

WatchDog For IntelliJ: An IDE Plugin To Analyze Software Testing Practices

Master's Thesis

Igor Levaja

WatchDog For IntelliJ: An IDE Plugin To Analyze Software Testing Practices

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Igor Levaja
born in Pozarevac, Serbia



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

WatchDog For IntelliJ: An IDE Plugin To Analyze Software Testing Practices

Author: Igor Levaja
Student id: 4318110
Email: I.Levaja@student.tudelft.nl

Abstract

Software testing is as old as software development itself – they could not exist one without the other. However, are they equally important? Do software developers devote an equivalent amount of time to both produce software and to test it? An ongoing study of the TestRoots project aims to examine and improve the state of the art of software testing and answer those questions, by observing developers' everyday behavior.

In order to support this effort, we evolved WatchDog, a single-platform software, to become the scalable, multi-platform and production-ready tool which assesses developer testing activities in multiple integrated development environments (IDEs). We further used WatchDog platform to perform a small-scale study in which we examined testing habits of developers who use IntelliJ IDEA and compared them to those of the Eclipse IDE users. Finally, we were able to confirm that IntelliJ users, similarly to the Eclipse users, do not actively practice testing inside their IDEs.

Thesis Committee:

Chair:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	M. Beller, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Zuniga, Faculty EEMCS, TU Delft
Committee Member:	Dr. R. Krebbers, Faculty EEMCS, TU Delft

Preface

Doing these master studies was an amazing and a challenging experience, enriched with people who have been the most important part of it. I owe my biggest gratitude to the Test-Roots team, especially to my supervisors Moritz Beller and Andy Zaidman, who gave me the opportunity to work with them on my master thesis project – developing a WatchDog platform for analyzing software testing practices. They provided me with enormous support, inspiration, guidance, understanding and constructive feedback, thus making my work on the thesis the culmination of this whole journey. Once again, thank you for everything!

During the studies, I have been privileged to meet the most amazing people, who have influenced not only those years I spent in Delft, but also my whole life. Moreover, encouragement and motivation from my lifelong friends and colleagues from Serbia kept me moving forward even in the hardest moments, when I questioned myself. The biggest support came from my family, which has always been the reason I made it this far.

To each one of you, even though the words are not enough to express how grateful I am, I still want to say: thank you for always being there!

Finally, to my parents, thank you for everything! Without your unconditional and wholehearted support, I would not be able to pursue my dreams!

Igor Levaja
Delft, the Netherlands
November 24, 2016

Contents

Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Motivation	1
1.2 Research questions	2
1.3 Thesis outline	3
2 Background And Related Work	5
2.1 Software testing	5
2.2 Data collection and reporting software tools	8
2.3 TestRoots	10
3 Building A Research Software Tool: WatchDog Implementation Process	11
3.1 Eclipse plugin	11
3.2 Migration from the Eclipse plugin	14
3.3 New architecture	16
3.4 Building multi-platform tool: Summary	18
4 Study Design	23
4.1 Practitioner’s perspective	23
4.2 Backend logic and metrics	27
4.3 Study participants	29
5 Analysis And Discussion Of The Research Questions	33
5.1 Analysis of the research questions	33

CONTENTS

5.2	Discussion	39
5.3	Threats to validity	40
6	Conclusions And Future Work	43
6.1	Conclusions and contributions	43
6.2	Future work	44
	Bibliography	45
A	Travis Scripts For Building WatchDog Plugins	49
B	Maven POM Files	53

List of Figures

3.1	WatchDog real-time statistics window.	13
3.2	WatchDog real-time statistics window after refactoring.	14
3.3	WatchDog's three layer architecture (source: [5]).	20
4.1	Exemplary workflow visualization with intervals (source: [3]).	24
4.2	WatchDog's immediate statistics view in the IDE (source: [27]).	25
4.3	WatchDog's project report (source: [5]).	26
4.4	WatchDog's intervals.	27
4.5	WatchDog's UML diagram.	31
4.6	Operating system developers use and their programming experience.	32
5.1	The immediate reactions to a failing test.	36
5.2	The delta between estimation and reality.	38

List of Tables

4.1	Overview of WatchDog intervals (based on:[3]).	28
5.1	Descriptive statistics for important variables. Histograms are in log scale. . . .	35
5.2	Actual and estimated time distributions	39
5.3	Reading and Writing distributions	39
5.4	Comparison of Eclipse and IntelliJ conclusions	42

Chapter 1

Introduction

In this chapter we provide a motivation for this study, define research questions and outline the thesis structure.

1.1 Motivation

More than 40 years ago Frederick Brooks published his famous book on software engineering and project management – “The Mythical Man-Month” [7]. Many ideas and conclusions from this influential work are widely quoted and discussed even today, such as “adding manpower to a late software project makes it later” and “fixing a defect has a substantial chance of introducing another”. However, there is one statement regarding software testing that has not received any follow-up – he estimated that among examined projects “most did indeed spend half of the actual schedule for that purpose (testing)”. This hypothesis survived the whole evolution of software engineering during the last four decades and was taken “as is” ever since. However, software engineering has changed tremendously during that time – new programming languages were created (e.g. C++, C#, Python, Java, JavaScript, PHP) and new integrated development environments – IDEs – have been developed, starting from historical Maestro I (first IDE, presented in the same year as Brooks’ book) and Softbench (1989), to modern, intelligent platforms, such as Visual Studio, Eclipse, NetBeans and IntelliJ IDEA. Furthermore, new engineering practices and programming paradigms (structured programming, object-oriented programming), as well as new development methodologies (globally distributed or test-driven software development) have also been established during this period. In the light of these advances, the topic of software testing has to be reinvestigated thoroughly in order to examine whether Brooks’ observation still stands.

To challenge Brooks’ claim, a prototypical Eclipse plugin called WatchDog has been developed [4] and tested, first among students and then with developers all over the world [3]. However, the presented results lacked generalizability – the major limitation of the experiment setup arose from the fact that it relied only on one IDE, Eclipse. In order to confirm those findings, it was necessary to perform a large-scale study and to extend the research and include other environments and settings. This gave a starting point and clear motivation for this work – implement WatchDog for another IDE. Choosing the next IDE

was not a hard decision – in recent years, many developers have started switching from Eclipse to the then recently open-sourced IntelliJ IDEA, which is now the second most used Java IDE in the market which was also confirmed by the input we got from our potential users. Furthermore, not only differences between IDEs mattered, but also similarities – both Eclipse and IntelliJ are built using Java and both support testing out-of-the-box. Finally, we cannot disregard the fact that IntelliJ IDEA is just one among a few other JetBrains’ IDEs¹ for different languages, such as Ruby, Python and JavaScript, so the plugin developed for one of these has a potential to cover others as well with minor effort.

Following our idea to develop the same plugin for another IDE, another problem emerged – how to use the existing plugin, tightly-coupled with a specific IDE, and create a solution that could be easily extended to even more platforms? Every IDE is specific and independent, and there is no easy way just to reuse an existing plugin in new environment. This problem introduced some new challenges, but also the opportunity to fully explore the tool building process in software engineering research and to give some guidelines for future use.

1.2 Research questions

The work presented in this thesis has two objectives - the first is to report on our experiences in building a multi-platform, production-ready tool used in software engineering research (*G1*); the second is to perform a replication study of the previous experiment [3] and compare results produced by the same tool, but in different environments (*G2*).

From the perspective of a (scientific) software engineer, it is essential to come up with a solution that is efficient, easily maintainable and expandable, platform independent, and, preferably, free of bugs. Because of the limited resources (having typically less than one full-time developer available at any given time), building a stable and scalable platform is an ongoing challenge in academic software development we have already faced and upon which we want to reflect by sharing our experiences and guidelines.

In order to test the newly created platform, we perform the initial analysis of the collected data from IntelliJ users and compare our findings to the results of Eclipse users, presented in the previous work [3]. This small-scale replication study will be used to generalize the previous findings by comparing and contrasting results of the same experiment executed in different environments. Moreover, with this study, we also verify our experiment process and design, which can strengthen the validity of the original work. Although our data-set is significantly smaller than the one in the original work, we still decided to re-use the main research questions from that study. Those research questions are the following:

RQ1 When and Why Do Developers Test?

RQ2 How and Why Do Developers Run Tests?

RQ3 How Do Developers React to Test Runs?

¹<https://www.jetbrains.com/>

RQ4 Do Developers Follow Test-Driven Development (TDD)?

RQ5 How Much Do Developers Test?

1.3 Thesis outline

The thesis is organized as follows – in the chapter following this introduction, we provide background information regarding software testing and discuss related work (Chapter 2). In Chapter 3 we explain the implementation and evolution of our tool and give guidelines for multi-platform plugin development. Following up we address experiment design and explain our approach in Chapter 4. The experiment and its results are presented and discussed in Chapter 5, while in Chapter 6 we conclude with a summary of our findings and contributions and propose future work.

Chapter 2

Background And Related Work

In this chapter we explain the fundamentals of software testing and give overview of the related work. Furthermore, we also explain basic concepts of software tool building, especially the development practices for IDE plugins, and provide definitions of essential terms used in this thesis.

2.1 Software testing

Software testing. The purpose of software testing and analysis is either to assess software qualities or else to make it possible to improve the software by finding defects [30]. It follows from this definition that software testing is a very broad topic – indubitably, it is as old as software development itself. To illustrate the age of this subject, we should look at the classification proposed by Gelperin and Hetzel [13] which divides software testing into five phases:

- Until 1956 *Debugging* oriented, when software testing was tightly coupled with debugging;
- 1957 - 1978 *Demonstration* oriented, when debugging was distinguished from testing, which had to show that software satisfies the requirements;
- 1979 - 1982 *Destruction* oriented, in which the goal was to find errors;
- 1983 - 1987 *Evaluation* oriented, when software testing had to provide product evaluation and quality measurement;
- From 1988 *Prevention* oriented, when testing aims to demonstrate that a product satisfies its specifications and to detect and to prevent faults in it.

Although each of the five directions can be observed as a separate part, all of them are driven by the same goal – achieve the highest quality of the software by removing defects from it. Software under test should at least meet the following conditions: to adhere to the

2. BACKGROUND AND RELATED WORK

defined requirements, both by design and stakeholders; to execute correctly within acceptable time and with an appropriate input; to be sufficiently usable and resistant to errors and failures; and to be easily deployable to the designated environments [29]. This list can certainly be extended with many other conditions and as large as it gets, the number of possible testing scenarios for every single software product is even bigger, possibly infinite, making it practically impossible to have a 100% tested product. Therefore, various testing methods and approaches were defined, usually dependent on the whole software development process and the level of the testing needed in a particular case.

By the broadest classification software testing can be divided into two categories – *static* and *dynamic* testing. Static testing implies methods such as (code)reviews, inspections and verification (correctness proof), whereas dynamic testing means that a set of test cases is executed against the software under assessment. Depending on the level on which those test cases are executed, we can divide testing into white- and black-box testing [21]. *White-box* testing is intended for internal structures of the observed software and knowing its implementation is required for writing test cases. On the other hand, in *black-box* testing software is observed as a “black box”, and its functionality is tested from the perspective of a user – without any knowledge of internal implementation. There is also a hybrid approach of the previous two, called *grey-box* testing, when system is tested as a whole (like black-box), but with knowledge of its internal structure.

Finally, fine-grained classification of software testing defines the following levels (lowest to highest):

- Unit testing;
- Integration testing;
- Acceptance testing;
- Refinement testing;
- System testing.

Unit testing, also known as *component testing*, is the representative of the white-box testing practices. It is firmly related to production code and it covers code paths (code branches, edge cases), but also individual components and functions. Interactions between those components are covered by *integration tests*, which are a form of grey-box testing. *Acceptance tests* belong to the black-box group of testing and they ensure that user’s paths through the system function correctly and meet client’s requirements. In addition to them, *refinement tests* consider end-user experience and ensure its usefulness, usability and value. Finally, *system testing*, a typical example of black-box testing, aims to cover the system as a whole and catch regressions.

Due to the nature of black-box approach, writing tests for these purposes does not require knowledge about implementation and it does not have to involve people who worked on software development – often, there are teams of testers, designers, user-experience experts or project owners, responsible for that. On the contrary, writing white- and grey-box tests requires extensive knowledge and understanding of the production code. People who are most familiar with code are those who write it – software developers. Since our study

aims to examine how much software developers test, unit and integration testing are subjects under scrutiny in our experiment, while we explicitly leave out external means of testing.

2.1.1 Software testing in IDEs

For building software products, programmers use a software application which provides various functions to facilitate the development process. This application is called the Integrated Development Environment (IDE) and its purpose is to maximize the developers' productivity by supplying them with all the necessary tools to build software, such as compiler, debugger or deployment mechanism. From our point of view, one of the most important functionalities in the IDE is a tool for testing – nowadays, most modern IDEs come already equipped with such a tool. Since IDEs are generally intended for developers, the focus is on the white-box forms of testing, namely unit testing and integration testing to some extent.

To illustrate this, we can look into two popular IDEs [8, 10], Eclipse and IntelliJ IDEA, for the popular programming language, Java [9, 11]. Java's testing framework, called JUnit¹, is an open-source project distributed as a library which can be included at compile-time in any Java project. This framework can be used for writing both unit and integration tests and both Eclipse Java Development Tools edition (JDT) and IntelliJ IDEA come with integrated JUnit support, which facilitates writing and running test cases in IDE. Namely, JUnit tests are first-class citizens in these IDEs, so it is possible to automatically create test case classes (in the same way it is possible to create Java classes, interfaces or packages), together with method stubs for setting up and tearing down the state of classes under test. Furthermore, running the test suite is enabled in the same way as normal or debugging executions, by providing developer with a dedicated screen for examining the results. Beside JUnit, IDEs also support other testing frameworks, such as Mockito² and Powermock³, which come as extensions to JUnit and facilitate mock based testing, when it is necessary to mock needed dependencies.

2.1.2 Empirical software testing

Software testing has also been the subject of various research in computer science. Back in 2006 Runeson performed a survey of unit testing practices in several companies [24]. In this survey participants discussed unit testing definitions, strengths and weaknesses according to their companies' internal policies. The results of this questionnaire show the importance of clear and shared understanding of unit testing. The need to understand testing practices is further expressed in Bertolino's influential work [6] which provides a roadmap for software testing research by discussing current and future challenges. Pinto et al. [22] and Zaidman et al.[31] studied tests evolution at the commit level, namely creation, deletion and modification of tests in practice (former) and test suite co-evolution with production code (latter). While these two efforts have many similarities with our research, there is also a significant difference: the subjects of observation were projects from version control systems (software

¹<http://junit.org/>

²<http://mockito.org/>

³<https://github.com/jayway/powermock>

2. BACKGROUND AND RELATED WORK

repositories), which means that they only learned from history and differences between versions, while we want to examine developers' testing behavior in real-time and understand how they *actually* test.

Kochhar et al. [16] studied more than 20,000 non-trivial open-source software projects and found that 61.65% of them contained at least one test case. The similar study has been performed multiple times by a software analytics company Takipi⁴ that analyzed usages of Java libraries in GitHub. In their first research [25] that was comprised of 10,000 most popular projects, they found that JUnit held the first place with 30.7% share, while Mockito had 6.83%. In their second study [26], when they analyzed library import statements of 5,216 Java projects, 64% out of 11,939 unique libraries imports were set on JUnit, while there was also extensive use of the Mockito framework (10.72%).

On the other hand, software testing was also examined in various surveys. LaToza et al. [17] conducted a set of questionnaires and interviews among Microsoft engineers about their work habits, in which 79% of the participants indicated the use of unit tests. Despite many experiments in this area during the last decade, a lot of questions still remain unanswered. In empirical software engineering, they appear as the second most interesting topic, as it was concluded by another survey conducted in Microsoft about current challenges [2] in software engineering.

2.2 Data collection and reporting software tools

For many computer science experiments that involve collecting real-world data, first it is necessary to build a software tool to fulfill that purpose and then to deploy that tool to an appropriate environment. While many tools are stand-alone applications which do not require to be delivered to end users (mining software repositories, for example), there are also plenty of distributed platforms which demand user acquisition and further effort to collect data from them. In academic research it is common to provide study participants with the feedback on the data they provide – although some authors tend to notify them about the published work and results, this generally comes a bit late, usually a few months after the data collection phase. Another way would be to apply immediate data analysis and give them real-time feedback, which can require investing more time into tool building, but can also prove beneficial in obtaining users, especially if the results are useful to them. Since many developers spend most of their time working in an IDE, data collection process can easily be attached to it. Furthermore, modern IDEs are powerful software platforms that enable background data analysis in real-time and graphical user interface (GUI) components to support displaying statistics directly in them. They are also designed having extensibility in mind, due to specific users' needs. This feature comes in a form of plugins, which are made either by creators of an IDE or by third-party developers. Many tasks can be achieved with plugins – automation of procedures, monitoring user's behavior and enhancing development process, data collection and analysis. Numerous tools that instrument the IDE to assess development activity in vivo have been developed [3] and can be divided into two groups: 1) data-collecting plugins, made in academic setting, for research purposes, and 2)

⁴<https://www.takipi.com/>

commercial plugins, which offer general reporting to developers regarding their behavior. Both these groups are of interest to our study, since our plugin has an intermediate position, as it allows both functionalities.

1) *Data-collecting tools.* Collecting data about events in IDE is not a novel thing – Spyware [23], an Eclipse plugin, instruments the IDE to perceive code changes as first-class citizens and accurately model software evolution in real-time. Following up on this concept, Hattori et al. developed a plugin for the same IDE called Syde [14], which aimed to ease team collaboration in software projects by making developers mutually aware of source code changes. While Robbes et al. used single-developer approach, Syde extended Spyware’s change-based software evolution model by introducing multi-developer approach and client-server architecture (similar approach was taken in our research, which will be discussed in the following chapter). With another family of Eclipse plugins CodingTracker and CodingSpectator, Negara et al. [20] collected code editing events and used them with a proposed refactoring inference algorithm to analyze manual and automatic, IDE-supported, code refactorings (differences in time needed to perform operations, quantity and size of operations, clustering). The “Change-Oriented Programming Environment”⁵ broadly captures all IDE interactions, targeting the small audience of developers employing TDD [3]. Their research is, beside ours, one of a very few that covers both Eclipse and IntelliJ IDEs. Minelli et al. [19] investigated usage of IDE’s graphical user interface (GUI) from a program comprehension point of view, for example: how much time is spent on reading versus editing code, or navigating through project files. Finally, the “Eclipse Usage Data Collector”⁶ was a framework run by the Eclipse Foundation from April 2008 to February 2011 and its large data set was primarily used for collecting fine-grained and Eclipse-specific data in order to gather information about how the community was making use of the Eclipse technology.

2) *Reporting tools.* Formerly QuantifiedDev, now 1self.co⁷ aims to provide developers with a full-fledged analysis platform on their development practices compared to their general habits. This platform collects data from various sources, such as IDEs (IntelliJ, Visual Studio), software repositories, social networks, sensors and applications from mobile phones, and connects disparate events from all sources. In IDEs they analyze general user’s activity supplemented by specific events, such as software build status, and correlate that with, for example, the temperature information from the mobile phone and behavior on social network. Codealike⁸ has a similar program comprehension focus as the work of Minelli et al., and provides users with advanced reports on their development behavior. However, even though they are monitoring many crucial IDE activities (editing, reading, debugging and building), testing is left out. Other examples of activity monitoring plugins are Codeivate⁹ and WakaTime¹⁰. Both tools are introduced as time-tracking plugins, which collect information about how much time developers spend writing code, accompanied by some high-level information, such as project or file names and programming languages, but

⁵<http://cope.eecs.oregonstate.edu>

⁶<https://eclipse.org/epp/usagedata>

⁷<http://www.1self.co>

⁸<https://codealike.com>

⁹<http://www.codeivate.com/>

¹⁰<https://wakatime.com>

both lack fine-grained intervals that previous tools have. The only minor difference between them is that Codeivate, which is closed-source, currently supports Sublime Text Editor and, recently, JetBrains' family of IDEs, while WakaTime provides 31 commercial, open-source, text editors and IDE plugins.

2.3 TestRoots

In order to fill the knowledge gap, dive deeply into the topic of software testing and understand how software developers test their products in the real world, the TestRoots¹¹ team developed an IDE plugin called WatchDog, on which basis this thesis is built upon. Back in 2013, WatchDog used to be a prototypical plugin, made for Eclipse IDE, that was able to monitor single developer's programming behavior. However, this prototype had not been made to scale up to more, possibly hundreds or even thousands of users, so further development was needed before releasing it publicly. The release candidate version was used for examining testing practices of 40 students during a programming course at TU Delft, which resulted in the initial ICSE NIER paper [4]. This experiment and its results were enough for a proof of concept, so WatchDog was ready to be released publicly and developers all over the world got a chance to use it. Having more diverse users from real world resulted in another, large-scale research [3] that analyzed testing behaviors of 416 developers who worked on 460 unique projects. However, being a plugin that supports only one IDE, "more [...] research in different environments and settings is needed", as noted by the reviewers. For various reasons already mentioned in Section 1.1, we made the decision to implement WatchDog for JetBrains' IntelliJ IDEA and refactor it into the multi-IDE platform which we present in this thesis and in demonstration paper [5].

¹¹<http://testroots.org/>

Chapter 3

Building A Research Software Tool: WatchDog Implementation Process

Creating a software platform to facilitate the academic research can be very different from usual, industry, software development. For many reasons, listed later in this chapter, the process would probably face a lot of challenges, especially when taking novel approaches and creating solutions that are likely among the first of their kind. In this chapter we present the process of building a multi-platform IDE plugin, called WatchDog (*GI*). During this process, we took a novel approach of creating the platform-specific tools on the top of the common shared module and here we give advice to academic and industrial software developers on how to create a family of multi-IDE plugins on the basis of our own experiences with WatchDog. We do this by describing its evolution from the prototype version to the multi-platform research software tool, scalable up to thousands of simultaneous users.

3.1 Eclipse plugin

The very first version of the WatchDog was created in 2012 by Wouter Willems. This prototype consisted of the monolithic Eclipse plugin which had been monitoring developers' behavior, while creating and locally storing the output in XML format. Its requirements were defined on the basis of companies involved in the research, which were using a specific configuration for the development process, namely programming language Java, JUnit testing framework and Eclipse IDE. The produced output files had to be sent manually by developers who were using the plugin, which significantly limited the scalability of the tool – the first research study covered data from only 9 developers.

Starting from this initial prototype, in 2014 Software Engineering Research Group (SERG) at TU Delft evolved WatchDog into an open-source and production-ready Eclipse plugin [4]. To fulfill the primary requirement – to have a reliable transfer logic that would utilize internet connection in order to collect usage data automatically – WatchDog had to adopt a new client-server architecture, following the same idea from Spyware and Syde [23, 14]. The complete internal logic is explained in detail in Chapter 4, whereas in this chapter we give a high-level overview and brief explanation of WatchDog, while we point

3. BUILDING A RESEARCH SOFTWARE TOOL: WATCHDOG IMPLEMENTATION PROCESS

out its development and evolution process.

WatchDog plugin is available for download from the Eclipse Marketplace¹ and Test-Roots update site². When users download it and install in the IDE, they are asked to register themselves and fill in a survey regarding their own estimation of programming habits about currently open workspace(s). For clarification, while Eclipse considers projects as modules belonging to a single workspace, in our study we observe a single *workspace* as a *project* without further granulation into modules, so in the remaining of the thesis we equate both terms (*workspace* and *project*). Users who register accounts can associate them with as many projects as they want. In that way, each developer will have their own unique user ID, while each of their project registrations creates a separate project ID. A choice whether or not WatchDog should be active for each of their projects is left to them – upon opening or creating a new workspace they are asked to opt in or out. After the registration, assuming that they opted for using WatchDog in the current project, they can continue with their usual development activities, while WatchDog is running silently in the background.

The client of the application, the Eclipse plugin, is activated during every start-up of the IDE and, in this phase, it creates listeners to receive internal IDE and UI events. These events, based on their type, are grouped into intervals which describe user's behavior in the IDE. In addition to their type, intervals also contain information specific for the events included in them. Records about these events and intervals are transferred to our server, while we also store them locally so we can provide real-time statistics from the last development hour to our users, as shown in Figure 3.1. Furthermore, upon the completion of the first research study, developers were promised a full development report, generated through our data analysis framework, which will compare their activity with fellow WatchDog users around the world.

For a research project that relies on data analysis, user acquisition is a very important aspect. However, a few months after WatchDog public release, statistics showed that the number of registered users exceeded the number of registered projects, although it would be expected otherwise – that a single user registers at least one or, probably, more projects. Although it is questionable what can be the exact explanation for it, one possible reason could be the length of the initial survey – users were asked to complete two registration processes after plugin installation, first to register themselves and afterwards their project, and it is likely that a user will simply quit after the first registration and ignore the latter. To address this issue, it was necessary to facilitate a survey process and make it shorter and more convenient, which was achieved by merging two registration wizards into one, and making the length of it twice shorter than the previous two together³.

While we identified several reasons for the difficult user acquisition, such as privacy-concerns, a lack of an incentive to use a tool without immediate benefit and use of other IDEs[4], we chose to give even more immediate insights to our users as a next step. Therefore, we introduced two additional real-time statistic overviews for our users: Eclipse *perspective* activity overview, which shows how much time developers spend in the most common Eclipse perspectives (Java, Debug, etc.); and testing execution overview, which shows

¹<https://marketplace.eclipse.org/>

²<http://updatesite.testroots.org>

³<https://github.com/TestRoots/watchdog/issues/150>

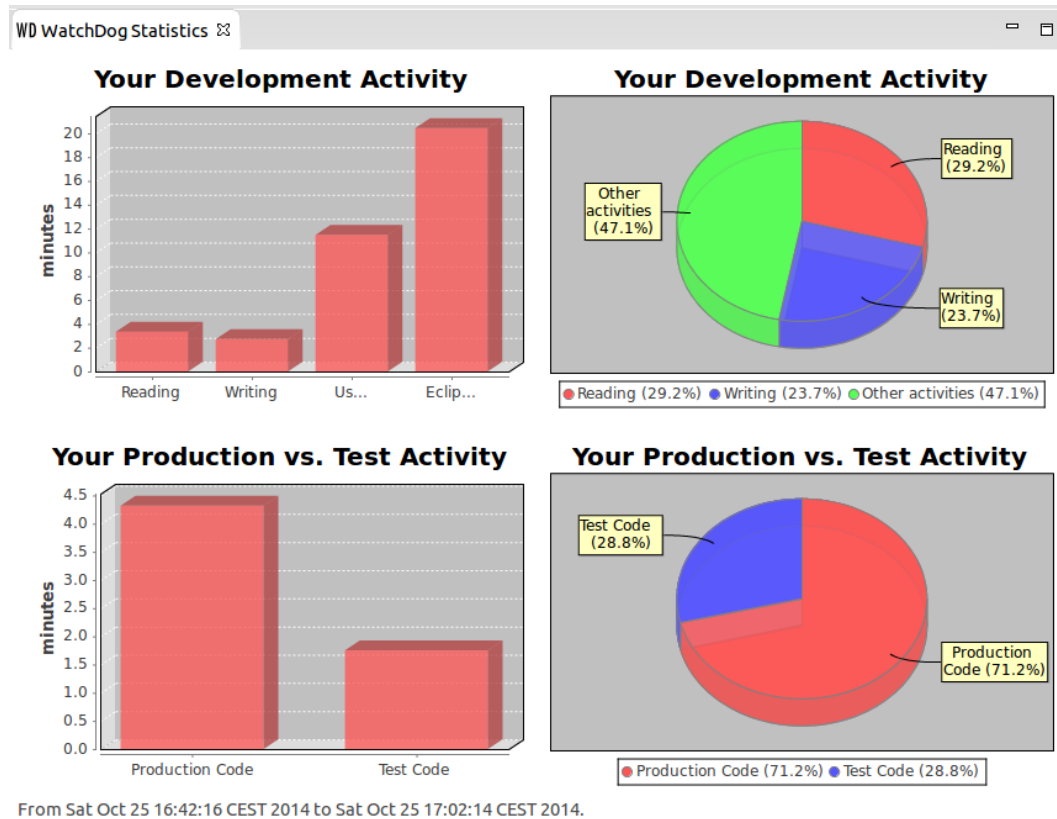


Figure 3.1: WatchDog real-time statistics window.

statistics about successful and failed test executions, followed by their durations⁴. Furthermore, time intervals for live statistics were extended, so we made it possible to select time periods of up to eight hours, which enabled developers to analyze their behavior at the end of the work day⁵. The newly added features are displayed in Figure 3.2. In the following period, we observed that our effort in the newest update affected the discrepancy between the number of users and the number of project registrations – it made the growth of both types of registrations consistent, at even higher rate than before.

Although WatchDog already had hundreds of users by the end of 2014, all of them were using the Eclipse IDE for development, which was previously identified as one of the problems for user acquisition. This revealed a space for expansion – first by conquering another popular Java IDE - IntelliJ, and, second, by creating a solution that can be ported to even more platforms. Starting from the WatchDog version 1.3.0, we began migrating the existing solution to the IntelliJ IDE.

⁴<https://github.com/TestRoots/watchdog/issues/142>

⁵<https://github.com/TestRoots/watchdog/issues/144>

3. BUILDING A RESEARCH SOFTWARE TOOL: WATCHDOG IMPLEMENTATION PROCESS



Figure 3.2: WatchDog real-time statistics window after refactoring.

3.2 Migration from the Eclipse plugin

In this section we explain the process of creating a functional WatchDog prototype of the IntelliJ plugin, based on the implementation of the existing Eclipse plugin.

The initial investigation of the IntelliJ plugin development consisted of gathering as much information as possible from the various sources in order to get familiar with the internal implementation of IntelliJ IDEA and recommended practices for its development and extension. As anticipated, good starting points for this step are JetBrains website⁶, official listing of the *IntelliJ Platform Software Development Kit (SDK)*⁷ and the source code of the IntelliJ IDEA Community Edition⁸.

IntelliJ IDEA is mostly written in Java, following its coding style and conventions. However, code and API documentation (Javadoc, for example) are scarce. Until the version 13.0.0 (dated December 2013) IntelliJ IDEA source code was available on GrepCode⁹, which provided a better and more systematic look into internal implementations. However, since then, many APIs have been changed or deprecated, so beside the official website and the community discussion board, exploration of the IntelliJ source code is necessary upon every new release.

As we explored the IntelliJ SDK and compared it to the Eclipse implementation, we came up with two possible approaches for the IntelliJ plugin implementation: the first would

⁶<https://www.jetbrains.com/help/idea/2016.1/plugin-development-guidelines.html>

⁷<http://www.jetbrains.org/intellij/sdk/docs/>

⁸<https://github.com/JetBrains/intellij-community/>

⁹<https://goo.gl/ALJXZy>

be to implement a new plugin from scratch, independently of the existing solution. The second approach implied an implementation that would be as similar as possible to the Eclipse plugin. While the first method could possibly facilitate the use of IntelliJ API, the second method, which assumed slow and careful code migration and adaptation to the new environment, appeared to be the better choice for the following reasons:

- The existing Eclipse plugin was stable, in production, and the code quality was high. The whole plugin was covered with unit tests;
- The logic would stay the same, thus facilitating future maintenance and establishing the basis for future expansions;
- The same server application would be able to handle both client plugins simultaneously.

The possible drawbacks of this approach were the potentially large amount of code clones, different APIs (some functions that exist in Eclipse are either not available or behave differently in IntelliJ), and the implementation of the logic that was not reusable.

3.2.1 Code organization

While we will not go further into the WatchDog logic in this chapter, we still need to give a brief overview of its code organization in order to be able to discuss the process of its implementation. Initially, WatchDog consisted of the WatchDog Server and the WatchDog Eclipse plugin, which had two parts – plugin production code and unit tests. Production code had three packages: *logic*, which accommodated the whole back-end logic; *ui*, which enclosed all necessary user interface elements; and *util*, which provided various utility functionalities.

Back-end logic of the plugin was divided into four smaller packages: *document*, which contained classes responsible for file analysis; *interval*, accountable for creating, managing, storing and transferring WatchDog intervals; *network*, in charge of communication with the server; and *ui*, which kept the IDE-monitoring logic (such as observing reading, writing and other types of events).

3.2.2 IntelliJ plugin prototype

The implementation of the new plugin started by studying thoroughly the infrastructure of IntelliJ IDEA and by comparing it to Eclipse. By finding similarities and differences between the two IDEs, it was easier to identify parts of the plugin which had to be adapted to the new environment.

The first step of creating an IntelliJ plugin prototype was to migrate low-level code from Eclipse plugin, such as enum types and data classes, followed by the platform-independent code, that is, classes which do not import nor rely upon any Eclipse API. After that, we had to implement listeners for capturing user's actions in IntelliJ in the same manner as we do it in Eclipse. Together with the interval management logic, at this point of time we were already able to execute basic functionalities of WatchDog on IntelliJ platform.

3. BUILDING A RESEARCH SOFTWARE TOOL: WATCHDOG IMPLEMENTATION PROCESS

The biggest issue that we have encountered in the IntelliJ plugin development phase was a “ClassLoader issue”: plugins have their own class loaders (in order to allow the usage of different versions of a same library between plugins and IDE), and this class loader was incompatible with reflection-enabled libraries, in our case MapDB, which we use for storing our collected data locally. Therefore, we had to wrap every operation from this library with the following two methods, in order to exchange the class loader manually:

```
protected void replaceClassLoader() {
    oldClassLoader = Thread.currentThread().getContextClassLoader();
    Thread.currentThread()
        .setContextClassLoader(WatchDogStartUp.class.getClassLoader());
}

protected void resetOldClassLoader() {
    Thread.currentThread().setContextClassLoader(oldClassLoader);
}
```

Building our solution continued by enriching our application with the rest of necessary features, first being network logic for transferring collected data to our server. We continued by adding plugin start-up logic and implementing graphical user interface (GUI). These GUI components: registration wizards, preferences page (for managing plugin settings) and in-IDE statistics display, had to be created completely from scratch because IntelliJ uses Java’s Swing¹⁰ library for graphical components, while Eclipse is built with The Standard Widget Toolkit (SWT)¹¹. Finally, we finished this prototype¹² by releasing its first stable version (0.9.1) on JetBrains plugin repository¹³.

3.3 New architecture

The successful implementation of the WatchDog IntelliJ plugin prototype was enough to prove that the same concept, which already existed for Eclipse, can be applied to the other IDEs. However, following the same approach – migrating a plugin from one platform to another – could easily be considered as unnecessary and over-engineering, while we should not disregard the amount of possible code clones as well.

Given the fact that resources for tool development are limited in academia, but also in the open-source community and smaller companies, creating an easily maintainable software platform can be a significant challenge. This is exactly what we had experienced with our tool which, at that moment, consisted of the server application and two, seemingly, independent plugins with a lot of code clones. To ease the future maintenance, but also to open space for new development efforts and potential expansions to other IDEs, we had to change WatchDog’s architecture and refactor both plugins.

¹⁰<https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>

¹¹<https://www.eclipse.org/swt/>

¹²<https://github.com/TestRoots/watchdog/issues/187>

¹³<https://plugins.jetbrains.com/plugin/7828>

Our next step was obvious – extract all code that exists in both plugins, make it completely platform-independent and build it as a separate “core” project, available in compile-time for both plugins¹⁴. While we could have regarded this common project, named *WatchDogCore*, as a separate library, developed it independently of plugins and simply imported it as a JAR library, we opted for another, novel approach: we set up WatchDogCore to be both Eclipse and IntelliJ project at the same time and available to both plugins in compile-time as another module. In practice, this meant that we could open both IDEs at the same time and have this project available in both of them simultaneously: a change of code in one IDE would be immediately available in another, thus significantly easing the maintenance of both plugins at the same time. The benefits of this approach were evident shortly – it notably facilitated the architecture refactoring process.

While for the majority of the classes it was easy to determine whether they belong to the Core or to the platform-specific project, there were a few classes which would, logically, belong to the Core, although they were tightly coupled with the specific IDE. For these classes, we made use of object-oriented paradigm and design patterns [12], such as Class Adapter pattern and Template Method pattern (which was applied to mitigate our “Class-Loader issue” in IntelliJ), where we abstracted over platform-specific code and made Core classes platform-independent, while each plugin project is able to override them with its own implementations.

The reasonable concern can be the question: what would happen if we wanted to develop a plugin for an IDE that would not support importing our WatchDogCore project into it? While our approach implies project import, creating a stand-alone library, which can be used in any IDE, can always be an option – even though the Core module is packed separately in our production releases.

3.3.1 Automating build process and Continuous Integration

As it was already mentioned, the initial WatchDog Eclipse plugin was equipped with a thorough test suite. Other than using the IDE to deploy the plugin, it could also be built with the Maven build automation tool, which at the same time included static code analysis and test execution. Being an open-source platform with a public repository available on GitHub, WatchDog also utilizes Continuous Integration (CI) practices, such as code merging through pull-requests, code reviews and automated builds with test executions on external CI engine, Travis CI.

On the other hand, the first version of IntelliJ plugin lacked many of these features, so the refactoring process was also a good opportunity to fill this void. While the unit tests, still placed within the Eclipse plugin, were easily adapted to cover Core module as well, configuring Maven to automatically build the whole project and output the production-ready versions of both plugins was not a straightforward task.

The first step in configuring our build process was to define a set of *pom.xml* files (Project Object Model, containing the configuration for a project and its dependencies). The initial version of Eclipse plugin consisted of one top-level POM file which had four modules – WatchDog implementation, unit tests and two Eclipse-required components, *Feature*

¹⁴<https://github.com/TestRoots/watchdog/issues/194>

3. BUILDING A RESEARCH SOFTWARE TOOL: WATCHDOG IMPLEMENTATION PROCESS

and *p2updatesite*. These two components are specific to plugin development and they provide some high-level functionalities, such as grouping of plugins and creating a deployable version for an update site. Furthermore, at run-time Eclipse uses OSGi standard, a set of specifications that defines a component and service model for Java, which means that every plugin for Eclipse should be an *OSGi Bundle* [28]. A bundle, which is the smallest unit of modularization in OSGi, can be dynamically managed in the run-time, which means that OSGi will automatically load other required dependencies that come in a form of bundles. This aspect led us to the logical conclusion to define our WatchDog Core module as an OSGi bundle as well. While this approach proved useful for Eclipse, we could not benefit from OSGi when it comes to IntelliJ, because plugins for this IDE may not be defined as OSGi bundles, so we had to use Maven for dependency resolution.

Another important aspect to consider was an IDE dependency – for the building process, it was necessary to provide source codes of both IDEs. While Tycho¹⁵, Eclipse’s Maven tool for building plugins and OSGi bundles, automatically provides sources of all necessary Eclipse versions, getting the IntelliJ source code has to be done manually. This task consists of downloading the latest IntelliJ Community Edition source code, building it and installing it to the local Maven repository.

The result of the full Maven build are production-ready IDE plugins, both packed independently – the Eclipse plugin in a form of an update site from which the plugin can be downloaded and installed, and the IntelliJ plugin as a ZIP archive, which can be distributed via the JetBrains plugin repository. The outputs are the same as if they were generated manually, directly from the IDEs. Furthermore, we also digitally sign our Eclipse plugin JAR files before each release¹⁶, in order to verify the integrity of the distributable plugin version.

Finally, we provide appropriate shell scripts which are used to automate the whole process on our Travis CI build server. We use Travis CI to inspect every pull request to our production repository on GitHub and passing all checks successfully is the first necessary condition for accepting them.

3.4 Building multi-platform tool: Summary

In this section, we classify and summarize the technical and organizational challenges that the creation of a multi-platform architecture poses, by the example of the development of the new WatchDog architecture for IntelliJ and Eclipse, which are the first objective of this thesis (*G1*). Then we share our experiences and solutions on how we solved these problems, which were also published in a workshop article [5].

3.4.1 Challenges

Below, we outline the technical and organizational challenges that we experienced while creating a family of IDE plugins.

¹⁵<https://eclipse.org/tycho/>

¹⁶<https://github.com/TestRoots/watchdog/issues/185>

The Plugins Must Be Easy to Maintain (C#1). If plugins are independent forks, every change needs to be ported. Inconsistently changed clones are one of the biggest threats to the development of multiple plugins [15].

The Host IDEs Differ Conceptually (C#2). While IDEs share many design commonalities, such as the editor model in which developers read and modify code, they also feature profound differences. As one example, IntelliJ does not have a workspace concept, based on which the Eclipse user could en- or disable WatchDog, while on the other hand, IntelliJ offers a lot more user actions which can be observed by a plugin, although these actions can be invoked in many different ways.

The Host IDEs Differ Technically (C#3). In practice, technical differences between IDEs and their tooling might be more problematic than conceptual ones. As an example, Eclipse employs the open OSGi framework for plugin loading and dependency management and the Maven Tycho plugin for building. For rendering its user interface, it uses SWT. By contrast, IntelliJ has a home-grown plugin and build system, and is Swing-based.

The Data Format Evolves (C#4). As researchers, we are eager to receive the first data points as early as possible. However, especially in the early stages of plugin development, changes to the data format are frequent and unforeseeable. Moreover, data structure from different plugins might deviate slightly, for example, because Eclipse requires additional fields for its perspectives.

The Project Has Few (Development) Resources (C#5). For example, we developed WatchDog with less than one full-time developer at any given time point.

3.4.2 Guidelines

Following up on the previous challenges, we now give concrete guidelines based on our own experiences with designing a multi-platform architecture for WatchDog. The guidelines link to concrete solutions in WatchDog that tool smiths can re-use in their own projects.

Assess the Expected Degree of Commonality (GL#0). Before starting the plugin design or refactoring, tool creators should assess the amount of features (and, thus, code) that can be shared between two different IDEs.

Create a mutually shared core (GL#1). Figure 3.3 introduces WatchDog’s 3-layer architecture. In its client layer, both plugins extend one common core. This alleviates the maintenance difficulties of two forks. Functionality that cannot be shared resides in WatchDog for Eclipse (right-hand side) and WatchDog for IntelliJ (left-hand side).

We strongly recommend to set up a dynamic project dependency to the core in each of the IDEs. A traditional approach would be to develop the core as a plain-old jar library. This scales well when we expect only changes from the library (core) to the plugins (clients), and not vice versa. However, since we expect frequent changes to the core from within the plugins, we need a dynamic solution; the source code of the core should be available as a shared component in the development space of both IDEs. This way, changes to the core from within one IDE are automatically pulled into the development project in the other. Plugin creators can import our Eclipse¹⁷ and IntelliJ¹⁸ project directories to have a starting

¹⁷<https://goo.gl/tC0iiV>

¹⁸<https://goo.gl/2CgHsN>

3. BUILDING A RESEARCH SOFTWARE TOOL: WATCHDOG IMPLEMENTATION PROCESS

example of how to setup such a development environment with minimal overhead.

Find Functional Equivalents (GL#2). A feature in one IDE can 1) exist equivalently in the other, e.g. the similar code editors in Eclipse and IntelliJ, 2) approximate another, e.g. Eclipse’s workspace and IntelliJ’s Project model, or 3) be missing. For example, Eclipse’s concept of perspectives is not present in IntelliJ. In the last case, one can also try to add the functionality over the plugin.

Abstract over Technical Differences (GL#3.1). Technical differences can limit the amount of commonality in the core. Ideally, we want the plugins to be as shallow as possible and the core to contain all logic. One problematic example is that IntelliJ’s home-grown plugin class loader is incompatible with reflection-enabled libraries such as MapDB, which we use as our cache. Caching should not differ per plugin, thus it is implemented in the core (see GL#1). As a result, we needed to prefix every cache method in IntelliJ to replace its current class loader with our own implementation, and switch back afterwards. We abstracted over this IntelliJ-specific technicality through the template method pattern.

Regarding how to build a deployable plugin, we were able to design a Maven module that automatically builds our IntelliJ plugin. While the Maven models for IntelliJ and Eclipse are very different internally, they can be built with the same Maven command from the outside. Our Maven POM files¹⁹ can serve as examples of how to create such a uniform project build setup.

¹⁹<https://goo.gl/Ajc9oJ>, <https://goo.gl/sE6E17>

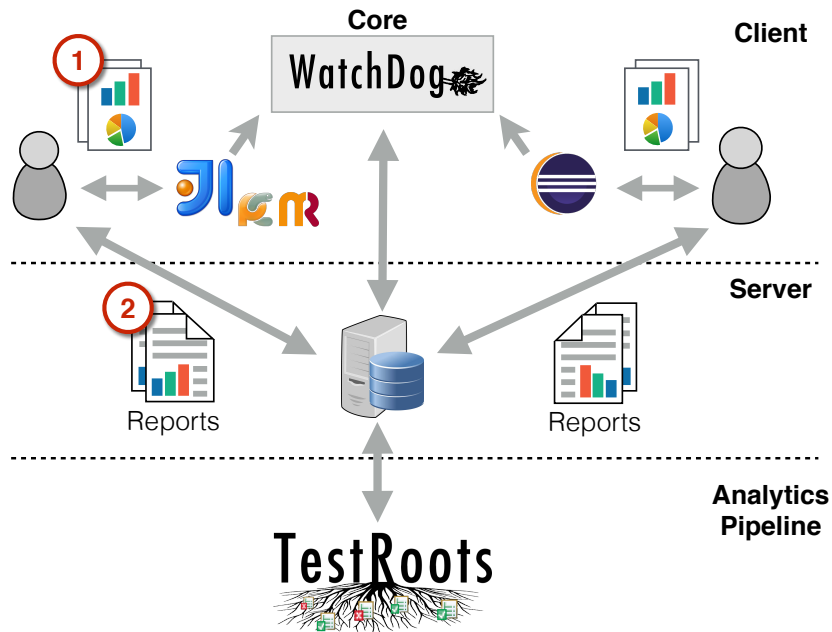


Figure 3.3: WatchDog’s three layer architecture (source: [5]).

Refactor away from the client (GL#3.2). On a higher level, we pushed the main part of our analytics in figure 3.3 to the back-end layer, which is agnostic of the clients and connects directly to the server: It starts with filtering data (for example, from pre-release versions or students), and then branches off to 1) generate computation-intensive nightly reports for our users, which are accessible over the server layer, and 2) extract data for our own research purposes. The server and analytics layer would not need to change if we added support for, e.g., Visual Studio or Netbeans.

Use a schema-free database (GL#4). Our Mongo database's NoSQL format allowed for rapid prototyping and different data format versions without migration issues. Our performance tests show that the WatchDog server, run on a four CPU 16-Gigabyte desktop machine, can handle a load of more than 50,000 users simultaneously.

When analyzing data from a schema-free database, in practice, a base format is required. As an example, WatchDog was not originally intended as a multi-platform architecture. Thanks to MongoDB, we could introduce the “IDE” field to differentiate between the plugins later. We did not need to perform a data scheme translation, and, most importantly, we can still accept the previous versions of the data format.

Use Automation & Existing Solutions (GL#5) Instead of relying on a heavy-weight application server, we use the free Pingdom service²⁰ to monitor the health of our services and an adapted supervise script²¹ to restart it in case of crashes. To minimize server costs, a backup of the database is time-synchronized on the cloud via the standard Unix-tools rsync and rdiffbackup.

We heavily rely on the use of Continuous Integration (Travis CI) to execute our static and dynamic analyses. No release may be published without all indicators being green. Our static analysis include manual code reviews in pull requests,²² but are mainly automated: FindBugs, PMD, Teamscale, and CoverityScan in the client layer and CodeClimate for defect and test coverage control in the server layer. We found these tools easy to setup and free to use.

The whole WatchDog project (all layers in figure 3.3) accounted for little more than 20,000 Lines of Code (LoC) on November 19th, 2015. We were able to achieve this level of conciseness by following GL#2 and GL#3.2. and building our technology stack on existing solutions. As an example, the WatchDog server (layer 2 in Chapter 3.3) is a minimalist Ruby application (200 LoC) that uses Sinatra for its REST API and unicorn to enable the parallelism.

²⁰<https://www.pingdom.com/>

²¹<http://cr.yp.to/daemontools.html>

²²E.g., <https://github.com/TestRoots/watchdog/pull/150>

Chapter 4

Study Design

In the previous chapter, we reflected on our experiences and practices in the development process of building a research tool, which we explain in detail in this chapter. We first give its overview from the user’s perspective, and then explain the logic behind it. Finally, we explore various aspects of our study design and describe the experiment process in details.

4.1 Practitioner’s perspective

In order to observe developers’ behavior and study our research questions we use the previously described tool to automatically collect the real-world data. WatchDog’s design and methodology can be explained by the following use case scenario:

Jenny is an open-source developer who wants to monitor how much she is testing during her daily development activities inside her IDE. Since Jenny uses Eclipse, she installs the WatchDog plugin from the Eclipse Marketplace. Once WatchDog is installed, a dialog guides Jenny through the registration process which has been mentioned in the previous chapter. During this process, we assign one user ID and at least one (or possibly more) project ID, in order to identify and group the collected data. Furthermore, we ask Jenny several questions regarding her development habits, although we have recently offered an option to perform an anonymous registration¹, during which we do not collect the survey data, but still assign user and project IDs. Afterwards Jenny continues to work on her project using Eclipse as usual while WatchDog is silently recording her testing behavior in the background.

When Jenny starts Eclipse, WatchDog creates three intervals: `EclipseOpen`, `Perspective`, and `UserActive` (1). She plans to refactor the production code, so she executes the unit tests to ensure the desired behavior of that code, triggering the creation of a `JUnit-Execution` interval, enriched with the test result “Passed” (2) and test execution duration. Having browsed the source code of the file (3) to understand which parts need to change (a `Reading` interval is triggered), Jenny then performs the necessary changes (recorded by a `Writing` interval). A re-execution of the unit tests shows Jenny it fails after her modification

¹<https://github.com/TestRoots/watchdog/issues/209>

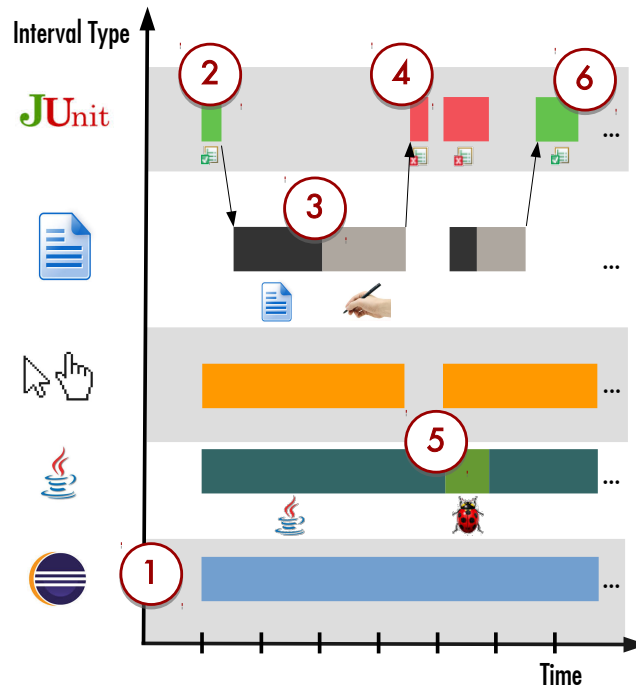


Figure 4.1: Exemplary workflow visualization with intervals (source: [3]).

(4). She steps through the test with the debugger (5) and fixes the error. The final re-execution of the test (6) succeeds.

After this development task, Jenny wants to know how much of her effort was devoted to testing, and whether she followed the test-driven development (TDD) practice. She can retrieve two types of analytics: the *immediate statistics* inside the IDE (marker 1 in figure 4.2), and her personal *project report* (2). Then, she opens the statistics view and selects 10 minutes as the time window to monitor. WatchDog will automatically analyze the recorded data and generate the view depicted in Figure 4.2. The *immediate statistics* view provides information about production and test code activities within the selected time frame. Sub-graph 1 in Figure 4.2 shows Jenny that she spent more time (over one minute) reading than writing (only a few seconds). Moreover, of the two tests she executed (marker 2), one was successful and one failed. Their average execution time was only 1.5 seconds. Finally, Jenny observes that the majority (55%) of her development time has been devoted to engineering tests (3), not unusual for TDD [3].

While the immediate statistics view provides an overview of recent activities inside the IDE, the *Project Report* can be used to analyze global, and more computationally expensive statistics for a given project throughout the whole project history. Jenny accesses her report through a convenient link in the IDE, or through the TestRoots website², entering the project's ID. Jenny's online project report summarizes her development behavior in the IDE over the whole recorded project lifetime. By analyzing this *general* report, Jenny observes that she spent over 195 hours of working time in total for the project under analysis, corre-

²<http://testroots.org/report.html>

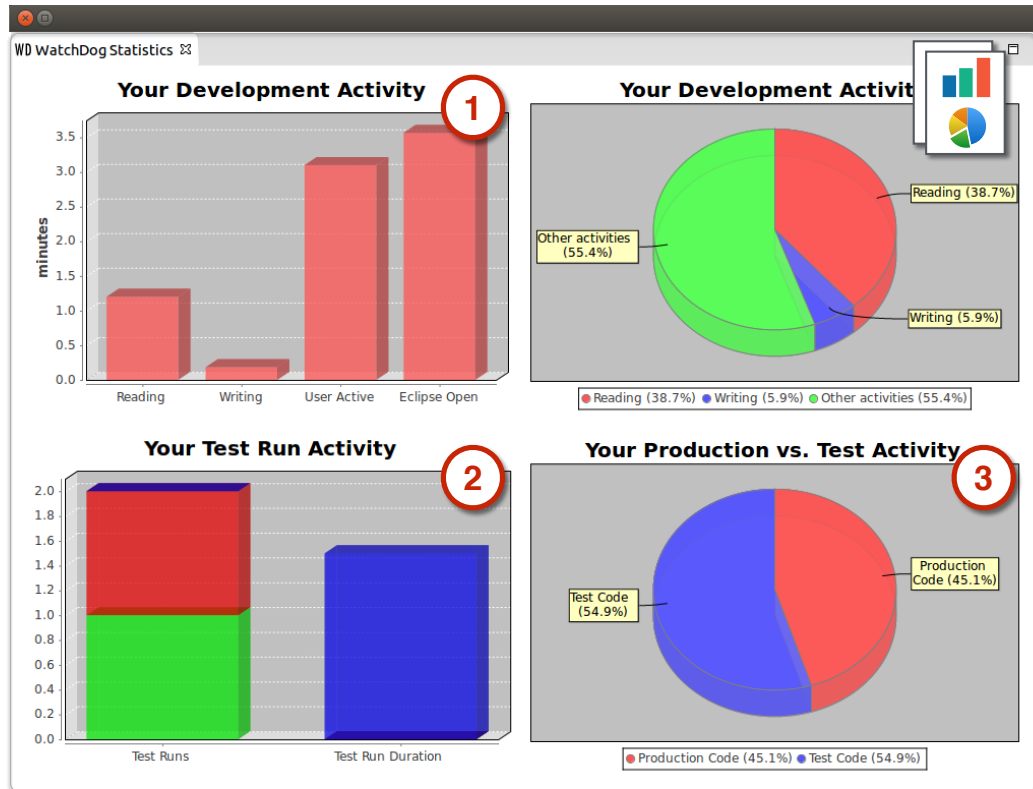


Figure 4.2: WatchDog's immediate statistics view in the IDE (source: [27]).

sponding to 36 minutes per day on average (marker 1 in the Figure 4.3). She was actively working with Eclipse in 58% of the time the IDE was actually opened. The time spent on writing Java code corresponds on average to 55% of the total time. She spent the remaining 45% reading Java code. When registering the project, Jenny estimated the working time she would spend on testing to equal 50%. Using the generated report, she figures out that her initial estimation was quite precise since she actually spent 44% of her time working on test code.

*Project Report*³ also provides the *TDD statistics* for the project under analysis (marker 2 in the Figure 4.3). Moreover, anonymized and averaged statistics from the large WatchDog user base allow Jenny to put her development practices into perspective, comparing them to those of other developers. This way, project reports foster comparison and learning among developers. In TDD, programmers systematically co-evolve production and test code, while constantly cycling between the states of succeeding and failing test cases. To measure to what extent developers follow it, we use an approach based on textual matching with regular expressions: in a nutshell, the analytics pipeline chronologically orders a stream of IDE activities. Then, it matches regular expressions modeling TDD against this stream. The portion of the matches in the whole sequence gives a precise indication to which extent a developer applied TDD. This method [3] has been already used to answer “how common

³Example report: <http://goo.gl/k9KzYj>

Detailed Statistics

In the following table, you can find more detailed statistics on your project.

1

Description	Your value	Mean
Total time in which WatchDog was active	195.8h	79h
Time averaged per day	0.6h / day	4.9h / day
General Development Behavior	Your value	Mean
Active Eclipse Usage (of the time Eclipse was open)	58%	40%
Time spent Writing	13%	30%
Time spent Reading	11%	32%
Java Development Behaviour	Your value	Mean
Time spent writing Java code	55%	49%
Time spent reading Java code	45%	49%
Time spent in debug mode	0% (0h)	2h
Testing Behaviour	Your value	Mean
Estimated Time Working on Tests	50%	67%
Actual time working on testing	44%	10%
Estimated Time Working on Production	50%	32%
Actual time spent on production code	56%	88%
Test Execution Behaviour	Your value	Mean
Number of test executions	900	25
Number of test executions per day	3/day	1.58/day
Number of failing tests	370 (41%)	14.29 (57%)
Average test run duration	0.09 sec	3.12 sec



Summary of your Test-Driven Development Practices

You followed Test-Driven Development (TDD) 38.55% of your development changes (so, in words, quite often). With this TDD followship, your project is in the top 2 (0.1%) of all WatchDog projects. Your TDD cycle is made up of 64.34% refactoring and 35.66% testing phase.

2

Figure 4.3: WatchDog's project report (source: [5]).

is TDD in practice?" The new feature, embedded in project reports, enables all WatchDog users to individually examine their own testing style and conformance with TDD.

Migration to another IDE. Jenny wants to migrate her project developed using Eclipse to IntelliJ without losing the testing statistics already collected by WatchDog. Since WatchDog is a multi-IDE solution, Jenny can easily migrate by installing the WatchDog plugin for IntelliJ available from the JetBrains Plugin repository. Jenny selects the alternative registration procedure available for already registered users. Using her personal user and project ID after migration, she can continue collecting data on the same project and the project report would provide joint results from the both IDEs.

4.2 Backend logic and metrics

While developers use the IDE for their regular, everyday activities and only retrieve their reports on demand, WatchDog constantly records their development behavior and periodically transfers collected data to WatchDog Server. On the lowest level of the WatchDog's back-end logic, we have various event listeners, instantiated on every start-up of the IDE, which observe the user's actions. Every single action (e.g. scroll, mouse click, typing text, etc.) is registered as a WatchDog Event, and propagated further to our *EventManager* which creates appropriate WatchDog Intervals. These intervals consist of one or many subsequent observed events, grouped together by our *IntervalManager*. The overview of the intervals is given in Table 4.1 and their UML diagram in the Figure 4.4. We should note that the bright yellow color is used for the classes which belong to the WatchDog Core, while the grey color represents classes from the specific plugin, particularly WatchDog for IntelliJ.

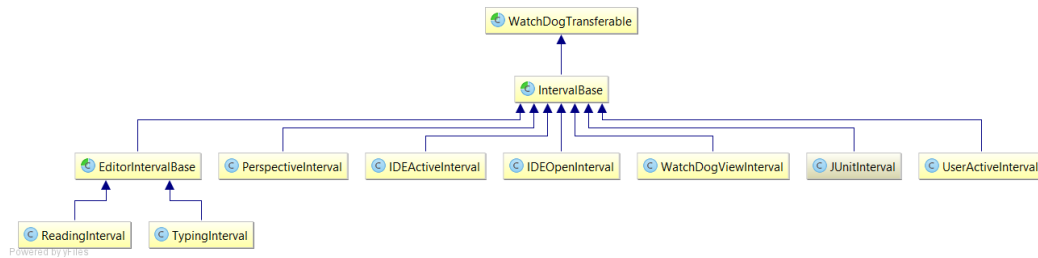


Figure 4.4: WatchDog's intervals.

Intervals concerning the user's activity (Reading, Typing, and other general activity) are backed by an inactivity timeout, so that we only record them when the user is actively working in the IDE.

However, if we detect that the IDE lost the focus (end of IDEActive interval), or the user switched from writing file X (Typing) to reading file Y (Reading), we immediately end the currently opened interval. Intervals may overlap. For example, Typing or Reading intervals are wrapped inside a user activity (which is again wrapped within the IDEActive, Perspective and IDEOpen interval). However Reading and Typing intervals are by nature mutually exclusive. We refer to an IDE session as the time span in which the IDE was open and not closed or interrupted, for example because the developer suspended the computer. All intervals that belong to one IDE session are hence wrapped within one IDEOpen interval, as is shown in Figure 4.1 (1).

Depending on the type of the interval, we enrich it with different numerical and categorical information: in the Reading or Typing interval, we store whether the underlying file is a Java class, a hash of its file name, its length in source lines of code without whitespaces (SLOC), and whether it is production or test code. Additionally, for Typing intervals, we calculate the Levenstein edit distance [18] between the content of the file before and after the modification in the interval. This gives us an indication of the size of the changes made in the Typing interval. If it is a Java class, we rate the file that the developer accesses in a Reading or Typing interval as either production or test code. We classify any other file type, for example an XML configuration file, as unknown.

4. STUDY DESIGN

Table 4.1: Overview of WatchDog intervals (based on:[3]).

Interval Type	Description
JUnitExecution	Interval creation invoked through the IDE-integrated JUnit runner (also working for Maven projects). Each test execution is enriched with the SHA-1 hash of its test name (making a link to a Reading or Typing interval possible), test result, test duration and child tests executed.
Reading	Interval in which the user was reading in the IDE. Backed by inactivity timeout. Enriched with an abstract representation of the read file, containing the SHA-1 hash of its file name, its SLOC, and an assessment whether it is production code, or test code. A test can further be categorized into a test (1) which uses JUnit and is therefore executable in the IDE; (2) which employs a testing framework; (3) which contains “test” in its filename; (4) or contains “test” in the project file path (case-insensitive). Backed by inactivity timeout.
Reading	Interval in which the user was reading in the IDE. Backed by inactivity timeout. Enriched with an abstract representation of the read file, containing the SHA-1 hash of its file name, its SLOC, and an assessment whether it is production code, or test code. A test can further be categorized into a test (1) which uses JUnit and is therefore executable in the IDE; (2) which employs a testing framework; (3) which contains “test” in its filename; (4) or contains “test” in the project file path (case-insensitive). Backed by inactivity timeout.
Typing	Interval in which the user was typing in the IDE. Backed by inactivity timeout.
UserActive	Interval in which the user was actively working in the IDE (evidenced for example by keyboard or mouse events). Backed by inactivity timeout.
IDEActive*	Interval in which the IDE had the focus on the computer. *) Not shown in figure 4.1.
Perspective	Interval describing which perspective the IDE was in (Debugging, regular Java development, ...). Specific only to Eclipse, since IntelliJ does not support the concept of perspectives.
IDEOpen	Interval in which the IDE was open. If the computer is suspended, the IDEOpen is closed and the current session ends. Upon resuming, a new IDEOpen interval is started, discarding the time in which the computer was sleeping. Each session has a random, unique identifier.

We have different recognition strategies to recognize test classes (see table 4.1): in order to designate the file as a test that can be executed inside the IDE, we require the presence of at least one `JUnit import` together with at least one method that has the `@Test` annotation or that follows the `testMethod` naming convention. This way, we support both JUnit3 and JUnit4. Furthermore, we recognize imports of common Java test frameworks and their annotations (Mockito, PowerMock). As a last resort, we recognize when a file contains “Test” in its file name or the project file path. It seems a common convention to pre- or postfix the names of test files with `Test` [31], or to place all test code in one sub-folder. For example, the standard Maven directory layout mandates that tests be placed under `src/test/java`.⁴ Thereby, we can identify and differentiate between all tests that employ standard Java testing frameworks as test runners for their unit, integration, or system tests, test-related utility classes, and even tests that are not executable. We consider any Java class that is not a test according to this broad test recognition strategy to be production code.

To further explain the logic and implementation, we can relate to Figure 4.5. Here, *InitializationManager* (1) creates IDE listeners (2), which use *EventManager* (3) to register the observed WatchDog events (4) of various types (5). *EventManager* (3) notifies *IntervalManager* (6) which groups the events into intervals, as we have already explained. All intervals are stored locally for a period of up to ten hours (by default), or longer, if necessary, until they are transferred to our server (e.g. when a user is connected to the inter-

⁴<http://maven.apache.org/guides/getting-started>

net again). For storing them, our *IntervalPersister* (7) utilizes the MapDB library ⁵, which caches intervals in memory, but also stores them on disk and ensures their persistence across sessions. Stored intervals are used for displaying real-time statistics inside the IDE, by using *IntervalStatistics* (8) which further makes use of JFreeChart ⁶ library for graph drawing.

Finally, *IntervalTransferManager* (9) is responsible for the repeated transferral of all closed intervals to the server by using REST API. While we try to transfer them every three minutes, we also handle edge cases such as a transfer on exiting of the IDE and back off when user is offline. On successful transfer, we remove intervals from MapDB instance, although they are still available on disk, when we use them for displaying the in-IDE statistics overview.

4.2.1 Data collection and analysis pipeline

All transferred intervals, upon sanity checking, are stored in a no-SQL database MongoDB on *WatchDog Server*. This set-up allows for scaling up to thousands of simultaneous connections, but also further changes in data format which would ease backward compatibility. All data on the server are processed in analytics pipeline – a set of R scripts made for data analysis which generate project reports on a daily basis. Beside these scripts, another batch of R scripts was made in order to investigate testing habits of Eclipse users [3]. Since we made our IntelliJ plugin fully compatible with the existing infrastructure, we are able to reuse these scripts in the research of IntelliJ users, which we present in the following chapter.

4.3 Study participants

While WatchDog Eclipse plugin has been advertised in various ways [3], we have also tried to acquire IntelliJ users by:

- Updating our project website.⁷
- Enhancing our project reports which are generated daily.
- Utilizing social media channels to promote it.
- Contacting open-source developers who previously expressed the interest in WatchDog, but were not using Eclipse.
- Publishing the plugin on the official JetBrains plugin repository.⁸

During the observation period of five months, we had 33 active users from 15 different countries who transmitted their data to our server, nearly half of them being from the United States and China (eight from each country). Although the number of WatchDog IntelliJ users is significantly smaller than the number of Eclipse users who participated in the previous

⁵<http://www.mapdb.org/>

⁶<http://www.jfree.org/jfreechart/>

⁷<http://www.testroots.org>

⁸<https://plugins.jetbrains.com/plugin/7828>

4. STUDY DESIGN

study (416 users), we still gathered enough data to perform the initial analysis, answer the research questions and compare the results.

Similar to Eclipse users, our IntelliJ users (figure 4.6) mostly use some version of Windows OS (57%), followed by Mac OS (37%), while only two of them are Linux users (6%). Their programming experience is normally distributed, mainly between three and six years (28%), while ranges of 1-2 years and 7-10 years both amount for 21% each. These 33 participants registered 48 unique projects and, in total, we observed 847 hours of working time in which IntelliJ IDE was open, during which we observed 1,334 distinct IDE sessions.

Figure 4.5: WatchDog's UML diagram.

4. STUDY DESIGN



Figure 4.6: Operating system developers use and their programming experience.

Chapter 5

Analysis And Discussion Of The Research Questions

The WatchDog for IntelliJ has been added to the ongoing study which assess software testing and which already produced results with Eclipse [3, 4]. Since its release on the JetBrains plugin repository, it has been used to collect the real-world data and perform analysis on it. In this thesis we address a period of five months, from the initial release in the beginning of September 2015, until the end of January 2016, when the further development effort started, going beyond the scope of this work.

The goal of this experiment (*G2*) is to perform a *replication study* of the work that was presented in the previous work of TestRoots [3] (referred to as the *original study* in the remainder of this chapter) and which addressed developers who use Eclipse. While using the same approach and methodology from the original study, we analyze the testing behavior of IntelliJ developers in this work and, furthermore, we compare the results from both studies and detail our conclusions with respect to the defined research questions.

5.1 Analysis of the research questions

In this section we detail our results per each research question and make a parallel to the results of the original study.

5.1.1 RQ1: When and why do developers test?

As in the original study, we analyze the first research question by assessing the following sub-questions:

RQ1.1 How common is testing?

By applying the broadest recognition of test classes as described in Chapter 4 (detecting test classes based on the name of files and importing statements), 27 of the 48 analyzed projects contain tests that a user either read or modified. Hence, for 44% of projects, we could not detect work on tests in the IDE, neither to execute, nor to read or modify them, which does not significantly deviates from 57% in Eclipse.

5. ANALYSIS AND DISCUSSION OF THE RESEARCH QUESTIONS

By narrowing down the recognition of tests to JUnit tests only, which can be executed inside the IDE, we find that 25 projects have such tests. This is in accordance with the results obtained in the registration survey, since 27 developers claimed to have been using JUnit. Although this implies that our method of identifying testing frameworks performs mostly correctly, in the original study it emerged a discovery that for only 47% of projects which claimed to have JUnit tests in the survey it was possible to detect them in the processed data. On the other hand, we detected tests in 73% of the IntelliJ projects which allegedly have them (note that we detected JUnit tests in some projects for which developers did not claim or did not know they have them).

Our second sub-research question is:

RQ1.2 How frequently are tests executed?

From 25 projects, we observed test executions in the IDE in 17 projects (68%), which is slightly smaller than in Eclipse (85%). Developers of these 25 projects contributed 1,184 sessions and ran 990 test executions. Compared to the original study, which had 3,424 sessions and 10,840 test runs, we have the first sign that the tests are performed less frequently in IntelliJ than in Eclipse, or, at least, in smaller amount.

Moreover, out of 1,184 sessions, in 134 of them (11.3%) at least one test was run, similarly to 15% (527/3,424) in Eclipse. This led to the significantly small average number of the executed tests per any session (0.82), but also to the small average number of tests per sessions in which at least one test was run (7.33). These averages are in accordance with the original study (3.2 and 20.7 respectively), coming as a consequence of the smaller number of single test executions.

RQ1.3 and following should give an indication as to why and when developers test, based on their behavior. It is expected that when developers change the testing code, they are likely to execute those tests more to inform themselves about the current execution status of the test they are working on. This is examined in the following sub-questions:

RQ1.3 Do developers test their test code changes?

The correlation between test code churn and the number of test runs yields a weak *Spearman rank-order (ρ) correlation coefficient*, $\rho = 0.41$ in our dataset. While there is an obvious relationship between the two variables, the correlation does not imply a causation or a direction. Therefore, we cannot say with confidence that developers executed more tests *because* they changed more test code, although this might be one of the possibilities.

A logical next step is to assess whether the same holds for modifications to production code: Do developers assert that their production code still passes the tests?

RQ1.4 Do developers test their production code changes?




This correlation is significantly weaker, in fact negligible, with $\rho = 0.27$, which means that we could hardly detect test executions after production code refactoring. This is in accordance with Eclipse, where the correlation was just a little bit higher, but still weak ($\rho = 0.38$).

Finally, we examine in how many cases do developers modify their tests when they change their production code (or vice versa):

RQ1.5 Do developers co-evolve test and production code?

A weak $\rho = 0.44$ suggests that tests and production code have some tendency to change simultaneously, but it is certainly not the case that developers modify their tests for every

Table 5.1: Descriptive statistics for important variables. Histograms are in log scale.

Variable	Unit	Min	25%	Median	Mean	75%	Max	Histogram
JUnitExecution duration	Sec	0.002	0.54	1.37	51.66	4.23	20,610	
Tests per JUnitExecution	Items	1	1	1	1.32	1	142	
Time to fix failing test	Min	0.22	0.87	4.78	15.17	17.22	190.9	

production code change, and vice versa. The similar correlation, although a bit weaker ($p = 0.35$), was observed in Eclipse, so we could confirm that production code and tests do not necessarily evolve together.

5.1.2 RQ2: How and why do developers run tests?

Software developers usually waste a lot of time for the maintenance processes, such as the compilation of large projects, building and re-building solutions and deployments. Because of the potential duration of these tasks, they are often shifted to some continuous integration service, which will perform full project build in the background and report results, while developers will execute them irregularly. Executing tests can also be observed as one of those long-lasting tasks, which is usually the part of the build process. To be able to explain how and why developers execute tests inside the IDE, we must therefore first know how long developers have to wait before they see a test run finish:

RQ2.1 How long does a test run take?

For a 50% of all test executions it takes one second and a half to finish, which is almost three times longer than in Eclipse. On the other hand, in both IDEs, test executions generally do not take long to complete – 75% are finished within five seconds (see Table 5.1). Interesting observation is that the percentage of executions which last longer than one minute and two minutes are almost the same in both IDEs (7.4% and 4.8% of runs respectively).

Having observed that most test runs are short, our next step is to examine whether short tests facilitate testing:

RQ2.2 Do quick tests lead to more test executions?

In the original study the hypothesis that the short test executions could potentially lead to a higher number of runs per session was not proven correct. This came as a conclusion from the lack of correlation between the two distributions, which we further confirm in IntelliJ. Moreover, while for the proof of the hypothesis we would expect a negative correlation between the test duration and the number of test executions, in IntelliJ we detect a weak positive correlation ($p = 0.42$) between those values, which could imply the opposite – short tests are executed less frequently.

While the original study addressed the issue of test selection, the small data-set in this work and a small number of executed tests in general are limiting our analysis, so we are not able to address this matter in the adequate way.

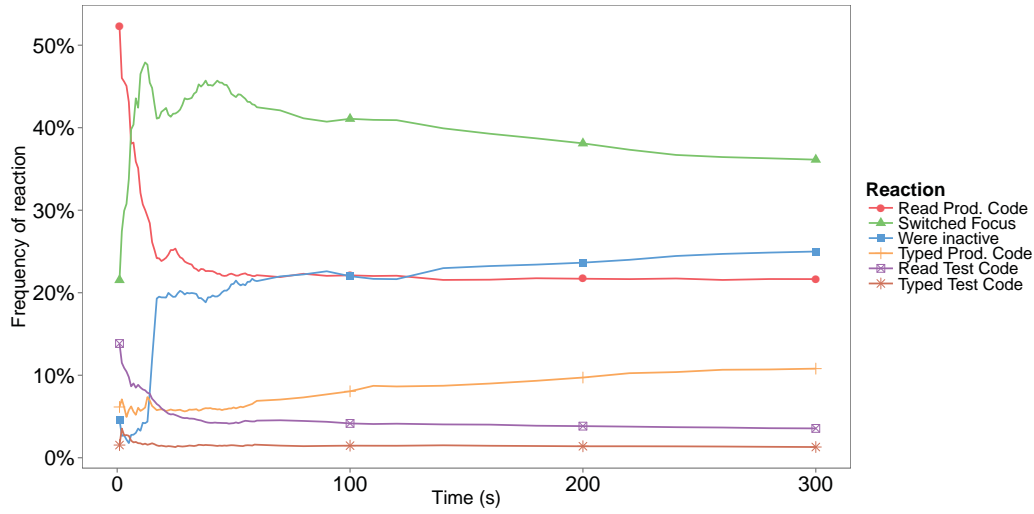


Figure 5.1: The immediate reactions to a failing test.

5.1.3 RQ3: How do developers react to test runs?

While we have examined the test execution behavior in the previous research questions, we will now assess consequences of those executions:

RQ3.1 How frequently do tests pass and fail?

Following the fail-fast strategy, everything but the successful completion of a test execution is considered a failure. These failures can come as a consequence of three possible scenarios: compilation errors, an uncaught and an unexpected run-time exception is thrown during the test case execution, or a test assertion is not met.

An interesting finding is that in our replication study we observed almost identical percentage – 65.35% (647 out of 990) of failed executions – as in the original study (65%, 7,047 out of 10,840). Even with the smaller data-set, we are able to confirm that the test runs fail almost two times more frequently than they succeed.

As test failures are apparently a situation developers are often facing, we ask:

RQ3.2 How do developers react to a failing test?

Following the original study approach, for each failing test case we create a sequence of consecutive intervals (ordered by their start time) and we analyze the following developers' reactions for a period of up to five minutes (300 seconds), as shown in Figure 5.1.

The common behavior of both Eclipse and IntelliJ users is that they read the production code within the first few seconds following the failing test (over 60% in Eclipse and over 50% in IntelliJ). While in the original work the second most common reaction is to read the test code first with 17%, we found that IntelliJ developers shortly after failure switch the focus of their IDE windows (in more than 20% cases), whereas they read the test code in slightly less than 15%.

Similarly to the Eclipse users, switching focus away from the IDE after five seconds becomes the primary reaction, which coincides with users becoming inactive inside their

IDE, while reading the production code curve decreases to the almost half of its initial value. During the first minute, we notice that the activity on the test code slowly decreases, while users mostly remain or become inactive, or deal with the production code. Likewise in Eclipse, after two minutes all reactions asymptotically reach their overall distribution, with a little variability.

In the RQ3.2 we saw the reactions to a failing test, while in RQ3.3 we examine the consequences of these reactions, namely:

RQ3.3 How long does it take to fix a failing test?

Among 647 failed executions we detected 145 unique test cases (according to their file name hash). For these 145 tests we observed at least one successful execution for 92 (63%), which means that 37% of tests were never fixed during our observation period, which is slightly more than 30% of unfixed tests in Eclipse.

For the 92 failing tests that we know have been fixed later, we examine how long did it take to fix them and get the following interesting result – while in Eclipse 50% of test repairs happen within 10 minutes, in IntelliJ 50% fixes are applied in under half of that time. IntelliJ developers appear to be faster at fixing tests, with 75% of test fixed within 17 minutes (see Table 5.1).

5.1.4 RQ4: Do developers follow TDD?

Test-Driven development (TDD) is a software development process originally proposed by Beck [1], which assumes that developers first write test cases based on requirements, followed by the production code which should pass these tests. Original study introduced two ways of recognizing TDD patterns – *strict TDD* and *lenient TDD*. The strict TDD pattern follows the TDD life cycle by definition, starting with the successful test execution (to ensure initial state of the code) and followed by test edits with failing executions, after which developers edit production code until the test execution succeeds. In practice it is usually very difficult to follow strict TDD for various reasons (lack of just-in-time compilation, co-evolution of production and test code). Because of this, a new pattern, lenient TDD, assumes that developers may immediately start with editing test code and allows for modifications of production code along with test code. Finally, while these two patterns reflect a process of developing a new functionality, refactoring the existing code can also follow the TDD practice, ensuring that the tests are successfully executed before and after code modifications. RQ4 examines the adoption of TDD in practice, by identifying interval sequences which match the defined TDD patterns.

While in Eclipse there were 10 developers (2% of all developers in the study, or 15% of developers who executed tests) who followed strict TDD, in IntelliJ we could not detect any developer who followed this pattern, which can potentially be caused by the size of our data-set. On the other hand, we successfully detected 4 users who, at some level, followed lenient TDD pattern (12% of all developers in the study, 31% of developers who executed tests). Only one among them followed lenient TDD in 19% of his time, while others did that for less than 5% of their intervals. We should note that 44.7% of lenient TDD intervals comes from the code refactorings, which is also the pattern that one user made the highest compliance with, following it with 97%.

In the original study, there were generally more developers who followed lenient TDD (33, 8% of all developers, or 49% of developers who executed tests), with 20% of them following this pattern in more than 10% of their intervals. It is a common finding that only more experienced users are following any kind of TDD – 67% of Eclipse developers have experience of >7 years, which is also the case for all 4 IntelliJ developers who followed lenient TDD (2 of them having more than 10 years of experience).

5.1.5 RQ5: How much do developers test?

Previous research questions were mostly based on test executions, while in this question we examine the developers' behavior during the code development phase. To answer this question, we follow the same approach as in the original study: we consider *Reading* and *Typing* intervals, and further split the two intervals according to the identified type of the document a developer works on: either a production or a test class. We can safely neglect test executions in this part, since we already saw that their duration is insignificant, especially when compared to the reading and writing code, and since developers can also work in the IDE at the same time. When registering new projects, developers had a chance to estimate the time they spend on testing in the project. While in the original study all participants had to fill the registration survey in order to use WatchDog, we have recently enabled anonymous registrations which allowed users to use our plugin without stepping through the whole process. Even though this approach could facilitate user acquisition, a known tradeoff we had to accept was to have less survey data, which we will use to compare to developers' actual testing behavior.

In Figure 5.2 we can see the difference of actual time invested in production compared to developers' estimations per each project. Likewise in the original study, a value of 0 means the estimation was accurate, and a value of 100 denotes that the programmer expected to only work on tests, but in reality only worked on production code (the opposite stands for

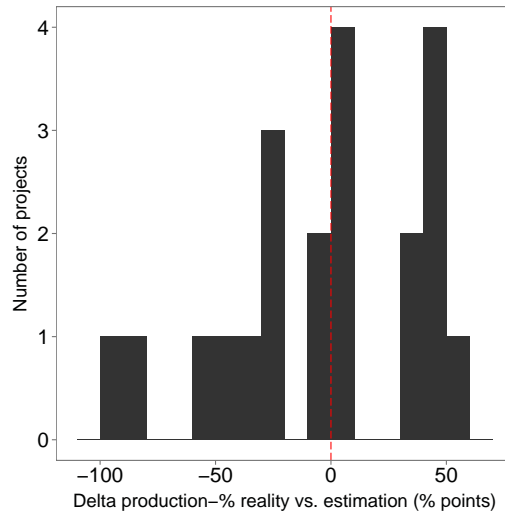


Figure 5.2: The delta between estimation and reality.

Table 5.2: Actual and estimated time distributions

	Actual (per project)	Estimated (per project)	Actual (per dataset)	Estimated (per dataset)
Production Eclipse	77%	63%	75%	52%
Test Eclipse	23%	37%	25%	48%
Production IntelliJ	60%	64%	64%	71%
Test IntelliJ	40%	36%	36%	29%

Table 5.3: Reading and Writing distributions

	Production Code	Test Code
Reading	52.28%	38.72%
Typing	47.72%	62.28%

a value of -100). While the Eclipse developers showed a tendency to overestimate the time they devote to testing by, on average, 14% per project, IntelliJ users were more realistic – they underestimated testing time by only 4% on average. We give the complete results in Table 5.2, columns marked as *per project*.

In the previous approach we evaluated the measured and the estimated times on a per-project-basis, so now we take another method and evaluate the whole dataset, by averaging across project and *not* normalizing for the contributed development time. While Eclipse developers in general also overestimated the time they spent engineering tests, IntelliJ developers appear to be more self-aware, by making their predictions deviate less from reality – they actually test 7% more than they anticipated. Furthermore, they spent less time engineering production code and more developing tests than the Eclipse users. In Table 5.2 we see the complete results in the columns *per dataset*. Finally, we also observed that for the production code users spent less than 5% more time reading it than typing it, whereas they wrote tests significantly more than they read them (61% versus 39%), see Table 5.3.

5.2 Discussion

In this section we clarify our results and compare them to the interpreted conclusions from the original study (see Table 5.4).

Even though we detected percentage-wise a bit more testing activities in IntelliJ than in Eclipse, that does not depart from the conclusion of the original study: the testing is not actively practiced (C.1). Even though we expected more testing-oriented users in our study, almost half of the projects did not contain any test.

For 65% of the projects we did not observe a single test execution, which leads to the conclusion C.2. Even when developers execute them, they usually have around 7 test executions per session on average, which is not as much as in Eclipse, but still not negligible – this comes as a consequence of three times smaller number of test executions in IntelliJ than in Eclipse. We can only assume that the tests are executed by using some external tool on developers’ machines, or pushed back to the CI servers which will automatically execute

them and notify developers about execution results. While we might expect that production code and test code evolve together, our findings do not firmly support this claim, since we have not observed a strong correlation between them (C.3).

Similarly to Eclipse, test executions inside the IDE are finished very fast (C.4), three quarters of them being finished in under five seconds. That, however, did not lead to more test executions, which the original study found that it was due to the test selection practice, when developers select only certain test cases to execute (C.5). Unfortunately, except for the fact that majority of IntelliJ test executions consists of a small number of tests, due to the small data-set size, we were not able to further examine this. On the other hand, we detected a small correlation between the test run duration and the number of test executions which could indicate that long tests are executed more frequently. If we assume that tests which take longer to finish are more complex, developers probably executed these tests more frequently to assure the correctness of some larger functionality. Other possibility is that it took more executions for these tests to assure their correctness.

Interestingly, the failing tests are observed two times more than successful test executions in both studies – 65% of the test executions fail (C.6). While Eclipse developers mostly check production code after a test failure, for IntelliJ developers production code is relevant only in the first seconds following the execution – afterwards, they showed a tendency to switch focus from IDE window, probably exploring the reason for the error in some external tool (or revert code changes using version control) or find the solution on the internet (C.7).

We found that TDD is not widely practiced, quite the contrary, we have not detected a single developer following it strictly (C.8). We were, however, able to observe some lenient use of it, although these developers followed it only during the small amount of their time. As expected, developers who practiced it have rather large programming experience.

In this study, we observed that IntelliJ developers spend a bit more time engineering tests than Eclipse users. They are also more self-aware – while Eclipse developers overestimated the time they spend on testing, IntelliJ users slightly underestimated that time (C.9). Considering the other analysis results and the survey input, in general we found that IntelliJ users tend to give more accurate and self-reflecting answers in the registration process (C.10), which was not the case in the original study.

5.3 Threats to validity

The original study introduced four possible types of threats and limitations that could affect the research process:

- *IDE limitation*: observing events only inside the IDE.
- *Construct validity*: integrity of WatchDog infrastructure and its correct functionality.
- *Internal validity*: threats inherent to the study, such as profile of the developers who contributed their data.
- *External validity*: generalizability of the results.

All of these threats refer to our research as well and in the remainder of the section we explain how we addressed them, finally coming to the conclusion **C.11** from Table 5.4.

IDE limitation. Identically to the original work, one of the main limitations is that we can only capture events that happen inside the IDE, that is, we are not able to capture work outside the IntelliJ, such as external compilation and test execution or changes which come from the version control systems (*git* or *svn*, for example). However, observing the absolute time was not the part of any of our research questions – we motivate our research by observing the proportion between time spent on production and test activities. Since external activities can include both production and test code engineering in the same amount as inside the IDE, we can assume that the ratio between them would be the same in both cases. Furthermore, it appears that another limitation can arise – many companies have dedicated quality assurance departments which only develop the test code, while developers write production code exclusively. However, this is out of the scope of this study, since we specifically examine how much *developers* test.

Construct validity. This threat concerns possible errors made in our study design, namely in the way WatchDog plugin works. Before the original study, the plugin was thoroughly tested and verified. IntelliJ plugin further contributes to this effort – although the results are not completely identical (which is expected), the majority of them raised the same conclusions. To further confirm the desired functionality of the plugins, we have also examined the scenarios in which the same routine is performed in both IDEs and we asserted the equal behavior and results in both cases.

Internal validity. This threat involves the problems with the collected input, such as tampered data and intentional misuse of our plugin, or unconsciously biased data. Because of the relatively small dataset, we were able to manually perform certain checks to ensure that the data reflects the real-world behavior. We did not observe any suspicious misuse of our plugin, nor the unusual activity from any user. However, since our plugin assesses testing behavior, we most likely have users to whom the testing is not the unfamiliar activity, hence we can conclude that our findings are probably an overestimation of the real testing practices. Finally, awareness of participants that they were being a part of an experiment and that they were being observed with respect to their testing habits can lead to slightly increased number of testing activities.

External validity. Ability to generalize the results was identified as a threat not only in the original study, but was also mentioned in the reviews of the work. While this study aims directly at that point, by extending the whole experiment to another IDE, a very obvious limitation of this work is the dataset size, which is significantly smaller than in the original study and which we have already mentioned multiple times. Nevertheless, this study is just a part of the whole TestRoots research effort which will eventually extend beyond Eclipse and IntelliJ IDEs.

5. ANALYSIS AND DISCUSSION OF THE RESEARCH QUESTIONS

Table 5.4: Comparison of Eclipse and IntelliJ conclusions

	Eclipse (original study)	IntelliJ (replication study)
C.1	The majority of projects and users do not practice testing actively.	Testing is not actively practiced in IntelliJ either.
C.2	Developers largely do not run tests in the IDE. However, when they do, they do it heftily.	Developers mostly do not execute tests in the IDE and when they do, they run them moderately.
C.3	Tests and production code do not co-evolve gracefully.	There is no significant correlation between production and test code changes.
C.4	Tests run in the IDE take a very short amount of time.	Test runs in IntelliJ last equally short as in Eclipse.
C.5	Developers frequently select a specific set of tests to run in the IDE. In most cases, developers execute one test.	Not assessable in this study.
C.6	Most test executions in the IDE fail.	Two thirds of test executions in the IDE fail.
C.7	The typical immediate reaction to a failing test is to dive into the offending production code.	The typical immediate reaction to a failing test is to either explore production code, or be inactive inside the IDE.
C.8	TDD is not widely practiced. Programmers who claim to do it, neither follow it strictly nor for all their modifications.	TDD is rarely practiced.
C.9	Developers spend a quarter of their time engineering tests in the IDE. They overestimated this number twofold.	Developers spend a bit more than a third of their time on testing activities. They slightly underestimated this amount.
C.10	Observed behavior often contradicted survey answers.	The actual results mostly reflect the survey answers.
C.11	Our conclusions are drawn from the precisely-defined and scoped setting of developer testing. To draw a complete picture of the state of testing, more multi-faceted research in different environments and settings is needed.	While the largest threat to validity in our study is the size of the dataset, we further address identified threats from the original study.

Chapter 6

Conclusions And Future Work

In this chapter we give an overview of the thesis’s contributions and conclusions, followed by the ideas for future work.

6.1 Conclusions and contributions

Beyond describing WatchDog’s architecture, we presented our experience with developing a family of IDE plugins for the WatchDog platform. We highlighted the benefits of light-weight, readily available solutions for software created by academic and start-up toolsmiths, often characterized by intermittent development and a low amount of available resources, both personal and financial. We also shared our concrete practical solutions so others can profit from the mature open-source WatchDog infrastructure.

In order to demonstrate its use, with release 1.5, we introduced the WatchDog IntelliJ plugin with the common core architecture described in this thesis in practice.¹ We used it to perform a small-scale study which assessed software testing practices of IntelliJ users and compared the results with the study of Eclipse users. We can safely conclude that users of different IDEs behave similarly, although IntelliJ users were more aware of their testing habits.

The main contributions of the work presented in this thesis are the following:

1. We explain the building process of a multi-platform, production-ready tool, developed in academic environment and used in software engineering research.
2. We perform a replication study of the previous TestRoots work [3] and compare results from different environments.
3. We set up a ground to facilitate and expand the existing research in the area of software testing.

Our contributions can be particularly beneficial for the following:

¹<https://github.com/TestRoots/watchdog/issues/193>

Researchers can utilize our methodology and approach in developing the multi-platform research software in academic environment. They should also be aware of the importance of testing, especially in scientific researches, and they should not allow themselves to underestimate it, like many engineers do.

Software developers, users of Eclipse and IntelliJ, can profit from WatchDog by obtaining 1) immediate and 2) aggregated feedback on their testing practices. If their results show the lack of testing activities, they should consider improving their testing practices. Furthermore, our open-source project can serve them as a model for building their own multi-platform tools.

IDE creators can consider improving the testing support inside the IDEs. On the top of the already made ideas[3], they can also add tools for automatic test generation, silent execution and provide the results of executions by using the “push” strategy (instead of “pull”, when developers inquire the execution and the results).

Even though we only present results from the small data-set, this study is essentially just a part of an open-ended, longitudinal field study which will most likely run for a period of several years and examine developers’ habits and patterns in detail. In this opportunity we can identify the third contribution of this thesis – it enables TestRoots team not only to gather more data, but also to easily expand their research to another IDEs and possibly other areas. Furthermore, we also propose some ideas for the future work in the following section.

6.2 Future work

The long-term TestRoots study of the software testing practices can be further enriched with the data from another programming languages and platforms. This includes, but is not limited to:

- Mobile applications development – Android (Java) and Android Studio can be taken as a very good example and first step. Android Studio is based on the IntelliJ IDE and applying WatchDog plugin to it should be straightforward, while Android uses Java programming language which we already support.
- The next step could be another IDEs from the JetBrains product family, such as RubyMine (the IDE for the programming language Ruby), PhpStorm (the IDE for PHP language) or PyCharm (the IDE for Python).
- Finally, the most challenging task would be to examine a completely different platform, such as .NET framework, with Visual Studio IDE and C# language.

Furthermore, it would be interesting to separately observe some seemingly special cases, such as the testing practices (or TDD patterns) for the new projects (a project which a developer starts writing from scratch) and compare the results with the projects which already have large code base. Additionally, we could also examine build automation files and get the tentative approximation of the usage of other tools to execute test executions.

Bibliography

- [1] Kent Beck. *Test Driven Development – by Example*. Addison Wesley, 2003.
- [2] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23. ACM, 2014.
- [3] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.
- [4] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (Much) Do Developers Test? In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, pages 559–562. IEEE Press, 2015.
- [5] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. How to catch ’em all: Watchdog, a family of ide plug-ins to assess testing. In *3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP 2016)*, page To appear. IEEE, 2016.
- [6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.
- [7] Frederick P Brooks. *The mythical man-month*, volume 1995. 1975.
- [8] Ivan Burazin. Most popular desktop IDEs & code editors in 2014. <https://blog.codeanywhere.com/most-popular-ides-code-editors>, Last visited: May 1st, 2016.
- [9] TIOBE Software BV. Tiobe index for april 2016. http://www.tiobe.com/tiobe_index, Last visited: May 1st, 2016.
- [10] Pierre Carbonnelle. Top IDE index. <https://pypl.github.io/IDE.html>, Last visited: May 1st, 2016.

- [11] Stephen Cass. The 2015 top ten programming languages. <http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>, Last visited: May 1st, 2016.
- [12] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
- [13] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.
- [14] Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 235–238. ACM, 2010.
- [15] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proc. Int’l Conf. on Software Engineering (ICSE)*, pages 485–495. IEEE, 2009.
- [16] Pavneet Singh Kochhar, Tegawendé F Bissyandé, Daniel Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 103–112. IEEE, 2013.
- [17] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.
- [18] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- [19] Roberto Minelli, Andrea Mocci, Mario Lanza, and Lorenzo Baracchi. Visualizing developer interactions. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 147–156. IEEE, 2014.
- [20] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming*, pages 552–576. Springer, 2013.
- [21] Jiantao Pan. Software testing. *Dependable Embedded Systems*, 5:2006, 1999.
- [22] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.
- [23] Romain Robbes and Michele Lanza. Spyware: A change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering*, pages 847–850. ACM, 2008.
- [24] Per Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.

- [25] Inc. Takipi. Githubs 10,000 most popular java projects - here are the top libraries they use. <http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/>, Last visited: May 14th, 2016.
- [26] Inc. Takipi. We analyzed 60,678 libraries on github - here are the top 100. <http://blog.takipi.com/we-analyzed-60678-libraries-on-github-here-are-the-top-100/>, Last visited: May 14th, 2016.
- [27] TestRoots. Testroots watchdog. https://testroots.org/testroots_watchdog.html, Last visited: October 18th, 2016.
- [28] Lars Vogel. Osgi modularity. <http://www.vogella.com/tutorials/OSGi/article.html>, Last visited: July 24th, 2016.
- [29] Wikipedia. Software testing — Wikipedia, the free encyclopedia, 2004. [Online; accessed 10-May-2016].
- [30] Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.
- [31] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

Appendix A

Travis Scripts For Building WatchDog Plugins

In this appendix we give the listings of our bash scripts for building WatchDog plugins. The first script executes the building process for the whole platform – WatchDog Server, Core, IntelliJ plugin and Eclipse plugin. The second script prepares IntelliJ SDK sources and it is executed during the main build, before the compilation of our sources.

```
#!/bin/bash
echo
echo Build WatchDogServer
echo
cd WatchDogServer
bundler
if [ ! -e config.yaml ];
then
  cp config.yaml.tmpl config.yaml
fi
rake
SERVER_STATUS=$?
cd ..

echo
echo Build WatchDogCore
echo
cd WatchDogCore/
mvn clean install
CORE_STATUS=$?
cd ..

echo
echo Build WatchDogIntelliJPlugin
echo
sh WatchDogIntelliJPlugin/fetchIdea.sh
cd WatchDogIntelliJPlugin/
```

A. TRAVIS SCRIPTS FOR BUILDING WATCHDOG PLUGINS

```
mvn clean verify
INTELLIJ_CLIENT_STATUS=$?
cd ..

echo
echo Build WatchDogEclipsePlugin
echo
cd WatchDogEclipsePlugin/
mvn integration-test -B
ECLIPSE_CLIENT_STATUS=$?

exit $((($SERVER_STATUS + $CORE_STATUS + $INTELLIJ_CLIENT_STATUS +
        $ECLIPSE_CLIENT_STATUS))
```

The following listing is the script for fetching IntelliJ IDEA source code, re-packing it, and installing it into the local Maven repository.

```
#!/bin/bash

idea_version="15.0.5"
idea_zip="ideaIC-${idea_version}.tar.gz"
idea_URL="http://download.jetbrains.com/idea/${idea_zip}"
build_dir="build_cache"

mkdir -p $build_dir
cd $build_dir

# Cache IntelliJ download. If not available, download anew (big!)
if [ ! -f $idea_zip ];
then
    echo "File $idea_zip not found. Loading from the Internetz ..."
    wget http://download.jetbrains.com/idea/${idea_zip}
fi

# Unzip IDEA
tar xzf ideaIC-${idea_version}.tar.gz

idea_path=$(find . -type d -name 'idea-IC*' | head -n 1)

if [ ! -f ${idea_path}.zip ];
then
    # Compress to ZIP file
    cd $idea_path
    zip -r ../${idea_path}.zip *
    cd ..
fi

cd ..

# Install IDEA to Maven repo
mvn install:install-file -Dfile=$build_dir/${idea_path}.zip
-DgroupId=org.jetbrains -DartifactId=org.jetbrains.intellij-ce
-Dversion=${idea_version} -Dpackaging=zip
```

Appendix B

Maven POM Files

In this appendix we provide the most relevant Maven POM (Project Object Model) files, which are used for dependency management and build process of WatchDog plugin. Here, we only present top-level build files, for each of the projects (Core, Eclipse plugin, IntelliJ plugin) while we omit the module specific POM files, but we make them available online¹. The presented files are important for establishing the automatic build process in the multi-platform environment, while the omitted files do not contain anything specifically relevant for this approach.

First, we have the POM file of WatchDog Core project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>nl.tudelft</groupId>
  <artifactId>parent</artifactId>
  <version>1.6.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>WatchDog</module>
    <module>WatchDogCoreFeature</module>
  </modules>

  <properties>
    <tycho.version>0.22.0</tycho.version>
  </properties>

  <build>
    <sourceDirectory>src</sourceDirectory>

    <plugins>
```

¹<https://github.com/TestRoots/>

B. MAVEN POM FILES

```
<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>tycho-maven-plugin</artifactId>
  <version>${tycho.version}</version>
  <extensions>true</extensions>
</plugin>

<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>target-platform-configuration</artifactId>
  <version>${tycho.version}</version>
</plugin>

<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>tycho-compiler-plugin</artifactId>
  <version>${tycho.version}</version>
</plugin>
</plugins>
</build>
</project>
```

We continue with the WatchDog Eclipse plugin main POM file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>nl.tudelft</groupId>
  <artifactId>parent</artifactId>
  <version>1.6.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>WatchDog</module>
    <module>WatchDogUnitTests</module>
    <module>WatchDogFeature</module>
    <module>p2updatesite</module>
  </modules>

  <properties>
    <tycho.version>0.22.0</tycho.version>
    <indigo-repo.url>http://download.eclipse.org/releases/indigo</indigo-repo.url>
    <juno-repo.url>http://download.eclipse.org/releases/juno</juno-repo.url>
    <kepler-repo.url>http://download.eclipse.org/releases/kepler</kepler-repo.url>
    <luna-repo.url>http://download.eclipse.org/releases/luna</luna-repo.url>
  </properties>
```

```
<repositories>
  <repository>
    <id>indigo</id>
    <url>${indigo-repo.url}</url>
    <layout>p2</layout>
  </repository>
  <repository>
    <id>juno</id>
    <url>${juno-repo.url}</url>
    <layout>p2</layout>
  </repository>
  <repository>
    <id>kepler</id>
    <url>${kepler-repo.url}</url>
    <layout>p2</layout>
  </repository>
  <repository>
    <id>luna</id>
    <url>${luna-repo.url}</url>
    <layout>p2</layout>
  </repository>
</repositories>

<build>
  <plugins>
    <plugin>
      <groupId>org.eclipse.tycho</groupId>
      <artifactId>tycho-maven-plugin</artifactId>
      <version>${tycho.version}</version>
      <extensions>>true</extensions>
    </plugin>

    <plugin>
      <groupId>org.eclipse.tycho</groupId>
      <artifactId>target-platform-configuration</artifactId>
      <version>${tycho.version}</version>
      <configuration>
        <environments>
          <environment>
            <os>linux</os>
            <ws>gtk</ws>
            <arch>x86</arch>
          </environment>
          <environment>
            <os>linux</os>
            <ws>gtk</ws>
            <arch>x86_64</arch>
          </environment>
        </environments>
      </configuration>
    </plugin>
  </plugins>
</build>
```

B. MAVEN POM FILES

```
        <os>win32</os>
        <ws>win32</ws>
        <arch>x86</arch>
    </environment>
    <environment>
        <os>win32</os>
        <ws>win32</ws>
        <arch>x86_64</arch>
    </environment>
    <environment>
        <os>macosx</os>
        <ws>cocoa</ws>
        <arch>x86_64</arch>
    </environment>
</environments>
</configuration>
</plugin>

<plugin>
    <groupId>org.eclipse.tycho</groupId>
    <artifactId>tycho-compiler-plugin</artifactId>
    <version>${tycho.version}</version>
    <configuration>
        <compilerArgument>-warn:+discouraged,forbidden</compilerArgument>
    </configuration>
</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-site-plugin</artifactId>
    <version>3.4</version>
    <configuration>
        <reportPlugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>findbugs-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <effort>Max</effort>
                    <threshold>High</threshold>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-pmd-plugin</artifactId>
                <version>3.2</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
```

```

        <artifactId>maven-surefire-report-plugin</artifactId>
        <version>2.17</version>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jxr-plugin</artifactId>
        <configuration>
            <linkJavadoc>>true</linkJavadoc>
        </configuration>
        <version>2.4</version>
    </plugin>
</reportPlugins>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

Finally, with the IntelliJ POM file we show how we manage the dependency to IntelliJ IDEA sources by using the extracted sources from the previous appendix.

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <relativePath>../pom.xml</relativePath>
        <groupId>nl.tudelft</groupId>
        <artifactId>parent</artifactId>
        <version>1.6.0</version>
    </parent>

    <artifactId>nl.tudelft.WatchDog</artifactId>
    <packaging>jar</packaging>

    <dependencies>

        <dependency>
            <groupId>org.jetbrains</groupId>
            <artifactId>org.jetbrains.intellij-ce</artifactId>
            <version>15.0.1</version>
            <type>zip</type>
            <scope>provided</scope>
        </dependency>
    </dependencies>

```

B. MAVEN POM FILES

```
<dependency>
  <groupId>nl.tudelft</groupId>
  <artifactId>nl.tudelft.WatchDogCore</artifactId>
  <version>1.6.0</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <!--We leave out other dependencies in this listing due to their
        length.-->
</dependency>

</dependencies>

<build>
  <sourceDirectory>src</sourceDirectory>
  <resources>
    <resource>
      <directory>resources</directory>
      <excludes>
        <exclude>zip.xml</exclude>
      </excludes>
      <filtering>true</filtering>
    </resource>
  </resources>
  <plugins>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.3</version>
      <executions>
        <execution>
          <id>unpack-intellij</id>
          <goals>
            <goal>unpack-dependencies</goal>
          </goals>
          <configuration>
            <includeArtifactIds>org.jetbrains.intellij-ce</includeArtifactIds>
            <outputDirectory>${project.build.directory}/IntelliJ-IDEA-CE</outputDirectory>
            <includes>**/*.jar</includes>
          </configuration>
        </execution>
      </executions>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
```

```

<version>2.5.1</version>
<configuration>
  <compilerArguments>
    <extdirs>${project.build.directory}/IntelliJ-IDEA-CE/lib/</extdirs>
  </compilerArguments>
</configuration>
<executions>
  <execution>
    <id>analyze-compile</id>
    <phase>compile</phase>
    <goals>
      <goal>compile</goal>
    </goals>
  </execution>
</executions>
</plugin>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/lib</outputDirectory>
        <excludeArtifactIds>org.jetbrains.intellij-ce</excludeArtifactIds>
        <excludeTransitive>true</excludeTransitive>
        <overwriteReleases>false</overwriteReleases>
        <overwriteSnapshots>false</overwriteSnapshots>
        <overwriteIfNewer>true</overwriteIfNewer>
      </configuration>
    </execution>
  </executions>
</plugin>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <outputDirectory>${project.build.directory}/lib</outputDirectory>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
      </manifest>
    </archive>
  </configuration>
</plugin>

```

B. MAVEN POM FILES

```
        </configuration>
    </plugin>

    <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.5.5</version>
        <configuration>
            <descriptors>
                <descriptor>resources/zip.xml</descriptor>
            </descriptors>
        </configuration>
        <executions>
            <execution>
                <id>make-zip</id>
                <phase>package</phase>
                <goals>
                    <goal>single</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>

</project>
```
