main functionality, `src/` , are used. These packages are the most representative for the elements in the project, as they group the underlying modules by functionality. In the end, Figure 4 shows a diagram displaying the overall structure.

Almost every package exports all functionality in an `index.js` , to ease the importing of classes in this package. Moreover, it allows developers to restructure a package later, without breaking the usages of the APIs exposed in the package if they would directly reference the source files.

Starting The Process

The main interaction between Yarn and its users is via the cli commands. Whenever a user issues a command, it will trigger a corresponding process. The details of this process are described in the corresponding command file in `Commands` . The whole process is then orchestrated using the `Package Resources` . One of the major processes is the installation process, which will be covered step for step in this section.

Resolving Dependencies

The first step in this process is to resolve all the dependencies of the user. Resolvers are responsible for resolving the location of the packages specified by the user. Based on the incoming package name and version, the resolver determines where the actual source code/distribution of the dependency is stored. Several implementations are available in the `resolvers` package, which can resolve to external sources such as GitHub.

After obtaining the location of a package, the resolver puts all the information inside a registry. `Registries` are responsible for retrieving information of packages from the global npm registry. Several registry strategies are implemented in this package.

Fetching Dependencies

After resolving all the dependencies, all the registries containing their information are received by `Fetchers`. Fetchers are used to retrieve the content of external dependencies from their respective sources. Multiple extensions of the base class are implemented for the specific sources.

Linking Dependencies

The last step of this process is to link everything together, after fetching all the dependencies from their respective sources. This is done in in the project folder of the user. Essential in this process is the `Lockfile`. The lockfile, as its name suggests, is responsible for locking down the package versions of all dependencies from a project. The functionality of parsing and writing the lockfile is implemented in this element. Details of this process are provided in the performance perspective.

Common Resources

The above process is described with the main workflow as a common thread. Due to this, some elements are not touched upon, as they are mostly used for tasks outside of the main scope. Here, these elements are listed and described.

- **Reporters.** Yarn can report its status via various reporters. The primary implementation of `BaseReporter` is the `ConsoleReporter`, which has its own subfolder `reporters/console/`. This package also handles localization in the `reporters/lang/` package.
- **Util.** The biggest package thus far is `util`, which contains numerous files with specific functionalities which would not fit in a different package. This package also contains classes that implement functionality that is used by multiple other packages. There does not seem to be a cohesive organisation in this package with one other than `index.js` exposing the other classes.

- **Top level resources.** The top level package contains several resources, including config and types for the application. Moreover, it contains the errors used to report to the user and logic to handle package content.

Figure 4 displays the relations between the elements that are described in the whole process above. An arrow from package A to B indicates that A depends on B.

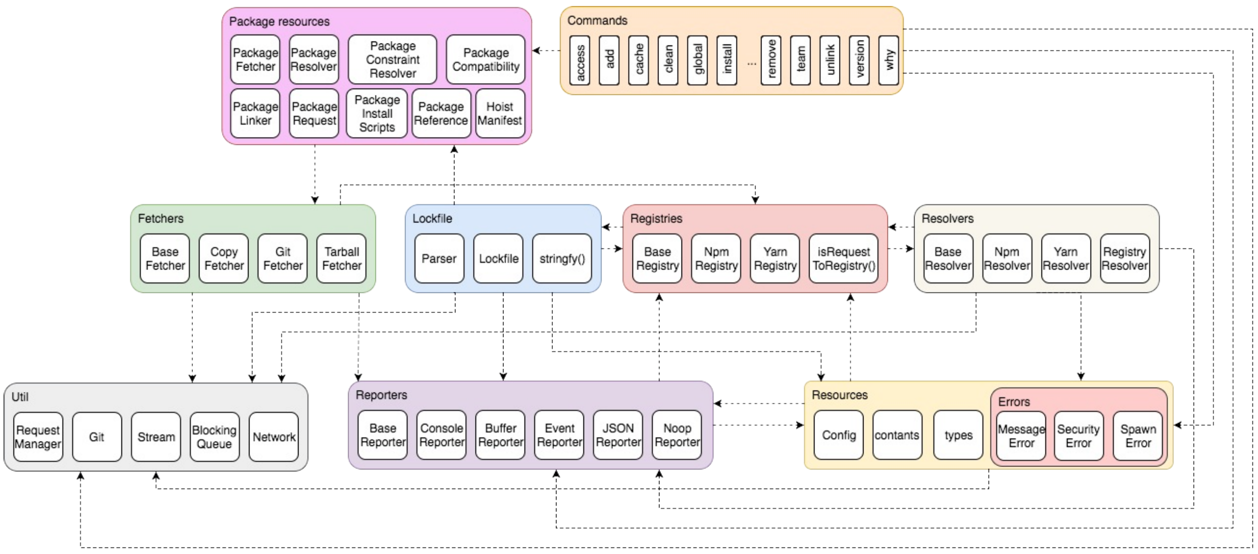


Figure 4. Diagram displaying the package structure of Yarn.

Identifying Key Design Patterns

Keeping a piece of software maintainable, stems in having a great baseline architecture. Part of this architecture is the proper application of design patterns.

Strategy Pattern in Fetchers and Resolvers

Since packages that can be installed by Yarn can come from different sources, it is important to consider these different paths. For example, resolving a request for a package can go through the npm registry most of the time, but for some packages this does not hold. An example is internal packages hosted on the organization's GitHub.

To summarize from Yarn's announcement blog [2], the installation process of a package is:

1. Recursively resolve all dependencies of a package.
2. Fetch the package by first checking if it is already downloaded, and if not, download it from the source.
3. Link everything together by copying the files from the global cache into the `node_modules` directory.

These steps ensure a deterministic install.

The first two steps make sure the packages can come from a wide range of sources. To achieve this, Yarn employs a strategy pattern. Yarn analyzes the `lockfile` and `package.json`, and retrieves all names and/or git repositories from all packages listed. By resolving these dependencies through the corresponding resolvers, it is possible to support many sources in a stable and governed way.

If a package is not already cached locally, it needs to be fetched. Again, depending on where the package is hosted, Yarn employs a different strategy to actually fetch the package. Some sources provide compressed archives, others are git repositories that need to be cloned, while even others may simply exist on the filesystem. These different strategies are all very neatly implemented, extending from a base class to deduplicate common logic. A caveat we encountered was that some functions were defined in the base class, to instantly reject the `Promise` returned. This is done because JavaScript does not have abstract classes, allowing the forced implementation by a subclass. With the current size of these classes, it is still manageable to catch these methods by hand. It should be noted that is not an ideal solution. This is a direct limitation of both the language and the typechecker used by Yarn.

Adapter Pattern for the JSON Reporter

The reporters can distribute the output for Yarn commands to various destinations in various formats. Among other use cases, this is primarily used to be able to report to users in the console, as well as save reports during CI processes. The JSON reporter [8], for example, writes all reports to a JSON structure, which is saved to a file. Since most of the messages can be handled in the same way, most methods in this reporter do the exact same thing: call the `_dump` method with the type of message and the details. That method then writes the object to the file. Because its job is to translate from one system to the other, this is an adapter pattern that transforms the workflow of the reporters to that of a file writer.

Testing Strategy

Testing is an integral part of the review process for Yarn. For every bug fix and feature addition, tests as well as a test plan are required. There are multiple practices for testing code. First of all, every bug fix requires a unit test that reproduces the original issue and (once the fix has been applied) verifies the issue is resolved. The unit test also serves as a regression test, to make sure that the original bug is not reintroduced after other changes are made to the system. Secondly, feature additions are tested with integration tests. Examples of such integration tests can be found in the test suite for CLI commands in `__tests__/commands/`.

Tests are run via `yarn test`, essentially bootstrapping Yarn's functionality to invoke scripts to also execute the tests. The actual tests are run by Jest, a testing framework also developed by Facebook. Jest can run tests concurrently, to speed up the build and also test potential concurrency issues for Yarn. When running the tests, several coverage metrics are also measured. The overall coverage can be inspected after all tests are executed. @kittens indicated that obtaining near 100% coverage is a goal of the development team. At the time of writing, the coverage is around 60%, which could be improved.

Ensuring Continuous Code Quality

One notable thing on the Yarn GitHub page is the usage of four different CI services, namely Travis CI, CircleCI, AppVeyor and Jenkins. Normally, one service is enough to check whether all tests succeed. In the case of Yarn, however, every CI service has additional functionality as a platform test. If the build fails on one of the services, this could indicate a platform specific bug. In practice, it often occurs that one of the CI services fails. This is caused by timeout errors that we also experienced locally. The failing test suite probably depends on an active internet connection and a slight delay in this has a probability to fail the test. Lastly, Jenkins also does nightly end-to-end tests in a Ubuntu Docker container [11]

Release Management

In #376, @kittens asks @bestander and @wycats suggestions for a regular release process. @wycats suggests a six week release cycle, as used in Rust and Ember (acquired from Chrome and Firefox), which means that every six weeks a new major version is released. Bugfixes can still be released as fast as possible, but under strict CI. At the point of writing, the focus of Yarn lies mainly at fixing bugs and resolving issues, and less on adding new functionalities. Despite this, new substantial features are still welcome. For this, developers are directed to follow the Request For Comments (RFC) process happening in the yarnpkg/RFCs repository.

The intention of the RFC process is the control of new features that people suggest to add to the project. This process is meant for people that want to suggest "substantial" changes to Yarn or its documentation. A few examples of these changes are:

- A feature with a new API surface area that requires a feature flag (described in feature flag section below).
- The removal of one or more features that are already shipped as part of a release.
- Introduction of new idiomatic usage or conventions, even when it includes no code changes to Yarn itself.

> The RFC process is a great opportunity to get more eyeballs on your proposal before it becomes a part of a released version of Yarn. Quite often, even proposals that seem "obvious" can be significantly improved once a wider group of interested people have a chance to weigh in. Source: Readme RFCs repository

All development to the Yarn project happens via pull requests, forcing the run through CI. The master branch must always have a succeeding CI run.

After the first major release, the focus is kept on closing issues as fast as possible. When that slows down, the six-week major release cycle will be adopted. This slow down has not yet happened at the time of writing.

The fast release process is largely automated by @bestander. The master branch is taken and a new version-stable branch is created along with a tag. A Jenkins build is triggered and builds a .tar file, that is deployed to GitHub releases. Then, @bestander tests it on internal Facebook repositories. Any bugs found will be reported in issues. When fixes for these bugs are merged into master, they are added to the version branch. After that, the changes are added to npm, and the CI builds and deploys the new version.

Feature Flags

As mentioned before, before adding a new feature, the RFC process is followed. When a new feature is being developed it is wrapped in a feature flag. This means that this feature is not used by default in the Yarn application, but can be enabled by enabling the corresponding flag. When a feature is stable enough, the decision can be made to enable this feature by default and therefore disabling the feature flag ability for this feature.

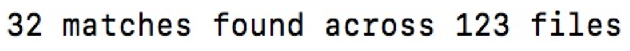
Technical Debt

Technical debt is "a metaphor for tasks that were left undone, but that run a risk of causing future problem if not completed" [9]. Identifying technical debt is therefore an interesting task to measure the maintainability and stability of Yarn.

To analyze the technical debt of Yarn, several inspections tools were run on the source code.

Code Smells

JSInspect provides a report in the console of the matches it found. A match consists of the detected common source code and the files between which this code is shared. Additionally, the diff between these files is also provided, highlighting the differences and similarities.



```
32 matches found across 123 files
```

Figure 5. Result of a JSInspect run on the Yarn `src/` folder.

At the time of writing, an execution of `jsinspect` on the `src/` folder of the Yarn project resulted in 32 matches found across 123 files (see Figure 5). Not all of them, however, are actual code smells, as JSInspect looks for any similarity between source code and matches onto that. The correctly identified code smells are mostly duplication, such as three instances of the same error reporting function in one file. The risk involved with this is the Shotgun Surgery code smell, where modifications require changes at multiple locations.

Code Maintainability

Plato, a JavaScript tool with the purpose of visualizing source complexity of JavaScript files and projects, was also used to analyze the project. Running Plato creates a detailed report focusing on Maintainability, Source Lines of Code (SLOC), Proneness to Errors and Complexity of the code. Since the computation of the Maintainability index mainly relies on lines of code, only the true lines of code are touched upon [10].

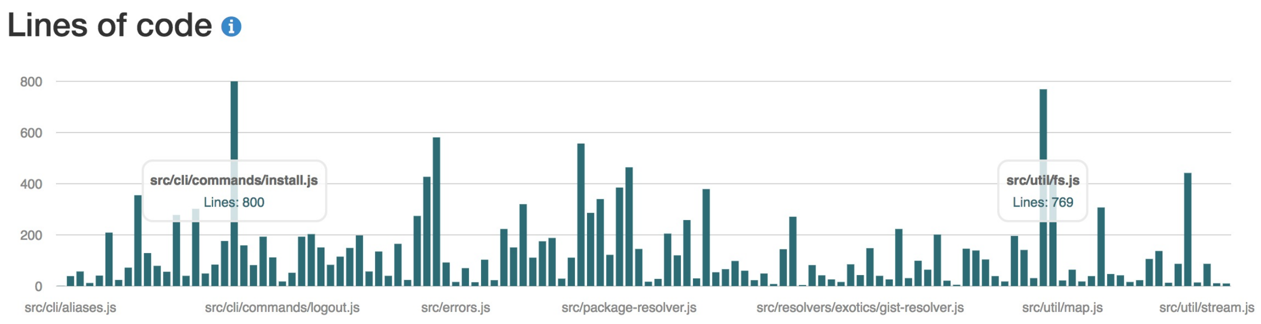
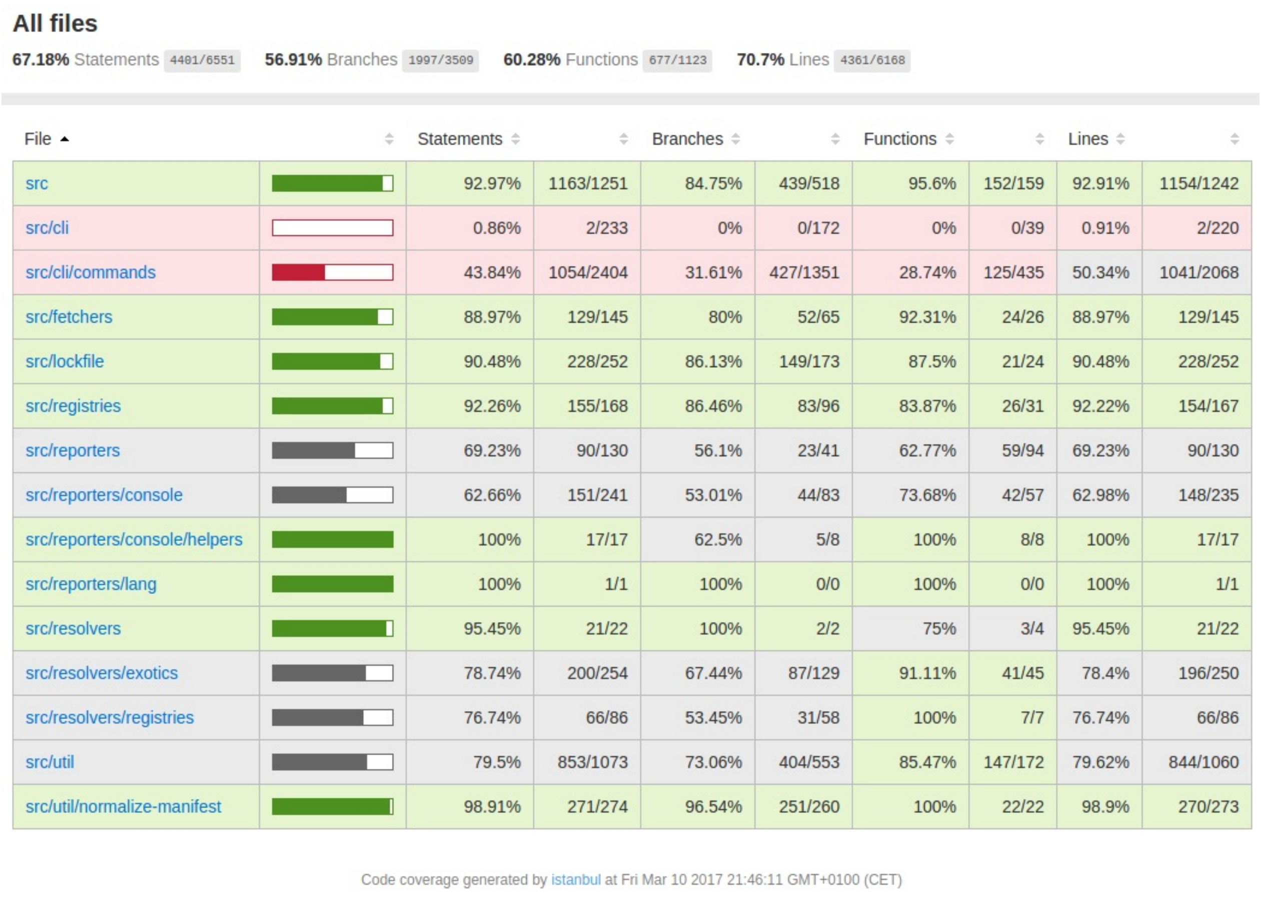


Figure 7. Source lines of code of each individual source file in Yarn.

The average source file of Yarn contains 136 SLOC. A more informative metric would be to see the relation of this metric between files. Therefore, the SLOC for every individual source file in the Yarn project is shown in Figure 7. A large amount of variation in SLOC can be observed between the source files of the project. Two files stand out significantly from the rest in terms of SLOC, namely `src/cli/commands/install.js` and `src/util/fs.js` with 800 and 769 lines respectively. Both of these files are susceptible of large functions and classes, and containing too many responsibilities. Consequently, these files are at risk of becoming a God Class.

Current Code Coverage

Testing is an integral part of the workflow of the Yarn core team. Issue #510 shows that the team is actively aware of its code coverage and aims to increase the overall coverage. The code coverage of Yarn is integrated in the test runner Jest per the `--coverage` CLI option. A `coverage/` folder is created which shows a webpage (see Figure 8) with the coverage per folder, file and even per line.



All files

67.18% Statements 4401/6551 **56.91%** Branches 1997/3509 **60.28%** Functions 677/1123 **70.7%** Lines 4361/6168

File ▲	Statements		Branches		Functions		Lines	
src	92.97%	1163/1251	84.75%	439/518	95.6%	152/159	92.91%	1154/1242
src/cli	0.86%	2/233	0%	0/172	0%	0/39	0.91%	2/220
src/cli/commands	43.84%	1054/2404	31.61%	427/1351	28.74%	125/435	50.34%	1041/2068
src/fetchers	88.97%	129/145	80%	52/65	92.31%	24/26	88.97%	129/145
src/lockfile	90.48%	228/252	86.13%	149/173	87.5%	21/24	90.48%	228/252
src/registries	92.26%	155/168	86.46%	83/96	83.87%	26/31	92.22%	154/167
src/reporters	69.23%	90/130	56.1%	23/41	62.77%	59/94	69.23%	90/130
src/reporters/console	62.66%	151/241	53.01%	44/83	73.68%	42/57	62.98%	148/235
src/reporters/console/helpers	100%	17/17	62.5%	5/8	100%	8/8	100%	17/17
src/reporters/lang	100%	1/1	100%	0/0	100%	0/0	100%	1/1
src/resolvers	95.45%	21/22	100%	2/2	75%	3/4	95.45%	21/22
src/resolvers/exotics	78.74%	200/254	67.44%	87/129	91.11%	41/45	78.4%	196/250
src/resolvers/registries	76.74%	66/86	53.45%	31/58	100%	7/7	76.74%	66/86
src/util	79.5%	853/1073	73.06%	404/553	85.47%	147/172	79.62%	844/1060
src/util/normalize-manifest	98.91%	271/274	96.54%	251/260	100%	22/22	98.9%	270/273

Code coverage generated by istanbul at Fri Mar 10 2017 21:46:11 GMT+0100 (CET)

Figure 8. Diagram displaying the code coverage per folder.

Initially the coverage report did not show the coverage for completely uncovered files. A pull request was submitted to trigger the coverage for all files in the source folder. The total statement coverage is therefore at the moment of writing 67,18%. Primarily the `cli/commands/` folder is largely untested, with a lot of commands 0% covered. There is no apparent reason, however in the pull request that introduced a lot of the commands, @kittens noted that tests will be deferred to a new PR. Up to this moment, these tests are non-existent and therefore a clear example of technical debt. Other folders (apart from `cli/`) are significantly better covered; the majority of them has a coverage of 80% or higher.

Mitigation of Technical Debt

In order to mitigate as much technical debt as possible, Yarn makes use of the issues and pull requests of GitHub. By reviewing each others work in pull requests, technical debt is prevented as much as possible.

Issues on GitHub are used a lot: on March 11th 2017 more than 700 open issues exist and over 1200 issues are closed. The big issue at the moment is that more issues are opened than closed in the same timeframe. Because of this, it is becoming more and more difficult to monitor all issues. A lot of these issues should have probably been closed, but as the amount is so big, keeping track is difficult.

In order to guide external developers trough the process of contributing, CONTRIBUTING.md lists five steps to follow before opening a pull request:

> 1. Fork the repo and create your branch from `master`.
> 2. If you've added code that should be tested, add tests.
> 3. If you've changed APIs, update the documentation.
> 4. Ensure the test suite passes.
> 5. Make sure your code lints.

Testing and documentation play a role in these steps, contributing to the mitigation of technical debt. Every opened pull request is reviewed by at least one core developer, and if needed, more developers are notified and asked for an opinion. It seems that the core developers have distributed categories of contributions, which can be seen in who replies to a pull request. For example, @bestander's main focus is on contributions related to implementation, while @daniel15 is more focused on reviewing project related contributions.

When someone wants to submit a "substantial" change, the Yarn team requests an additional design process through the yarnpkg/rfcs repository. The description of a substantial change is described in the README of the RFCS repository. In the section Release Management, the RFC process is explained. This process is used to "produce consensus among the Yarn core team".

A great example of the complete contributing process can be found in #2836. Quoting the reaction of @bestander on this pull request:

> Great job on pushing through the whole feature from RFC to great implementation with tests and docs, @dguo!

Conclusion

With the speed Yarn was brought to market, the challenges it faces in the ecosystem, and the dependency on the npm registry, it is maneuvering in a difficult space. However, Yarn has a solid foundation with multiple big companies expressing interest and contributing their time to make sure it is a qualitatively excellent tool. Beyond that, the open source community is very vocal about their issues, needs and desires.

In this chapter, we have analyzed Yarn's architecture from different viewpoints, including the package structure and inter-dependencies. Here, we found that Yarn is ready for growth, with an open structure to add more functionality. We identified technical debt in the codebase, and how Yarn deals with this debt. For some of the issues identified, we have been able to propose fixes by opening several pull requests.

With the adoption of Yarn in the ecosystem, the transparent RFC process, and the solid core team, we are confident Yarn can overcome its challenges and will continue to be a key layer in the JavaScript ecosystem.

References

1. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
2. Sebastian McKenzie, Christoph Pojer, James Kyle. Yarn: A new package manager for JavaScript. https://code.facebook.com/posts/1840075619545360. 2016.
3. Contributor Covenant Code of Conduct. https://github.com/yarnpkg/yarn/blob/12ff2bca446f2173de8c0861cb61b075fbf726f9/CODE_OF_CONDUCT.md. 25 Aug 2016.
4. Yehuda Katz. Why I'm Working on Yarn. http://yehudakatz.com/2016/10/11/im-excited-to-work-on-yarn-the-new-js-package-manager-2/, Oct 11, 2016.
5. Ben Mann. Using Bower with Yarn. https://bower.io/blog/2016/using-bower-with-yarn/. 12 Oct 2016.
6. Google Chrome Developers. What's New in Polymer Tools (Polymer Summit 2016). https://youtu.be/guYHn0P8bKQ?t=17m10s. Oct 18, 2016.
7. Yarn README.md. https://github.com/yarnpkg/yarn/blob/19eb5007510f039c61630948b01a491c3ccdde23/README.md. February 27, 2017.
8. https://github.com/yarnpkg/yarn/blob/89f181491e1258032c2b0365855ee2f1c37a913d/src/reporters/json-reporter.js. March 6, 2017.
9. Carolyn Seaman. Measuring and Monitoring Technical Debt, University of Maryland Baltimore County. https://pdfs.semanticscholar.org/81c0/8b976f959b092f3768c74c4c307cba55a853.pdf. March 27, 2013.
10. Arie van Deursen. Think Twice Before Using the “Maintainability Index”. https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/ August 29, 2014.
11. Yarn Jenkins builds https://build.dan.cx/view/Yarn/ May 5, 2017.

Contributions for DESOSA 2017

This chapter outlines some of the contributions made by several teams to their open source project. Contributions have been categorized based on their types. At the end, a list of all mentioned Pull Requests is provided for more information. It should be mentioned that much more Pull Requests were filed during the course, this chapter only sketches a broad overview of the diversity.

Documentation fixes

Several teams proposed changes to the documentation of their projects. This could go as far as fixing typo's or updating the documentation for deprecated / added features. One of the teams to file such Pull Requests was Kibana (#10709 :white_check_mark:, #10714 :white_check_mark:, #10715 :white_check_mark:). Often, these Pull Requests were merged without much discussion.

Bug fixes

Several teams went through the issue lists of their projects to find bugs to fix. An example would be the bug in the search functionality of Visual Studio Code: when a user searched for two terms in Visual Studio Code, the 'clear results' button was clicked and alt+up was used to get the last search item, the second last was shown.

To get the last history item the function `showPreviousSearchTerm()` was called in the file `src\vs\workbench\parts\search\browser\searchWidget.ts` . This function is shown in Snippet 1

```
public showPreviousSearchTerm() {
    let previous = this.searchHistory.previous();
    if (previous) {
        this.searchInput.setValue(previous);
    }
}
```

Snippet 1 - Original method for showing previous search terms

The problem was however, that this function returns the second last element from an array of previously searched search terms. When a user hits the 'clear results' button, no element is added to this array. This means that after hitting 'clear results' the function `showPreviousSearchTerm()` will return the second last item, which is obviously not the last searched item. First, a solution was created using a flag to indicate that the 'clear results' button was hit, but after looking further into this a new bug was introduced. A better solution was to change the function such that when the search term value is empty the `showPreviousSearchTerm()` should return the current history element instead of the previous. The solution is shown in the Snippet 2.

```
public showPreviousSearchTerm() {
    let previous;
    if (this.searchInput.getValue().length === 0) {
        previous = this.searchHistory.current();
    } else {
        previous = this.searchHistory.previous();
    }
    if (previous) {
        this.searchInput.setValue(previous);
    }
}
```

Snippet 2 - `showPreviousSearchTerm` with the fix applied

To propose this solution for Microsoft Visual Studio Code a pull request was filed (#21859 :white_check_mark:). After signing the CLA the Travis build had to be fixed and some more feedback had to be processed. A maintainer (@sandy081) found a problem with the proposed solution. After resolving this problem the pull request was accepted by the project.

Another bug that has been resolved is a problem with prefix matching within Visual Studio Code. When a user searched for the number '4' in both command palette and the settings editor of Visual Studio Code. The search function did not only match '4', but also matched the letter 'T'. So these search functions matched false positives.

Visual Studio Code calculates the distance between alphanumeric values. This works nicely for lowercase and uppercase letters. However, the distance between `4` and `T` happens to be 32 as well. The original method is shown in Snippet 3.

```
if (ignoreCase) {
    if (isAlphanumeric(wordChar) && isAlphanumeric(wordToMatchAgainstChar)) {
        const diff = wordChar - wordToMatchAgainstChar;
        if (diff === 32 || diff === -32) {
            // ascii -> equalIgnoreCase
            continue;
      }

    } else if (word[i].toLowerCase() === wordToMatchAgainst[i].toLowerCase()) {
        // nonAscii -> equalIgnoreCase
        continue;
    }
}
```

Snippet 3 - Original method `_matchesPrefix`

The pull request that was filed to fix this has been closed (#22743 :x:). The maintainer pushed a fix for the problem which reused an already-existing method for this comparison. This was a better approach than adding the same functionality again.

While some teams fixed reported bugs, other teams also discovered bugs themselves. An example issue was found by team Yarn, where the initial run of the test suite failed on one of the team members computers. The underlying issue was usage of spaces in the local directory of Yarn. Based on a similar issue on the Node repository, the fix was to escape (with quotes), the executing location of Yarn in the test suite. Consequently, a pull request (#2700 :white_check_mark:) was opened and quickly merged thereafter by @bestander.

A similar issue was found, where running the test suite broke on the initial checkout. This time, pre-existing usage of Yarn influenced the outcome of the test suite. The tests were therefore not run in isolation and could be influenced by the configuration of the user. An initial fix was submitted (#2725 :construction:), but was insufficient as @bestander pointed out the test suite should use mocks instead. At the moment of writing, the pull request is still ongoing as mocking the configuration has been unsuccesful thus far.

Several members of team Yarn use Yarn in their daily development toolkit too. In a different course, they were using Yarn as well and discovered a bug. Packages which do not supply binaries would be logged by Yarn. However, the call to the reporter missed an argument of the package name. As team Yarn was familiar with the architecture of the project, finding the issue took little time and #2969 :white_check_mark: was submitted to fix this small issue.

Technical debt

Another popular topic was technical debt. As part of the course the (amount of) technical debt was analysed using tools, such as SonarQube. Teams could use their findings to improve the projects that they were analysing.

For instance, the Kibana team tried to remove unnecessary usage of the Lodash library (#10746 :white_check_mark:). There are multiple methods in Lodash that can easily be replaced by very similar native ES6 methods. These unnecessary lodash methods have been removed in the contribution. Using different tools to look for technical debt the Kibana team found more issues. Some syntax errors were found in the test code. There was an incorrect `.json` file and a quoting issue with a `.sh` file. This was another possibility for a contribution (#10747 :white_check_mark:).

Besides removing packages, outdated dependencies can also be upgraded. The Yarn team did a cleanup (#2812 :white_check_mark:) to upgrade the old dependencies. Additionally, several breaking changes of the typechecker Flow were fixed. This pull request was merged within a couple of days by @bestander. As a consequence of this pull request, now whenever Flow releases a new version, Yarn upgrades within a couple of days in contrast to the months it took before.

While external dependencies can be outdated, internal function calls can be too. An example is an internal logging call in Yarn, which was using the `console` directly, rather than their new `Reporter` infrastructure. Team Yarn submitted a simple line change (#2844 :construction:), but @bestander pointed out that there was an undocumented reason for sticking to `console`. In fact, @kittens pointed out in a different pull request (that is still open) that it would break external tooling integration. Up to this point, there is no test that verifies this issue to ensure no breaking changes are introduced. For this reason, #2844 is blocked and left open until the underlying issue has been resolved.

New features and behavioral changes

Besides documentation/bug fixes and resolving technical debt, teams also contributed new features and (potentially breaking) behavioral changes.

The direct competitor and inspiration of Yarn, npm, changed in its latest major version a behavioral change regarding the output of error logs of npm. Since compatibility with npm is one of the main goals of Yarn, it is important to update the code of Yarn accordingly. Therefore, #2870 :construction: was submitted to fix this. This pull request also included a brand new test suite to test the entrypoint of Yarn: `src/cli/index.js`, which was untested up to this point. Initially, @bestander was reluctant to change the behavior, but pointed out a different and more user-friendly solution. At the moment, the pull request still has to be updated to incorporate the feedback.

An interesting side-note is that in #2870 a different bug was found in the `--cache-folder` option on the commandline. While the pull request is not merged, this particular bug fix was incorporated by Facebook employee @arcanis in #3033.

Measuring and maintaining good code coverage is required to remain with a healthy software project. Correctly calculating code coverage is therefore crucial, to measure which parts of a project require tests to keep confidence in the product. While investigating technical debt, team Yarn discovered that the code coverage tool of Yarn was incorrectly configured and did not report completely untested files. Luckily, Jest (the testrunner used by Yarn) incoporates code coverage calculation and has configuration available. A one-line change pull request (#2892 :white_check_mark:) was submitted to fix this inconsistency and sadly point out that the code coverage of Yarn was 20 percent lower than previously reported. As a consequence, the Yarn core developers shifted more focus on tests and now more strictly enforce tests when integrating pull requests.