Delft University of Technology

TI3806 Bachelor Project

Coda: A Change Impact Analysis Tool for Scala

Marc MACKENBACH & Aaron ANG

Coach
Annibale PANICHELLA, PhD.

Client
Charlotte GOEDMAKERS, KeyLocker BV

Bachelor Project Coordinator
Martha LARSON & Felienne HERMANS

August 28, 2015

Preface

This report concludes the TI3806 Bachelor Project course at the Delft University of Technology and is part of the compulsory curriculum of the Bachelor of Computer Science. This report contains all information about the project that was carried out by two students over the course of sixteen weeks and the resulting product. The client for this Bachelor project was KeyLocker BV, a start-up that develops cryptography products and operates from Delft. During these sixteen weeks the students worked at least 24 hours a week, scrutinizing the software development process at KeyLocker and developing a software application that can assist in this development process. The goal of this report is to inform the reader about the work carried out by the students and serve as a reference to the company, documenting the considerations and design decisions made during the project. The report extensively discusses the quality of the delivered product and provides ample recommendations for future work.

Summary

KeyLocker, a start-up developing cryptographic products that put the control of encryption keys into the hands of the end-users, has requested a review of their own software development process. During their first year of existence, the company experienced problems in following software development methodologies, sharing knowledge effectively between employees, and with testing the software being developed. Being a start-up where its employees are always busy, the company requested outside assistance in analyzing their own processes.

Consequently, two students, who previously worked part-time at the company, were tasked with researching the development practices at KeyLocker, identifying problem areas. Subsequently, the students were charged with developing a software application that could assist in the development process, remedying some of the highlighted problems. Two problem areas of knowledge sharing and test maintenance were selected and research was conducted in the field of Change Impact Analysis - the identification of potential consequences of a change to components in software.

A tool was designed that could perform an analysis of the source code projects at the company. As Scala was the chief programming language used at the company, the application needed to be able to parse Scala source code and interpret changes to this source code. Then, the application needed to be able to analyze the code for potential consequences of those changes. Needless to say, developing such an application required an in depth knowledge of the Scala programming language.

The three main goals of designing such an application were utility, usability, and maintainability. In short, the software needed to solve the problems experienced at KeyLocker effectively, in a user-friendly manner, and be easy to maintain in the future. Over the course of sixteen weeks, the application has been developed with exactly these three goals in mind. During the completion phase of the project the application has been tested by the development team at KeyLocker, receiving a positive response. Nevertheless, there is room for improvement and some recommendations are provided for future work.

Acknowledgements

Although carried out by only two students from the Delft University of Technology, this project could not have been completed without the invaluable help of some people. We would like to take this opportunity to thank a few of them in particular. First and foremost, we would like to thank Annibale Panichella, our coach from the TU Delft, for his guidance on the technical aspects of the project and valuable advice when decisions needed to be made. Moreover, we would like to thank him for our incredibly fun and motivating weekly talks that went beyond the compulsory status update and did more than just keep us focused, they spurred us along. Secondly, we would like to thank Charlotte Goedmakers, our company supervisor, who closely managed us and our development process. She was instrumental in our self-improvement during the project, always asking us where we could have performed better and making us reflect on our own modus operandi. Furthermore, we would like to thank our colleagues at KeyLocker, especially Steffan Norberhuis and Remco Verhoef, for the insightful discussions we had during the project on the intended use and utility of our product under development. Finally, we would like to thank Martha Larson and Felienne Hermans for their assistance to our project in their capacity as Bachelor project coordinator.

Abbreviations

AST	Abstract Syntax Tree
CLI	Command-line Interface
IDE	Integrated Development Environment
JVM	Java Virtual Machine
MVP	Minimal Viable Product
MVC	Model-View-Controller
RegEx	Regular Expression
SIG	Software Improvement Group
UML	Unified Modeling Language

Contents

1 Introduction

Founded in 2014, KeyLocker is a start-up that is currently developing cryptographic protocols that put the control of encryption keys into the hands of the end-users, instead of service providers. Founded on the philosophy that Internet services should be 'given back' to the consumer, KeyLocker develops cryptographic products from the ground up that are unparalleled in the market. The one-year-old start-up has experienced problems in their development process over the course of their existence and has asked us to research and solve these problems (to some extent). The company has identified that the main issues in development lie with scaling the workforce from zero people with very little experience to a team of multiple people with various degrees of experience and different skill sets.

KeyLocker wants to know where in its history better decisions could have been made and what practices worked well. More importantly, KeyLocker wants to gain insight in how its development team can be more productive and produce higher quality code in the future. Therefore, after evaluating the problem areas of software development at the company, KeyLocker wants us to tackle at least one of the problem areas by developing a tool that will assist a small development team in the process of developing robust software for now and for the future.

1.1 Assignment

The assignment consists of performing a case study that researches the development practices of KeyLocker, identifying problem areas, and determining their root causes. Most importantly, we are required to focus on the scalability of the development team and on evaluating whether traditional practices scale well with the development of the company. After identifying the challenges of scaling the small company, we are charged with developing a tool that assists the company in the development of software and tackles at least one of the identified challenges and/or root causes.

What makes this bachelor's project unique from other projects, is the almost boundless freedom we enjoyed during the project. We were free to determine the problem the tool should solve and what form our solution should take, as long as it satisfied the needs and wishes of the development team. Of course, quality was a principal requirement, as well as integration into the current development work flow of KeyLocker's developers, but aside from that we enjoyed a lot of freedom.

1.2 Structure of this Report

This report will document the process and the results of the assignment. The report is structured as follows. In chapter 2 the KeyLocker company and its development team are closely studied and problems in its development process will be highlighted. After discussing the set of problems of software development discovered at KeyLocker, chapter 3 will select two problems from this set, further analyse the selected problems, and provide change impact analysis as a direction for a solution to develop. Then, in chapter 4 a software application named Coda will be proposed and its design will be discussed, outlining the design goals, requirements and providing an overview of the intended application. After having discussed the design for Coda, the actual implementation of the application will be examined in chapter 5, discussing the challenges, considerations and quality of the solution. Subsequently, chapter 6 will review the resulting product and chapter 7 will discuss both the product and its development process. Finally, the report will be concluded with recommendations for future work in chapter 8.

2 Case Study

As part of the assignment, the KeyLocker company and its software development approach shall be examined before choosing a problem area to develop a solution for. First, a brief history of the company will be provided, followed by a description of the development process at Keylocker. Then, a characterization of the encountered problems in software development at KeyLocker will be provided and the startup environment will be compared with an industrial software engineering company. Finally, we shall examine the methodologies applied at KeyLocker and determine why these did not work adequately.

2.1 History of KeyLocker

KeyLocker was founded in the summer of 2014 by an experienced management team, when its founders saw a growing need for encryption for both business and consumer and a great lack of expert products fulfilling that demand. The main problem in current cryptography products, they noticed, was not the implementation of strong and secure encryption algorithms (although that poses a significant challenge in itself), but rather keeping the secret keys used in these encryption schemes a secret. They set out to create a unique product, breaking new ground on encryption schemes and creating a new kind of market for encryption key management.

To create this product, the founders enlisted the help from some talented Bachelor and Master students from the Delft University of Technology. Aside from programming skills, the main requirement for these youngsters was that they would be able to handle the steep learning curve of the company. Most of what was under development had never been done before, in that way at least. Having to invent the wheel each day generated a huge amount of knowledge and experience for the developers, making the development as much a job as an education. Therefore, knowledge sharing was a crucial component for the company's success and this was promoted heavily by management.

Over the course of the first year of its existence, the size and composition of the development team at KeyLocker fluctuated greatly. In a matter of months, the development grew from two people to three, then to four people and then shrank back to three. After that, an experienced senior developer - a veteran of many projects - joined the team. This changed the composition of the team greatly, which before had consisted of mainly part-time student developers. During the months following these changes, two more students were hired. These fluctuations in size and composition of the development team increased overhead and called for a systematic on- and off-boarding

policy. Among other things, documentation of source code, learned principles, theory, and practice formed a vital part of this on-boarding process.

As the development team evolved, so did the design for the product. Not only the architecture changed, but also the intended application of the application. Being a cryptography product, KeyLocker's product could be used for many types of applications and over the course of the first year different market opportunities 'pulled' the design of the product to be tuned to that specific purpose. Moreover, new insights often provided a basis for (some) change in direction. However, in the summer of 2015 an order for a first proof-of-concept was placed by a Dutch government agency, firmly setting the direction of the product design. On top of that, other market opportunities started to play out, solidifying the design of the product even more.

2.2 Development Process at KeyLocker

In this section an overview of the development process of KeyLocker will be given. Although being inexperienced at developing industrial grade applications, the development team tried to apply some of the widely used practices in software development, such as agile and test-driven development. The development team of KeyLocker has defined a definition of done, which states the required procedure for approving an implemented feature. This procedure can be divided in three phases: design, implementation, code review.

2.2.1 Design

The development process starts off by ensuring that the requirements of the implemented feature are clear. Once the requirements are clear, the developer designs the required components by drawing UML diagrams. Afterwards, the technical design must be reviewed by another team member. The design review often results in a small brainstorm session in how the design can be improved, which then can be implemented. In the early stages of the development process it is important that mistakes are detected early, preventing unnecessary delays in development. Having brief review sessions enforces this.

2.2.2 Implementation

The definition of done requires that the implemented feature is thoroughly tested, therefore the developers try to follow the test-driven development process. However, sticking to this software development approach has proven to be difficult. Currently, the developers work with Eclipse[1] as integrated development environment. Version control is done by using Git[2] and GitLab[3]. Additionally, the builds are managed by

[1] `http://scala-ide.org/`
[2] `https://git-scm.com/`
[3] `https://about.gitlab.com/`

using Typesafe Activator[4], which is built upon the default Scala build tool sbt[5]. To test the implemented feature, the developers use Activator to run the tests. Once the implementation phase is done, the reviewing process starts.

2.2.3 Code Review

During the code review phase, another developer reviews the code of the first developer. The code review has two goals: to enforce software quality and to promote knowledge sharing. The reviewer checks if the code is readable, understandable and well engineered. The reviewing process makes use of pull requests. This makes it possible for the reviewer to see the differences between the old and new code. Furthermore, GitLab facilitates commenting on code, to which the author can reply and/or make another code change until the reviewer and the developer are satisfied with the implementation. The review is as much a control mechanism as it is a collaboration process.

2.3 Encountered Problems

After reviewing the development procedure, interviewing the developers and manager, and considering our own experience at the company, we identified some problems in software development at KeyLocker. One should note that we, Marc and Aaron, have been working at the company for almost a year and half a year, respectively, at the time of writing this report. Although often closely related and amplifying each others effects, these problems can be categorized as related to Scrum and product design, related to knowledge sharing, and related to testing:

2.3.1 Scrum Product Design

- **Estimation of task effort proved difficult**: during planning, the team often had difficulty estimating the effort of a proposed task and often estimated incorrectly, without improving much over multiple sprints.
- **Prioritization was an issue**: aside from uncertainty about task size, prioritizing the right tasks proved difficult sometimes. Often, sprint reviews indicated that some tasks were not finished that should have been prioritized over other tasks that were.
- **There was no clear product backlog**: the product changed much and often, resulting in a product backlog that was valid for a limited amount of time. Eventually, this resulted in the team neglecting to groom the backlog, leaving an unclear product backlog riddled with holes.

[4] `https://www.typesafe.com/community/core-tools/activator-and-sbt`
[5] `http://www.scala-sbt.org/`

- **Ad-hoc sprint-backlog selection**: the team decided ad-hoc on a per sprint (or a few sprints) what the backlog should be and what should be included in the next sprint.
- **Little oversight and direction**: overview/oversight depended on a few individuals who were already overburdened. These individuals were responsible for directing the team and the team did not effectively 'self-manage'.
- **Individuals were not equally employable**: not only did every team member work at different hours, all had different levels of experience and had enjoyed or were still completing various types of education. Therefore, team members were not equally able to pick up a random task, making the division of labor a complex matter.
- **Irregular part-time 'rhythm'**: because nearly all developers worked part-time, most time would be spent on meetings when developers would work on the same day, instead of on solving problems together. Not only did this influence the way problems would be solved, it caused the team to have great difficulty finding a rhythm. As opposed to working five days a week as a team, all developers being able to continue the next day where they left off the evening before, the team worked more as individuals at irregular intervals. This meant that the team would hardly ever gain the momentum one would expect.
- **Team maturity was not achieved**: as a team works together for more sprints it is assumed that the team improves itself iteratively, eventually reaching a 'mature' state where the different members can work together effectively and efficiently. As the team did not always improve enough each sprint, this 'mature' state was not achieved.

2.3.2 Knowledge Sharing

- **Areas of expertise**: as team members started to focus more on a specific area of development, areas of expertise started to form. Moreover, these areas of focus were carried over from one sprint to the next. Consequently, puddles of "inexpertise" arose, widening the gap between developers and reducing their ability to work on any component of the code base.
- **Poorly documented design decisions**: although documentation existed for what code was written, often design decisions were poorly documented. When poorly reviewed together, sharing these design decisions and the reasoning behind them became a burden.
- **On- and off-boarding strained company resources**: in such a small development team, the on- and off-boarding of a team member had an enormous impact. Especially because not one individual knew everything about the code base, the

entire team had to be actively involved in on-boarding a new employee. Because of the areas of expertise, this expert knowledge had to be transferred to the team upon off-boarding, requiring active participation of all team members.

2.3.3 Testing

- **No use of code metrics**: no code metrics were used for reviewing code, such as code coverage metrics. This meant that detecting poor quality code and early detection of bugs was a lot harder, if performed at all.

- **Absence of continuous integration**: no form of automated continuous integration was in place. Although code reviews were performed, code was often not run on the machine of the reviewer. This meant that the integrity of the 'working product' could not bet guaranteed entirely. Moreover, code would be tested by the developer before being pushed to the version control system, but it was by no means guaranteed that all tests had been run by the developer, neither did any mechanism exist to verify afterwards that the version of source code was working. This meant that there was no guarantee that the code worked in any environment other than the workstation of the developer, let alone in a production environment.

- **Inadequate code reviews**: although reviewers would walk through the implemented functionality for review, hardly ever did reviewers perform in-depth analysis of the test code belonging to that functionality.

- **Test maintenance**: although tests of the touched pieces of source code were updated when introducing a change, it was not uncommon that a developer would stumble upon some outdated tests. Apparently, not all tests were updated each time a change was introduced. Moreover, tests were rarely added to existing components.

2.4 Root Causes

Once these problems are identified, it is necessary to analyze why these problems occurred, and how they could have been avoided. Something that is obvious from the start is that most of these problems have not much to do with personal ineptitude, but constitute a collective organizational problem.

Interesting to note is that many theories and best practices exist for development in teams of 3 to 9 people (such as Scrum [26]), but little is written for building a software development team from scratch, starting from zero people with very little experience. There are very few best practices, or even guidelines, for building and growing the development team as happened at KeyLocker. The development team tried to employ a hybrid form of Scrum, which offered some relief in the form of guidelines and practices.

However, applying these practices correctly and consistently to a process subject to so much change as was the case at KeyLocker proved a difficult task.

When examining the history of KeyLocker, and comparing it against an 'industrial' development team within an established environment, three striking differences become clear: team dynamics, team homogeneity, and direction drivers.

2.4.1 Team Dynamics

In this start-up the team dynamics differ a lot from a mature software development organization. At KeyLocker, team size was determined by the course of the company. As start-ups have the tendency to fluctuate heavily between explosive growth and downsizing, team size at KeyLocker tended to fluctuate as well. On the other hand, a mature organizations is not impacted (that much) by the course of the organization, as the continuity of the organization is more or less guaranteed.

2.4.2 Team Homogeneity

The development team at KeyLocker consisted of individuals who greatly differ in age, experience, working hours and education. However, the development team of a mature organization will generally have individuals who are more equal in these matters. This can greatly impact the synergy of the team and the ability of members to be equally capable of picking up random tasks during develpoment. In a team such as the one at KeyLocker, individuals were employable for a very distinct task for a very limited amount of time, at various skill levels.

2.4.3 Direction Drivers

In a start-up the direction of product development is often determined by the vision of its founders, whereas the direction of an industrial development team is largely determined by concrete customer demand. As the start-up grows, this vision is adjusted and/or fundamentally changed to newfound market insight.This means that the direction of the product can change a lot and very often in a start-up environment. When KeyLocker entered the cryptography business, with little knowledge of the market beforehand, this was especially the case.

In short, the average start-up provides a far less stable environment than its industrial established counterpart. Many software development methodologies, such as Scrum, exist that enable developers to be flexible in the product under development. For this to succeed, however, a certain stability in the environment is generally necessary. The prevailing assumption is that a more or less a stable team can improve itself as time progresses. Additionally, having a more or less homogeneous team where each member is equal in his/her ability to pick up a random task is beneficial to the

process. Finally, many modern software development practices, such as Scrum, are designed for fine-tuning the product development to the customer's wishes.

This is exactly what caused many of the problems experienced at KeyLocker. For, the team changed too much and changes were too impactful to reach this team maturity, the team was too heterogeneous, and there was no concrete customer demand with direction being driven by a changing vision of what the customer would want in the future, or ought to want. With this in mind, it is not surprising that the development process at KeyLocker suffered as much as it did. Being very critical of our own approach while working at the company, we see these problems pose a serious challenge to the company's future success.

3 Problem

The case study has brought to light many problems in software development at KeyLocker, some of which are solvable within the scope of this project. In this chapter a selection of those problems that we intend to solve will be presented, gathered into one comprehensive problem definition and finally change impact analysis will be discussed as a solution to these problems.

3.1 Problem Analysis

Now that the problems have been identified and some light has been shed on their causes, we will select an area of improvement and propose a solution that will solve some of the issues. One thing that strikes even the most superficial reader is that most of the problems highlighted by the case study are organizational problems. With enough effort and discipline, adhering to best practices of software development and accounting for the development team's unstable nature, most of these problems can be remedied.

Although reorganizing the company's development process falls outside the scope of the project, an obvious solution would be to eliminate the three root causes outlined before. Instead, our solution must consist of a technical application that supports the development process and helps developers to adhere to the best practices in software development.

Therefore, we will tackle some problems that are more technical in nature, problem areas where developers could benefit from some assistance by ways of a software application. More specifically, a solution will be developed that tries to improve the spread of knowledge throughout the company and simultaneously increase the quality of the code base. Below, the two problem areas to tackle will be defined as a knowledge sharing problem and a testing problem, both of which reinforce each other.

3.1.1 Knowledge Sharing Problem

As indicated before, sharing knowledge between developers at KeyLocker had become a burden. Due to the complexity of code, most developers do not have the required knowledge and insight about the (complete) code base. This makes it difficult for the developers to change or add code and to create integration tests and end-to-end tests for parts of the code base they are not familiar (enough) with.

This can be an absolute productivity-killer. Although asking another developer for assistance might seem beneficial for sharing knowledge, doing so will cut the productivity

of the other developer. One way or another, the productivity of the team suffers because of this disparate division of knowledge. Therefore, we are looking for a solution that can provide insight into code projects and increase the developers understanding of a code base, independently from other developers.

3.1.2 Testing Problem

Due to developer's lack of insight, small code changes are bound to introduce bugs into the system that the developers can remain unaware of until it is too late and the bug disrupts a user's experience. Something that ought to catch most erroneous changes and additions to source code is test maintenance. Keeping test suites up to date to software changes provides, to some extent, assurance that no bugs are inadvertently introduced.

Moreover, maintaining the complete test suite has proved difficult for the KeyLocker development team. Neglecting to maintain a test suite creates a false sense of confidence. This is known as the pesticide paradox principle [14]. According to the pesticide paradox principle, a test suite should be maintained and new test code should be added for already tested code to catch possible new bugs introduced in later stages.

Additionally, written test cases are not reviewed during code reviews. A good test suite ensures that code quality is high and should focus on finding defects in the code. If a test suite is not written well and not reviewed, there is a high probability that defects are not detected. According to the RIP (Reach, Infect, Propagate) [3] principle all code should be covered (reached) to detect defects in the code. Correct usage of test coverage can benefit the development team to enforce the RIP principle.

So, productivity suffers because of poor knowledge sharing and software quality suffers because of poor test maintainability. Because developers are not familiar enough with the code base, tests are maintained even worse. For, how is a developer supposed to maintain tests, if he does not know which tests to maintain? Moreover, because tests are incomplete, outdated or simply absent, developers will have a worse time trying to understand the code base. Without high quality and up-to-date test suites, knowledge sharing suffers. These two problems are clearly tightly related and can strongly influence each other, spiraling downwards to a state of poor productivity and poor software quality. It is exactly these two productivity and quality problems that will be tackled during this project by developing a tool that performs change impact analysis.

3.2 Change Impact Analysis

In this chapter change impact analysis will be discussed, starting off with an explanation what exactly change impact analysis is. Afterwards, we will discuss why change impact analysis is relevant for the productivity and quality problems mentioned before. Subsequently, dynamic and static change impact analysis will be discussed. Finally,

an overview of existing code based change impact analysis techniques will be given.

Change Impact Analysis is a technique used in software engineering for estimating affected areas when a change is made to source code. A definition of change impact analysis has been proposed by Bohner and Arnold as:

> "Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change." [7]

Impact Analysis is necessary because code changes might work on their own, but might break or change existing functionality. This is especially the case for object-oriented programming languages, because object-oriented code is based on the communication and interaction between objects. This interaction can cause ripple effects of impact throughout a codebase, if improperly designed.

For example, assume components A, B and C exist and component B is directly dependent on component A, because it interacts with component A. Changing component A might affect or break the functionality of component B. Additionally, assume that component C is directly dependent on component B. Changes in component A might affect component C, because component C is indirectly dependent on component A by being directly dependent on component B. Changes in component A can greatly impact dependent components B and C. Consequently, several ways exist in which changes in component A can render testing of dependent components B and C incomplete or incorrect:

- **Tests of dependent components fail.** This is actually good, as the impact is detected by failing tests (assuming tests are evaluated after introducing changes). The developer can then simply correct the broken functionality.
- **New behavior is not tested.** If new behavior is introduced in a class upon which other classes depend, the tests of the dependent classes are not aware of this change. The current set of test cases of the dependent components is not aware of the new behavior of the changed component. This means that either the new behavior makes the existing test cases fail, which is good, or the new behavior is simply not tested, which is very bad. When left untested, it remains unknown whether the dependent classes have been broken or otherwise affected. This creates a false sense of code quality and is also known as the pesticide paradox principle [14].
- **Tests of dependent components pass, but are no longer correct.** For instance, if tests for component B use a mocked version component A, not reflecting the new changes in component A, test results of component B will be incorrect. This is where things start to go wrong. When running tests for the dependent components, nothing shows up. However, the dependent components might very well have been impacted, without the developer or any form of continuous integration detecting it.

The first type of effect above is relatively obvious to developers, however the other two might not be so straightforwardly apparent. This is where change impact analysis comes in.

3.2.1 Relevance

Software that employs change impact analysis could solve the problems defined above. Change impact analysis gives developers an overview of what to change when code is refactored or redesigned. This knowledge of what to change not only ensures that the right tests are written, but also confronts the developers with what tests should be written, making it harder to neglect testing. This results in better testing, code coverage, improving executable documentation and code quality, lessening the burden of knowledge sharing. In addition, it is possible to use change impact analysis for estimation of task effort, before actually starting on the implementation of that task. By touching the right components, the analysis could provide insight into the complexity of fixing a bug or implementing a feature.

As far as we know, currently there is no tool for Scala that does such kinds of analyses. With an ever growing Scala code base both at KeyLocker and elsewhere, this practice becomes especially relevant. Therefore, a tool that can perform change impact analysis will greatly benefit the KeyLocker development team.

3.2.2 Scope

Change impact analysis has already undergone a lot of research [20] [21]. S. Lehnert [20] reviews most of the change impact analysis techniques and mentions that change impact analysis can be done in three different scopes: source code, models and miscellaneous artifacts. The majority of change impact analysis approaches are done in source code. This approach focuses mainly on the relations between source code entities and how they interact with each other [5] [8]. Model based change impact analysis focuses on high level representations of the system, for example UML diagrams and requirements models [23] [6]. Miscellaneous artifacts change impact analysis deals with changes made to documentation, configuration files, and other auxiliary tools [25] [17].

During the bachelor's project we will focus on source code-based change impact analysis, because KeyLocker's system models and miscellaneous artifacts are relatively small in size compared to the wealth of information we could extract from their code base. Moreover, most research is done on source code-based change impact analysis [20], improving the chances of developing a tool that produces useful results.

3.2.3 Analysis Techniques

Source code-based change impact analysis can be performed in various ways, most notably static and dynamic analysis of source code [21]. Static analysis is mainly

focused on analyzing graphs constructed from source code and is able to detect impacted components without executing source code. However, this can be inaccurate in object-oriented programs, especially when polymorphism is used [2]. In contrast to static analysis, dynamic analysis does execute source code. Dynamic analysis can be more accurate than static analysis by analyzing the execution traces of a program and does solve the polymorphism problem [18]. However, dynamic analysis does only analyze executed statements, unable to detect impact in parts of source code that are not executed. Moreover, dynamic analysis causes a bigger overhead in both time and space [16] than static analysis. Because the solution to build for KeyLocker needs to be kept as light as possible, as will be discussed in chapter 4, we therefore chose to implement static analysis and only combine it with dynamic analysis if it fit in the project's timeline (which it did not).

3.2.4 Algorithm

Many techniques exist for performing static change impact analysis [1]. For this project, we will be using an algorithm proposed by Rajlich named "top-down-approach" [24] due to its simplicity and rapid execution. This algorithm presupposes that a dependency graph has been generated, on which the algorithm can be run. This dependency graph consists of nodes that represent source code components, e.g. classes or methods, that share a directed edge if the components depend on each other, e.g. extend or use each other. The algorithm computes the minimal distance of nodes in the graph to the changed nodes and uses this as a quantification of the likelihood that a node is impacted.

In figure 3.1 a dependency graph is shown containing four classes: A, B, C, and D. The edge from A to B indicates that A is using B, therefore a dependency exists, e.g. A extends B. For the same reason edges (B, C), (B, D), and (D, A) exist. Note, that this notation will be used in further discussion, where (u, v) refers to a directed edge that exists between node u and node v, starting from node u and ending in node v. The number next to the name represents the distance of the node, which is equal to the minimum distance to the change set. The change set contains class nodes that are modified. In this case, the change set consists of node B. The goal of impact analysis is to use the dependency graph and the change set to determine the impact set. The impact set contains all classes that are impacted by the change set, i.e. classes that potentially no longer function correctly due to a change. In figure 3.1 the impact set consists of class nodes A and D. Class A is directly dependent on class B and class D is indirectly dependent on class B via class A.

Determining the depth of a node can be done by using a breadth-first traversal described in algorithm 1.

The algorithm visits every node and edge once, therefore the runtime complexity of this algorithm is equal to $O(n + m)$, where n represents the total amount of nodes and m represents the total amount of edges in the dependency graph. Of course, the run time can be improved by introducing a stopcriterion such as the maximum depth to

Algorithm 1 Impact analysis using breadth-first traversal.

Require: $G \leftarrow$ Dependency graph
Require: $C \leftarrow$ Change set, where $C \subseteq G$

```
S ← Empty set
distance ← 0

for all n ← node in G do
  n.distance ← ∞
end for

for all n ← node in C do
  S.add(n)
end for

while !S.isEmpty do
  N ← Empty set
  for all n ← node in S do
    if distance < n.distance then
      n.distance ← distance
      for all e ← incoming edge of n do
        s ← source node of e
        if distance + 1 < s.distance then
          N.add(s)
        end if
      end for
    end if
  end for
  distance ← distance + 1
  S ← N
end while

return G
```

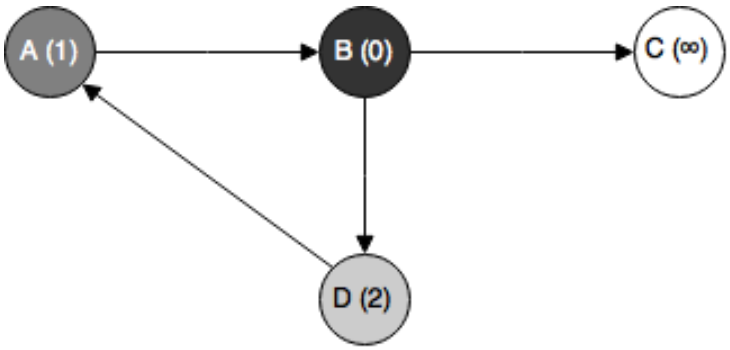

Figure 3.1: Depth-based change impact analysis. The darkest color indicates the change set. The varying grey nodes indicate the impact set.

analyze. One could imagine a scenario where analyzing very far from the changed nodes can produce diminishing returns when it comes to usefulness of the results.

Once the depth for each node is computed it can be concluded that nodes with a lower depth are more likely to be impacted. Although the distance to changed nodes does not represent actual impact, it does provide a quantification of the likelihood that a node is impacted, compared to the other nodes.

Now that the problem has been analyzed and change impact analysis has been proposed as a technique to solve the problem, the next chapter will discuss the design of a software application that employs this technique to do exactly that.

4 Design

In this chapter, the design of the product under development will be discussed. First, the design goals will be formulated, followed by a definition of the requirements of the project. Subsequently, an overview of the design of the application will be provided.

4.1 Design Goals

The design goals form the guiding principles in designing, implementing, and deploying the proposed solution. Moreover, originating from these goals the requirements have been determined that will be discussed in the next section. The design goals can be roughly categorized as utility, usability, and maintainability.

4.1.1 Utility

First and foremost, the solution should be useful. It should fulfill the developers' needs and solve the problems defined before. Most notably, it should fill the knowledge gap of dependencies between source code components. The utility of the solution is determined chiefly by three factors: result quality, performance, and developer satisfaction.

- **Result Quality**: the solution is only useful if the impact set computed by the algorithm is relevant for the development process. It is important that, for each suggested component, the computed impact accurately describes the actual impact. The application should strike a balance between suggesting relevant pieces of source code to inspect and not flooding the user with false positives. Moreover, the impact set should contain all the impacted components. For, if developers cannot rely on the produced results being precise, i.e. containing all impacted components, they would still need to perform a manual impact analysis themselves. Finally, the results should be consistent across multiple runs, providing a reliable basis for development. In short, results should be accurate, precise and consistent. Accuracy and precision can be measured by computing the expected results manually and comparing them with the results of the application. Additionally, consistency can be determined by comparing multiple runs of the application with each other.

- **Performance**: as a rule of thumb, the solution should be able to produce the same results or better in less time than an experienced developer with in-depth knowledge of the system would. Consulting the analysis should take less time

than asking an experienced developer for help. Therefore, performance of the solution is determined by the speed at which results are produced. The speed of the implemented solution can be determined by starting a timer at the start of the application and ending the timer once the application has finished its analysis.

- **Developer Satisfaction**: for the application to be of any use, the developers themselves must believe in the utility of the application. The tool should provide a clear added value to the developer and developers must be satisfied with the information the application can provide, without having to ask someone for help. Developer satisfaction can be measured by surveying developers who have used the tool in a development situation.

4.1.2 Usability

Aside from being useful, the solution should also be usable. The solution should assist the user and not hinder them in their development process. The usability of the solution is determined by three main factors: comprehension, integration into the development process, and intuitiveness.

- **Result Comprehension**: the results of impact analysis that the tool provides should be comprehensible to the user. The solution should make every effort to make the output legible and understandable to the user. On a most basic level, this boils down to not only providing the 'what', but also the 'why'. For instance, the user should not only be able to see that some part of the code is impacted, but should also be able to understand to some extent why and how the code is impacted. Most importantly, the results should not confuse the user, but rather empower them.

- **Integration into the Development Process**: the solution should not be a hindrance to the developer, but rather speed up and simplify their work flow. Therefore, seamless integration into the development process is a principal design goal. Using the application should take place at a natural point in the development process, e.g. just before committing source code changes.

- **Intuitiveness**: users should intuitively know how to use the solution without much instruction. This means that the user interaction with the application should fit well within previous experience of the users. As the users will be software developers, the user interaction should resemble that of applications they are familiar with. In short, interaction with the solution should make sense to the developer and should require as few additional overhead for the developer as possible.

4.1.3 Maintainability

As development will be iterative and during the course of the BEP the solution will grow more and more sophisticated, maintainability is a key design goal. Moreover, the company will most likely continue development after the BEP is finished. This means that the code should be easily understandable and modifiable by another developer, even after the original authors might be unavailable for counsel. Below, three vital elements of maintainability are outlined: code comprehension, test quality, and modular independence.

- **Code Comprehension**: the source code must be easy to understand by a developer with no prior knowledge of the project itself. For this to be achieved, the code must be readable, documented, simple, cohesive, and tested. First and foremost, improving readability of source code involves using consistent and self-explanatory naming schemes, consistent and functional use of white-space and indentation, limited line-length, and more. Additionally, documentation that accurately and concisely describes the function of source code is indispensable for writing comprehensible code. This means that documentation should provide additional information where needed, but refrain from flooding the reader with useless and superfluous comments in source code. Moreover, crucial for comprehensible code is simple code. In short, code complexity should be kept to a minimum, refraining from using deeply nested and convoluted control flows. Additionally, code cohesion should be high for comprehensible code. Each component in the system should have one clearly defined job and components should be grouped according to those jobs. Finally, tests function as executable documentation, providing the developer with information of the expected behavior of components. Therefore, source code should be tested extensively to increase code comprehension.

- **Test Quality**: besides functioning as executable documentation, testing source code provides quality assurance. Tests ensure that the application functions as intended and still functions properly after introducing a change in the source code. When testing is performed well, changing source code will be less likely to introduce bugs. Therefore, test quality is vital for maintainability. Two main factors determining test quality are code coverage and test coverage. Code coverage metrics express the extent to which tests cover the source code. Code coverage can be analyzed on different levels, be it class, method, or line level. High code coverage means that most behavior of the application has been evaluated at least once. Almost as important as code coverage is test coverage. Similarly, test coverage measures the extent to which the tests cover the business scenarios of the application. High test coverage mostly ensures the application meets the requirements of the customer.

- **Modular Independence**: modular independence improves testability and error traceability in source code, greatly improving maintainability. Moreover, modular

independence makes the system much more open to modification and extension. Especially since the company might want to make the tool Open Source in the future, easy modification and extension is key. Adhering to the best practices of software engineering, the project should contain low coupling between components, i.e. contain few direct dependencies between components. To achieve this, it is important that software engineering best practices are employed where needed.

4.2 Requirements

Stemming from the design goals outlined above, the following requirements have been defined for the application. The requirements are divided between functional and non-functional requirements. Functional requirements specify *what* the application should do and non-functional requirements place constraints on *how* the application should do so. It is important to note that fulfillment of each requirement should be measurable. Functional requirements are measured by testing whether the desired functionality exists, while non-functional requirements can be quantified and measured.

4.2.1 Functional Requirements

The application must ...

1. ... be able to parse Scala source code.
2. ... be able to detect class definitions.
3. ... be able to detect full names of classes.
4. ... be able to detect parent-child relations between classes.
5. ... be able to compute the impact of a change on class level.
6. ... be able to create a dependency graph of a Scala project on class level.
7. ... be able to detect the changed set of classes when a change is introduced in source code.
8. ... be able to output impacted classes to the user, prioritized by relevance.
9. ... be executable from the command line.

The application should ...

10. ... be able to parse most commonly used types of valid Scala source code and combinations thereof.

The application could ...

11. ... visualize the impact graph to the user.
12. ... parse all possible combinations of valid Scala source code.
13. ... persist the results in a log or a database.

The application won't ...

14. ... automatically select and execute test cases for impacted classes.

4.2.2 Non-functional Requirements

Utility

1. The application should produce results within 5 seconds.
2. The application should have a precision of at least 76%.
3. The application should have a recall of at least 94%.
4. The application should consistently produce the same results on the same input.
5. Users should rate accuracy at least average.
6. Users should rate usefulness of the results at least average.
7. Users should indicate that they would use the application in the future at least average.

Usability

8. Users should rate result comprehension at least average.
9. Users should rate the seamlessness of the integration into the development process at least average.
10. Users should rate the intuitiveness of the application at least average.

Maintainability

11. Statement coverage should be at least 85% [12].
12. Methods should not consist of more than 100 lines of code and hardly ever exceed 20 lines of code [22].
13. Each class and method should be documented and contain ScalaDoc comments.
14. Each class should contain as few in-line comments as possible. Where it is present, it usually indicates poor adherence to naming conventions and convoluted control flows.
15. Components must have a higher cohesion than coupling.
16. The medium to display results should be isolated from the results themselves, i.e. the display must be able to change without needing to change anything else.
17. Different algorithms for computing impact must be easily interchangeable.
18. Different types of input must be easily interchangeable.

Requirement 1 is based on the compile speed of sbt, used by the KeyLocker development team. Running the complete test suite of KeyLocker took at least 6 seconds, therefore the impact analysis application should produce results within 5 seconds, being faster than sbt.

Requirements 2 and 3 are based on research done by Lile Hattori et al. [15]. Lile Hattori et al. developed Impala to perform static analysis on Java source code. The re-

sults of Impala had an average precision of 76% and an average recall of 94%. According to Jesse Davis et al. precision is computed as follows [11]: $Precision = \frac{TP}{TP+FP}$, where TP is the total amount of true positives, and FP is the total amount of false positives detected by the application. Furthermore, their research states that recall can be calculated as follows: $Recall = \frac{TP}{TP+FN}$, where TP is the total amount of true positives, and FN is the total amount of false negatives detected by the application.

Requirements 5, 6, 7, 8, 9, and 10 should all score a score at least average. The developers of KeyLocker will be asked to experiment with the application. Subsequently, they are asked to answer a survey that will verify if these requirements are met.

Finally, requirement 15 is defined, because KeyLocker demands that software quality of written code is high, the coupling and cohesion of classes will be analyzed. Coupling is determined by the degree of interdependence between parts of the design [10], i.e. the amount of external method calls. Cohesion is determined by the degree of internal dependence within parts of the design [10], i.e. the amount of internal method calls.

4.3 Application Overview

To meet the requirements formulated above and fill the knowledge gap of dependencies in source code, we propose **Coda**. **Coda** is short for CODe Analysis and consists of a software application that analyzes source code with its changes and provides the developer with a collection of impacted components. In figure 4.1 the activity diagram of the application's core is depicted.

The application starts with the source code of a Scala project that has to be analyzed by Coda. Based on the source code a dependency graph is constructed and changed source code is derived which results in the change set. Subsequently, a computation can be done with the dependency graph of the source code and the change set resulting in an impact graph. Finally an impact set can be derived which can be used in various ways. In-depth analysis of every component in the activity diagram will given in the next chapter.

The impact set can be used for various purposes. In the context of the bachelor's thesis and the needs of KeyLocker the following potential applications were formulated:

- **Visualization of the Impact Set**

  Visualizing the impact set can help the developer with identifying possible impacted classes. Furthermore, additional information can be provided to the developer like type of relationships between classes and likelihood of impact. The possibilities to visualize the impact set are virtually endless, therefore only two solutions will be discussed.

  - **To-Do List:** A to-do list is a simple solution that provides the developer with a list of possible impacted classes that should be reviewed. Additionally, the

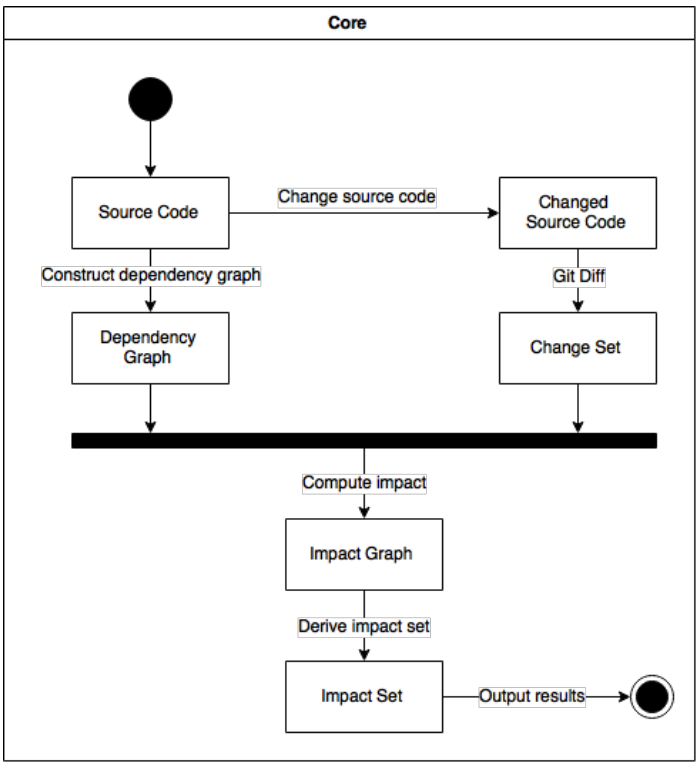

Figure 4.1: An activity diagram of the application

likelihood of impact can be shown by sorting the list from likely impacted to unlikely impacted.

– **Graph Visualization:** A graph gives an overview of all classes and relationships between them. Usage of colors, variation in edge length and node size can all be used to provide a way to indicate impacted nodes.

- **Automatic Regression Testing**

  Automatic regression testing can be done by running all test suites related to the impacted classes. Automatic regression testing is useful when running the full test suite is impractical and time consuming.

The purpose of Coda has been discussed with the KeyLocker development team and these potential applications were presented to the team. The development team then provided feedback and chose for a simple to-do list accessible from the command-line interface (CLI). KeyLocker suggested that the project focuses on the application's core logic, i.e. parsing source code, parsing changed code, analyzing it for dependencies, and computing the impact set. Displaying the impact set was seen as a minor activity, one that could always be altered later. In fact, they required Coda to only

output an ordered list of impacted components. Moreover, adhering to a model-view-controller (MVC) architecture enforces that core logic is separated from the view, which allows implementation of different views, e.g. graph visualization.

Furthermore, the team deemed that automatic regression testing had little to no added value for the company, because of the relatively small size of the existing code base. The developers had no issues with always running the complete test suite. Therefore, this application type was discarded for this bachelor's project.

Now that the design goals, requirements and intended application of the product have been discussed, the next chapter will discuss the implementation of that application.

5 Implementation

In this chapter, an overview of our own development process will be provided, followed by a description of the most impactful design decisions made during development, along with their motivation. The chapter will be concluded with the feedback from the Software Improvement Group[6] (SIG) on the quality of our code and our actions in response to that feedback.

5.1 Development Process

In this section an overview of our development process will be given. We developed in sprints of 1 week following a kind of hybrid Scrum method. The core of the process was focused on defining a narrow as possible Minimally Viable Product (MVP) that included the full stack of the application flow described in chapter 4, to be able to test out the application as soon as possible and validate its usefulness and usability. After the initial design phase described in chapter 4, the iterative development process of the sprints can be divided into three distinct phases.

5.1.1 Design

The development process starts off by selecting a set of features to be included in the next sprint. This selection happens during the weekly meetings with our company supervisor and the meetings with our TU Delft supervisor. We ensure that the requirements of the chosen features are clear and that both supervisors are satisfied with the direction of the project. Once the requirements are clear, the design is often done in a small brainstorm session and some UML might be drawn to communicate. If necessary the design decisions will be documented accompanied by some diagrams, after which one of the developers can take up the task of implementing the feature.

5.1.2 Implementation

Because we require that every implemented feature is thoroughly tested, we try to follow the test-driven development process. Especially for interpreting different types of Scala patterns proved to be heavily reliant on testing, due to its complex and sometimes unpredictable nature. Naturally, our Scala analysis tool is developed in Scala. We use IntelliJ[7] as an integrated development environment and version control is per-

[6] https://www.sig.eu/
[7] https://www.jetbrains.com/idea/

formed by using Git and GitHub[8]. Additionally, the builds are managed by using the default Scala build tool sbt. To test implemented features, developers use sbt to run the tests. Once the implementation phase is finished, the reviewing process starts.

5.1.3 Review

During the review phase, the authoring developer makes a pull request and the other developer will have to review it, before it can me merged into the sprint branch. This makes it possible for the reviewer to see the differences between the old and new code. The reviewer can comment on code, to which the author can reply and/or make another code change until the reviewer and the author are satisfied with the implementation. The review enforces the quality of the software and promotes knowledge sharing. The reviewer checks if the code is readable, understandable, tested, and engineered well. When both author and reviewer are satisfied, the feature is merged into the sprint branch. Using Travis[9] for continuous integration ensures that the product and intermediate products work and pass the tests. At the end of the week, the features are merged into the working product and reviewed with the supervisors, their feedback is processed and the development cycle can start anew.

During development many design decisions were made, the most notable of which have been documented in the next section.

5.2 Design Decisions

Over the course of the project, we encountered many challenges. Faced with a multitude of potential solutions to those challenges decisions had to be made, choosing the right solution where the appropriate solution was not always obvious. The most notable design decisions that had the greatest impact on the resulting product are described below.

5.2.1 Model-View-Controller

During the project, one of the major design choices is the implementation of the Model-View-Controller (MVC) software architecture pattern [13]. The MVC is implemented to enforce the separation of concerns into three parts:

- **Model**: The model contains the functionality of the application, e.g. construction of dependency graph and computation of impact.
- **View**: The view is responsible for displaying the computed results of the model, e.g. displaying a list of impacted classes. In other words, the view is responsible for the user interface.

[8] `https://github.com/`
[9] `https://travis-ci.org/`

- **Controller**: The controller is responsible for delegating work to the model and view based on events triggered by the user.

The implementation of the MVC is shown in figure 5.1.

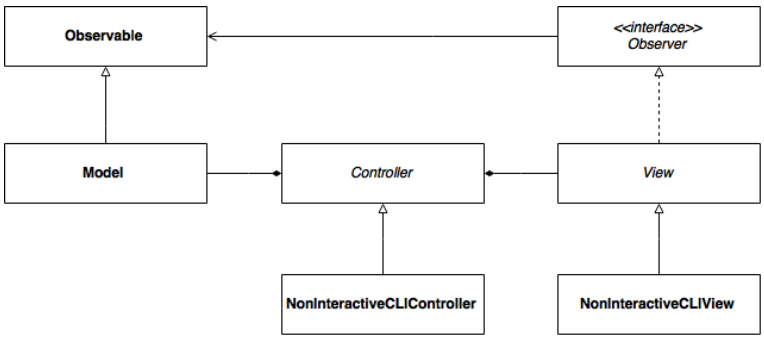

Figure 5.1: Class diagram of MVC pattern in Coda

The MVC pattern is currently implemented using the observer pattern. The observer pattern is used to decouple the view from the model, where the model is extending an `Observer` class and the view is implementing the `Observer` interface. Once the model is finished with computing the impact, the model notifies the view with the impact graph. Subsequently, the view displays the impacted classes based on the provided graph.

Currently, the view is displaying data via the command-line interface (CLI). Decoupling the view from the model allows different views for the same model. If, in the future, a developer would like to implement a view that displays the impacted classes via a visualized graph. The developer could extend the provided abstract class `View` and visualize the impact graph. Using the observer pattern ensures that changing the view does not have any impact on the model.

5.2.2 Graph

Constructing a dependency graph of the source code requires a graph data structure. Unfortunately, we have not found a graph library that was easy to use. A known graph library for Scala is "Graph for Scala"[10]. Due to the rich features of this library it had a steep learning curve and would have cost too much time to implement. Therefore, we decided to build a custom graph data structure in Scala. The graph required methods to add nodes, add edges and get incoming edges of a node. At first, a graph was created by having two fields: a set of edges and a set of nodes. However, asking the graph for the incoming edges of a node n resulted in a run time complexity of $O(m)$ where m is the amount of edges, because all the edges had to be traversed to find the incoming edges by comparing node n to endpoint v of an edge (u, v).

Consequently, it was desired to change the custom graph data structure. The goal was to improve the run time complexity. Redesigning the graph resulted in a hash map

[10] `http://www.scala-graph.org/`

with the hash of an edge as the key and the edge itself as the value. Saving the hash values of the edges in the node where the node is part of made it possible to look up the incoming edges in $O(1)$. Asking the graph for the incoming edges of a node could be accomplished as follows.

1. Graph asks the node for the hash values of the incoming edges.
2. Look up the hash values in the map of edges.
3. Return the set of edges.

Looking up a hash value of an edge in the hash map of edges has a run time complexity of $O(1)$. In worst case a node has an edge to every other node in the graph. This results in a run time complexity of $O(k)$ where k is the amount of nodes. $O(k)$ is not a huge improvement compared to $O(m)$ assuming that there are more edges than k. However, the worst case where a component has an edge to every component is often not present. Therefore, in practice a run time complexity of $O(k)$ is considered an improvement compared to $O(m)$.

The class diagram for the described graph data structure is depicted in figure 5.2. An edge and a node are part of one graph. A graph is composed of multiple nodes and multiple edges. An edge is always composed of one or two nodes.

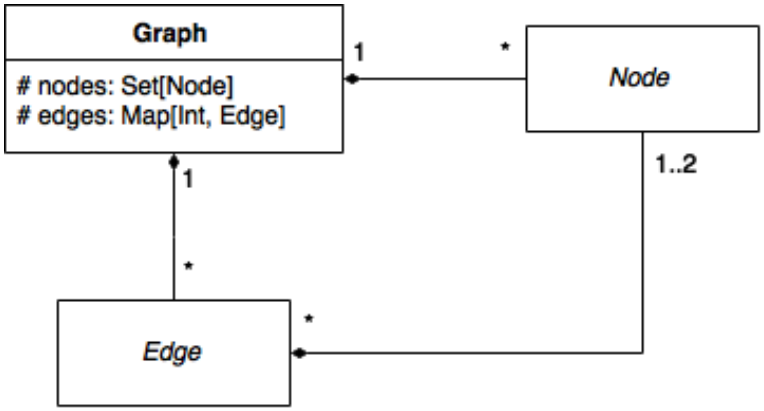

Figure 5.2: Class diagram of graph data structure.

5.2.3 Impact Analysis Algorithm

Change impact analysis can be done in various ways. Currently, a breadth-first traversal is used to calculate the minimal distance for each class node relative to nodes marked as changed. As explained in chapter 3 this algorithm determines how changed components impact unchanged components by looking at the minimal distance. This algorithm is implemented using the strategy pattern [28]. Extending the `ImpactAnalysisStrategy` trait allows usage of different impact analysis algorithms without breaking

existing code. Additionally, the strategy pattern allows change in impact analysis behavior during run-time if necessary. In figure 5.3 a class diagram of the implemented strategy pattern is shown.

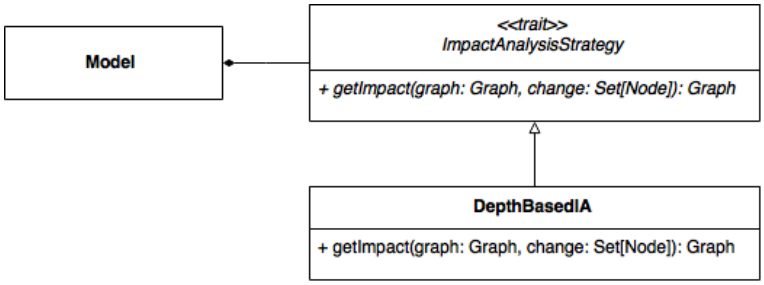

Figure 5.3: Class diagram of strategy pattern used for impact analysis algorithms

The Model in figure 5.3 computes the impact based on the changes by calling the `getImpact` method of the `ImpactAnalysisStrategy` concrete class. Implementing a new impact analysis algorithm requires that the trait `ImpactAnalysisStrategy` is extended. Subsequently, the concrete class should implement the abstract method `getImpact`.

5.2.4 Parsing Challenges

In order to derive a dependency graph and a change set, the application is required to be able to parse source code. The application must be able to extract class definitions and dependencies from the source code files.

Extracting the right information from source code can be performed by parsing compiled Java bytecode or by parsing plain Scala source code. As the Scala language is built to work with the Java Virtual Machine (JVM) and compiles to Java byte-code, a possible solution would be to look in this byte-code for class definitions and dependencies. However, Karim Ali et al. [2] have demonstrated that analysis of Java byte-code yields very imprecise results, because information about typing is lost during the compilation process from Scala to Java byte-code. In addition, Scala provides features that are difficult to analyze on a Java byte-code level because they have no Java equivalent, such as traits and abstract type members. Therefore, this project will attempt to parse plain Scala source code for dependency analysis.

One possible solution to extract class definitions and dependencies from plain Scala code would be to parse the source code using Regular Expressions (RegEx). RegEx is a technique that allows applications to match for patterns of characters occurring in text. The set of all possible patterns that can occur as valid Scala code is prohibitively large to write a parser for, requiring an effort equal to writing a full Scala parser. However, for this application, the RegEx parser would need to be able to parse class definitions and usage of classes. This might seem like a relatively small subset of patterns, when compared to all valid Scala patterns, but is still prohibitively large as

there are too many ways in which classes can be used syntactically. Moreover, parsing on a class level requires the application to be able to parse at the package level. Subsequently, parsing on lower levels such as method or statement level in the future would require being able to parse on all higher levels first. Because writing a RegEx parser would, therefore, be poorly extensible for future modification and too time consuming to implement, the RegEx solution was discarded. For an interesting read on the types of Scala code such a parser would have to be able to interpret, the reader is referred to Karim Ali et al. [2].

A discussion with the KeyLocker team resulted in a different approach: the Scala Toolbox[11], which provides the parsing ability of the Scala compiler. The Scala Toolbox parses any string containing Scala source code and provides an abstract syntax tree (AST) that can be traversed using a traverser provided by the Scala Compiler library. This traverser made the graph construction considerably easier compared to constructing a graph with RegEx.

Scala provides the ability to create packages just like Java does. One of the advantages of packaging is that the developer can create their own name-space. For example, if the developer would like to create a class named `Set`, he can create and use a class named `Set` in his own package without breaking the built-in `Set` of Scala as demonstrated in listing 5.1. Instantiating a class in another package requires the full name of the class, e.g. `foo.Bar` where `foo` is the package name and `Bar` is the class name. Using the full name of a class to instantiate seems troublesome and therefore imports are provided by Scala. `import foo.Bar` allows instantiation of class `Bar` without its full name. In addition, classes in the same package can use each other without imports.

Listing 5.1: Scala source code using packaging.

```
package foo {
  class A

   class B {
    val a = new A
    val s = new Set
  }

   class Set
}
```

Class `foo.B` can instantiate an instance of `foo.A` without importing class `foo.A`. Additionally, class `foo.B` instantiates an instance of `foo.Set`.

Constructing a dependency graph requires that full names of classes can be detected. Assumed is that most programmers do not use full names to instantiate an object, because using full names reduces the readability of the code. The obvious first

[11] `http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html`

step would be to parse the imports and to save them before a class is parsed. However, with this solution does not cover the possibility that classes in the same package can use each other without the need for an import statement. Therefore, this solution would produce an incorrect result. For example, in listing 5.1 class `foo.B` instantiates an instance of class `foo.A`. Constructing the graph of this source code would contain an edge (`foo.B`,`A`), which is incorrect. The correct graph should contain an edge (`foo.B`,`foo.A`). Moreover, Scala provides a default object, named `Predef`[12] including basic functionality, e.g. printing to console, commonly used types like `Set` and `String`, assertions, and implicit conversions. In listing 5.1 class `foo.B` instantiates a new `Set`. Detecting the full name of `Set` has proven to be difficult. In the AST of listing 5.1 (see Appendix B) there is no clue found for determining which of the two `Set` classes is instantiated. In further discussion this AST will be referred to as $AST1$. Compiling and running the code snippet would result in an instantiation of `foo.Set`.

To solve this problem, the AST of the source code is being traversed twice. During the first traversal all class definitions are detected and saved in a `Map`, where the key is the full name of a class and the value is its corresponding node object. The values of the resulting `Map` is equal to the node set of the dependency graph. In further discussion this `Map` will be referred to as $MapFN$. The goal of the second traversal is to detect the use of classes, creating the dependencies between class nodes, resulting in the map of edges of the dependency graph. To clarify these two steps, listing **??** will be analyzed.

The first traversal of the $AST1$ would result in the following map containing the full names.

```
Map(
  "foo.A" -> ClassNode("foo.A")
  "foo.B" -> ClassNode("foo.B")
  "foo.Set" -> ClassNode("foo.Set")
)
```

During the second traversal of $AST1$, the dependencies will be created. Moreover, in this step it will become clear why a map of full names ($MapFN$) is created. Walking through $AST1$ a stack is used to keep track of the current package level and another stack to keep track of the previous encountered class name. In $AST1$, the package stack would contain ”foo” during the traversal of the class definitions. Encountering the class definition of `foo.B` results in pushing ”foo.B” to the class stack. Once the instantiation of `Set` is encountered, the full name of `Set` is resolved to `foo.Set`. Subsequently, the full name is looked up in $MapFN$ and results in the creation of a dependency between `foo.B` and `foo.Set` with help of the class stack. The second traversal results in the following dependencies.

```
"foo.B" -> "foo.A"
"foo.B" -> "foo.Set"
```

[12] `http://www.scala-lang.org/api/current/index.html#scala.Predef\$`

5.2.5 User Interface

In this section the user interface will be discussed by exploring the possibilities of displaying the impact set. Additionally, the chosen solution will be motivated. Finally, displaying the impact will be discussed.

The user interface is an important aspect of the application and greatly affects the usability of Coda. As discussed in chapter 4, the impact set, displayed by Coda, should be comprehensible, fit in the development process of the developer, and is expected to be intuitive.

The first solution that emerged from a brainstorm session with the KeyLocker development team was to deploy Coda as an IDE plugin, e.g. integrating into Eclipse or IntelliJ. However, creating an IDE plugin would require knowledge of an existing IDE and would be too time consuming. Moreover, writing an IDE plugin would limit the usability of the apllication to only development environments making use of that specific IDE.

Furthermore, an analysis of the workflow at KeyLocker, see chapter 2, indicates that the developer is always using the command-line interface (CLI) to run a test suite with sbt. Additionally, the developer will use git from the CLI when they are finished with making changes to the source code and are ready to commit the changes to the code base. Therefore, based on integration into development process, a command-line interface application would fit most seamlessly into the development process of the KeyLocker team. Moreover, it does not make sense to analyze impact after each small change, but just before merging a feature into the master branch.

The last solution that emerged from the brainstorm session was to deploy Coda as a web application. The advantage of a web application is that many visualization libraries exist. This would enable a more graphical visualization of the impact and could greatly improve the usability of the application. However, the KeyLocker team suggested that working on the CLI would layout a fundamental core for the web application. For the sake of staying on schedule, this option was abandoned, but kept as a useful addition for future work. The KeyLocker team added that, should the Coda CLI be able to provide useful results, they would like to build their own interface to integrate in their own respective development environments.

We have, therefore, decided that a CLI application would satisfy the developers and will layout a fundamental core for future extension. Another important aspect of Coda is how impact is displayed. The CLI application will display the impacted classes resulting from impact analysis. Various ways exist to display impact. One potential way is to display the shortest distance for each impacted class to a changed class. Because this representation requires knowledge of the concept of distance analysis on the part of the user, the impact is represented as a score on a relative scale. That way, no actual knowledge about impact analysis is necessary for using Coda, merely reading that one component is impacted more heavily than another suffices for the user. Moreover, using numbers can falsely imply a certain precision of an absolute impact, which is not the case. Dependency analysis merely suggests ways in which components could be

impacted, but has no actual clue about whether this is true or not. All impact is relative, one component being more likely to be impacted than another.

Now that the design decisions during development have been discussed in detail, the consequences of those decisions shall be examined in the next section where the results of a maintainability review will be discussed.

5.3 SIG Feedback

In context of the bachelor project, the source code of Coda had to be submitted to the Software Improvement Group (SIG) twice. SIG is a Dutch company that reviews source code from a multitude of companies, governmental organizations, and student submissions from the TU Delft. For the student submission, they mainly review the maintainability of the software based on a model they developed, an example of which is shown in figure 5.4[13]. Using metrics to measure volume, code duplication, unit complexity, unit size, unit interfacing, module coupling, component balance, and component independance, the SIG model computes a maintainability score. This section will discuss the SIG feedback and subsequently document our response.

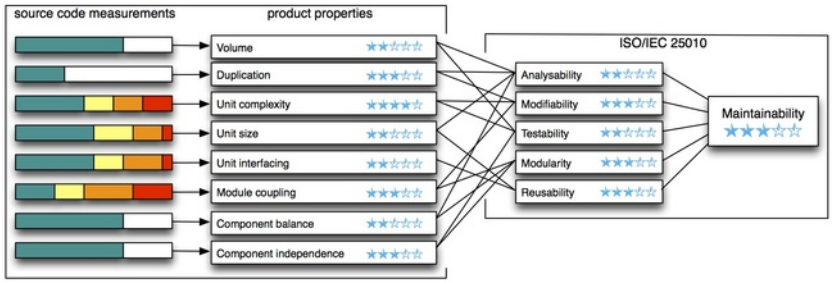

Figure 5.4: SIG maintainability model of an example project.

5.3.1 First Evaluation

The first quality assurance done by SIG resulted in the following feedback in Dutch.

[13] https://www.sig.eu/nl/over-sig/sig-research/sig-model-maintainability/

> ”[Analyse]
>
> De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere deelscore voor Duplication.
>
> Voor Duplicatie wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren.
>
> Bij jullie is de hoeveelheid gedupliceerde code beperkt, maar omdat jullie qua score al vrij hoog zitten is dit alsnog een verbeterpunt. Je zou bijvoorbeeld kunnen beginnen met het duplicaat tussen ClassBasedTS.scala:traverse-PackageDef() en Namespace.scala:traversePackageDef(), die methode is namelijk tussen de twee bestanden gekopieerd.
>
> Daarnaast is het goed om te zien dat jullie test-driven werken: de hoeveelheid testcode is zelfs hoger dan de hoeveelheid productiecode. Hopelijk lukt het jullie om dit in stand te houden tijdens het vervolg van het project.
>
> Kortom, zowel qua onderhoudbaarheid als qua testcode scoren jullie boven-gemiddeld, en we hopen op een soortgelijke score in het vervolg van het ontwikkeltraject.”

The feedback indicates that code quality of Coda is above average, obtaining a near perfect score. The Coda project scored 4 stars out of 5, nearly missing the 5-star mark because of some small code duplication between two components. Consequently, SIG expects the final submission to acquire the same maintenance rating, if not better.

5.3.2 Response

According to the feedback that SIG provided, we have reviewed the source code of Coda by mainly looking at code duplication. However, as a matter of fact, inheritance should be used to create class hierarchies that makes sense, not to avoid code duplication [4]. Implementing a class hierarchy here did not make sense, especially simply to remove a small code duplication of a few lines of code. For this reason and because said duplication was actually tangential in functionality, we elected to keep some small duplication in place and only remove where necessary. Furthermore, we have implemented the Model-View-Controller architecture pattern to improve maintainability in the future, e.g. creating different views for the same core application.

At the time of writing this report we have not yet done the final submission to SIG. Therefore, the final feedback of SIG is not included in this report, but will likely be presented during the final presentation.

5.3.3 Second Evaluation

The final code submission resulted in the following feedback of SIG.

> ”[Hermeting]
>
> In de tweede upload zien we dat zowel de omvang van het systeem als de score voor onderhoudbaarheid is gestegen. Het systeem scoort nog steeds 4 sterren, wat betekend dat het bovengemiddeld onderhoudbaar is.
>
> Het is echt goed om te zien dat jullie de score voor Duplicatie een stuk hebben verbeterd. Deze is nu ook bovengemiddeld. Verder zijn er geen opmerkingen in de code te vinden. Complimenten daarvoor.
>
> Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.”

The feedback indicates that the code quality has improved. SIG appreciates the fact that we took their feedback into consideration, resulting in a better score for code duplication.

6 Conclusion

In this chapter an overview of the product, named Coda, will be given by looking at its functionality and its user interface. Furthermore, the product will be compared with the functional and non-functional requirements discussed in chapter 4.

6.1 Product

Coda is an analysis tool that performs change impact analysis on Scala projects. A prerequisite for using Coda on a Scala project is that the project is under git version control, because the application uses git to derive the change set. Currently, Coda should be used before merging changes into the production code. Running Coda in the root directory of a project without any arguments will display the following output below. Coda will output a summary of impact analysis followed by a list of classes in descending order based on the likelihood of impact. Naturally, when no change is applied to the analyzed project no impact classes will be shown, as can be seen in listing 6.1.

Listing 6.1: Coda analysis for an unchanged project

```
$ coda

Coda 1.0

Analyzing project from ~/coda/.

Total classes:   65
Changed classes: 0
Impacted classes: 0
```

In listing 6.2 a change is applied to the code base of the Coda project itself. Note that this time an argument, a project directory, is provided to Coda. Using Coda, it can analyze the change to its own code and compute the impact. Coda displays a list of changed nodes, impacted nodes, and an indication of the likelihood of impact. Based on the impact indication the developer can walk through the list ensuring that missing test cases are added and/or impacted classes are fixed.

Listing 6.2: Coda analysis for a changed project

```
$ coda ~/coda

Coda 1.0

Analyzing project from ~/coda/.

Total classes:   65
Changed classes: 1
Impacted classes: 13

IMPACT     CLASS
CHANGED    eu.keylocker.model.graph.Graph
[#####]    eu.keylocker.model.ia.DepthBasedIA
[#####]    eu.keylocker.model.Model
[#####]    eu.keylocker.model.ast.ClassBasedTS
[#####]    eu.keylocker.view.NonInteractiveCLIView
[#####]    eu.keylocker.model.ia.ImpactAnalysisStrategy
[####-]    eu.keylocker.model.graph.GraphSpec
[####-]    eu.keylocker.Application
[####-]    eu.keylocker.model.ast.TraverseStrategySpec
[####-]    eu.keylocker.controller.NonInteractiveCLIController
[####-]    eu.keylocker.model.ia.DepthBasedIASpec
[####-]    eu.keylocker.controller.Controller
[####-]    eu.keylocker.model.ast.ClassBasedTSSpec
[###--]    eu.keylocker.view.View
```

Furthermore, the developer could use the default `--help` flag to read more information about Coda and its use cases.

6.2 Product Review

This section will review whether the requirements formulated at the beginning of the project have been met. First, an overview of all functional requirements will be provided and whether they have been met or not. Where necessary, the overview will be followed by an explanation. Then, a similar overview of the non-functional requirements shall be provided, along with its respective explanations. Finally, it is concluded that Coda can help to solve the knowledge sharing and testing problems it set out to solve.

6.2.1 Functional Requirements

The functional requirements are evaluated in tables 6.1, 6.2, 6.3, and 6.4 below, followed by an explanation where necessary.

Table 6.1: The application must . . .

Nr.	Requirement	Requirement met
1	. . . be able to parse Scala source code.	Yes
2	. . . be able to detect class definitions.	Yes
3	. . . be able to detect full names of classes.	Yes
4	. . . be able to detect parent-child relations between classes.	Yes
5	. . . be able to compute the impact of a change on class level.	Yes
6	. . . be able to create a dependency graph of a Scala project on class level.	Yes
7	. . . be able to detect the changed set of classes when a change is introduced in source code.	Yes
8	. . . be able to output impacted classes to the user, prioritized by relevance.	Yes
9	. . . be executable from the command line.	Yes

Table 6.2: The application should . . .

Nr.	Requirement	Requirement met
10	. . . be able to parse most commonly used types of valid Scala source code and combinations thereof.	Yes

Table 6.3: The application could . . .

Nr.	Requirement	Requirement met
11	. . . visualize the impact graph to the user.	No
12	. . . parse all possible combinations of valid Scala source code.	No
13	. . . persist the results in a log or a database.	No

Table 6.4: The application won't . . .

Nr.	Requirement	Requirement met
14	. . . automatically select and execute test cases for impacted classes.	No

Requirement 11 was not met due to the time constraint of the bachelor project. However, the Model-View-Controller architecture pattern is implemented in Coda as discussed in chapter 5, making it simple to extend the application with different views in the future.

Requirement 12 was not met, because not all possible class definitions are parsed. To verify this statement, two existing open-source Scala projects, scct[14] and browse[15], and the Coda project have been analyzed.

Table 6.5: Project Class Count

Project	Manual Class Count	Coda Class Count
scct	62	64
browse	72	62
Coda	55	53

[14] `https://github.com/mtkopone/scct` - a Scala code coverage tool.

[15] `https://github.com/harrah/browse` - a tool that generates browsable HTML pages from source code.

In table 6.5 the amount of encountered classes are shown for each Scala project. The amount of class definitions were counted manually and with Coda. Based on the results it can be concluded that most class definitions are detected, but there is still plenty of room for improvement.

Requirement 13 was not met, results of Coda are not persisted in a file or database. Once printed to the terminal and the Coda application exits, the results are only left on screen. However, an impact analysis by Coda takes less than 5 seconds for most projects, making executing the analysis multiple times fairly simple. Therefore, storing the results had a low priority and has not been implemented.

Requirement 14 was not met. As explained in chapter 4, KeyLocker deemed that automatic regression testing had little to no added value for the company. Therefore, this feature has not been implemented.

6.2.2 Non-functional Requirements

The non-functional requirements are evaluated in tables 6.6, 6.7, and 6.8 below, followed by an explanation where necessary.

Table 6.6: Utility

Nr.	Requirement	Requirement met
1	The application should produce results within 5 seconds.	Yes
2	The application should have a precision of at least 76%.	No
3	The application should have a recall of at least 94%.	Yes
4	The application should consistently produce the same results on the same input.	Yes
5	Users should rate accuracy at least average.	No
6	Users should rate usefulness of the results at least average.	Yes
7	Users should indicate that they would use the application in the future at least average.	Yes

Table 6.7: Usability

Nr.	Requirement	Requirement met
8	Users should rate result comprehension at least average.	Yes
9	Users should rate the seamlessness of the integration into the development process at least average.	Yes
10	Users should rate the intuitiveness of the application at least average.	No

Table 6.8: Maintainability

Nr.	Requirement	Requirement met
11	Statement coverage should be at least 85%.	
12	Methods should not consist of more than 100 lines of code and hardly ever exceed 20 lines of code [22].	Yes
13	Each class and method should be documented and contain ScalaDoc comments.	Yes
14	Each class should contain as few inline comments as possible.	Yes
15	Components must have a higher cohesion than coupling.	Yes
16	The medium to display results should be isolated from the results themselves.	Yes
17	Different algorithms for computing impact must be easily interchangeable.	Yes
18	Different types of input must be easily interchangeable.	Yes

As described in chapter 4, the precision and recall of Coda's analysis is computed using the formulas below. To determine the precision and recall of Coda, the true positives, false positives, true negatives, and false negatives found in an open-source Scala project have been used in a confusion matrix. The numbers shown in table 6.9 were obtained by manually counting the actual positives and negatives, and comparing these numbers with the predicted positives and negatives by Coda. The true positives are determined with $PP \cap AP$. True negatives is not necessary for calculating recall and precision, therefore it is not determined. False positives are determined with $PP \cap AN$. False negatives are determined with $PN \cap AP$. These numbers have been obtained by analyzing a git commit[16] in Coda, where name changes were made to GraphUtility.

Table 6.9: Confusion Matrix of Coda's Analysis

	Actual Positive (AP)	**Actual Negative (AN)**
Predicted Positive (PP)	3	2
Predicted Negative (PN)	0	-

Based on the confusion matrix, we can determine precision and recall as follows.

$$Precision = \frac{TP}{TP + FP} = \frac{3}{3 + 2} = 60\%$$

$$Recall = \frac{TP}{TP + FN} = \frac{3}{3} = 100\%$$

As the recall is 100%, requirement 3 has been met. The precision, however, is below 76%. The reason for this is that Coda, currently, does not analyze changes on statement level. In GraphUtility some value names were changed. These values are not used in every class that is using the GraphUtility. However, Coda does perform analysis on class level and did not catch this, leading it to predict that every class using the GraphUtility was impacted.

For this project, code quality is measured by comparing coupling with cohesion. Good software design is achieved by minimizing coupling and maximizing cohesiveness [10]. In table 6.10 coupling and cohesion of every class in Coda is shown. Note that only method calls between classes of Coda are counted, i.e. method calls to external classes are not counted. In addition, it is important to note that "the number of method invoked" is counted, which is not to be confused with "the number of method invocations" [19].

[16] Commit 9536e9cb8443b305cf87a1a5c3da28cd5134d82e

Table 6.10: Coupling & Cohesion for the Coda Project

Class	**Coupling**	**Cohesion**
Controller	1	1
NonInteractiveCLIController	3	0
ASTFactory	1	1
ClassBasedTS	6	16
Namespace	5	10
TraverseStrategy	0	0
Edge	2	4
Dependency	0	0
Inheritance	0	0
AbstractClassNode	0	0
ClassNode	0	0
Node	0	4
ObjectNode	0	0
TraitNode	0	0
Graph	12	8
GraphFactory	1	0
DepthBasedIA	5	2
ImpactAnalysisStrategy	0	0
Correctable	0	2
DiffParser	3	4
ScalaFile	2	3
ScalaSourceCode	1	0
SourceComponent	0	0
SourceComposite	2	1
Model	9	2
NonInteractiveCLIView	4	4
View	0	3

Based on table 6.10 Coda is meets the coupling requirement 15 well, because many classes have a higher cohesion than coupling. However, there is still room for improvement, because some classes that combine functionality have a higher coupling than cohesion.

Requirements 5, 6, 7, 8, 9, and 10 are measured with a survey, see Appendix C. The developers were asked to answer a survey by giving a rating from 1 to 5 for each aspect shown in table 6.11.

Table 6.11: The average of the results of the developer survey, 1 being very bad and 5 being very good.

Aspect	Average Rating
Accuracy	2.5
Usefulness	4
Comprehension	3.5
Integration	3
Intuitiveness	2.5
Use in future	3.5

The results show that intuitiveness (2.5) and accuracy (2.5) score below average. The intuitiveness was rated below average because an ambiguous exception message was shown to the user, during a developer's test. Thrown exceptions were caught and handled, but the error messages were insufficiently informative. Furthermore, a developer felt that the accuracy of the program was low because the test project we provided did not allow him to introduce changes that resembled real-life code changes closely enough, thus resulting in inaccurate suggestions by Coda.

Finally, as the highest priority functional requirements and most of the non-functional requirements have been met, we conclude that Coda solves the two knowledge sharing and testing problems it set our to solve.

We can conclude that impact analysis with the use of Coda solves some of the knowledge sharing issues at KeyLocker. Developers acquire knowledge by using Coda without having to ask other developers, maintaining productivity. Although, we demonstrated that Coda could benefit from improvement and is far from perfect, it provides insight in dependencies between classes, making it easier for developers to understand the code base. Moreover, based on the confusion matrix, Coda is able to obtain a 100% recall.

Furthermore, we can conclude that impact analysis can improve testing at KeyLocker. Writing end-to-end tests and integration tests have proven to be difficult. Based on the dependency analysis that Coda provides, developers can acquire insight into the inter-dependencies between components, making it easier to write tests. Furthermore, the pesticide paradox principle is enforced. As developers are confronted with which classes are impacted, they are moved to maintain and extend existing test suites to keep up with the introduced changes and improve test quality.

7 Discussion

In this chapter, the results and process of the project will be reviewed. The following sections will critically examine the resulting product and process, respectively. The product section will discuss where functionality partially works, is faulty, or is missing. Furthermore, the process section will discuss some of the challenges and pitfalls of our development process.

7.1 Product

7.1.1 Value to the Company

When examining the resulting product, the most useful functionality Coda provides is its ability to parse Scala source code into a dependency graph. At the moment, this graph is only used for impact analysis. However, analyzing the dependency graph can tell a developer way more about the Scala project besides the impact of a change: the dependency graph can be used to identify poorly engineered code using coupling metrics, or node in- and out-degrees; the dependency graph can be used to identify anti-patterns, such as God objects [27] to suggest ways of improving the code base using many types of algorithms (e.g. an interesting approach would be to use algorithms like Google's PageRank algorithm [9] to identify most important components in a project); combining coverage information with dependency analysis might offer insight for regression testing and where impact remains untested. Moreover, simply visualizing the dependency graph could potentially yield great insight into a code base for a developer. In short, the dependency graph analysis can prove a lot more useful than only for impact analysis.

The point under discussion is whether impact analysis truly provides the most value for the company. Knowing what we know now, it would have been possible to part from impact analysis and pursue one of the different type of analysis suggested above. However, impact analysis clearly solves the problem defined at the start of the project. Although the freedom we enjoyed during the project might have allowed us to change this problem definition, the problem we set out to solve was broadly agreed upon with the development team and our supervisors. Moreover, although dependency analysis for purposes such as identifying poor software engineering might have more commercial value to the company, our purpose was to solve some of the problems experienced in the company's software development process. Poor software engineering was not among those problems. Therefore, we feel confident that the choice for change impact analysis was the right one.

7.1.2 Unable to Separate Class Changes in File

At the moment, Coda is able to detect which files have been changed in a Scala project. Coda is then able to read the changed files, parse the classes inside those files and mark those classes as changed. Usually, files contain only one class and this approach results in a fairly high accuracy. However, when multiple classes are defined in the same file, Coda is not able to distinguish which classes inside the file were touched and marks them all as changed. This causes a significant drop in accuracy, suggesting many false positives of classes that were, in fact, left intact. Due to time constraints we were unable to develop a mechanism that can successfully distinguish changed classes from unchanged classes, even if they are contained in the same file. This is definitely an issue up for improvement in future work.

7.1.3 Obscure Scala Code

Although Coda is able to parse most forms of valid Scala syntax, it is limited by the Toolbox[17] used to parse Scala code. This Toolbox will throw compile errors for certain syntax patterns that do, in fact, compile.

For instance, the Toolbox does not parse the example pattern in listing 7.1 correctly.

Listing 7.1: Scala source code using a rare package declaration.

```
package a
package b
```

Which is equivalent to writing `package a.b` followed by `import a.*`. This provides access to all of the contents of `a` in in the subpackage `b`. Although this is valid Scala code and compiles successfully, the Toolbox parses this incorrectly. Parsing these occurrences manually would have been very time consuming. However, luckily and not surprisingly, this obscure use of package declarations almost never occurs in real Scala projects in our experience.

Another syntax pattern that is not matched for currently are abstract type members. This is one of the difficult to parse patterns in Scala, as its use is not straightforward. Abstract type members are components whose type is not precisely known and whose type can be overridden in subclasses. Take, for instance, the example in listing 7.2.

[17] `http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html`

Listing 7.2: Scala source code using abstract type members.

```
abstract class SomeAbstractClass {
   type T
   val element: T
}

abstract class SomeSubClass extends SomeAbstractClass {
   type U
   type T <: Seq[U]
   def length = element.length

}
```

In this example, we only know that each instance that extends `SomeAbstractClass` has a type member `T`, but it is up to the implementation of the subclass what this type `T` looks like. As can be seen in the definition of `SomeSubClass`, the type of `T` is tightened to be a subtype of a `Seq` (a sequence) of some other type `U`. This means that the method `length()` can be called on the element, as this is a method of `Seq`. As the type of abstract type members are by nature undefined, it is not difficult to see that the dependencies can become very complex very fast. Altough the Toolbax can parse these types, we elected not to interpret these syntax patterns in order to stay on schedule. Interpreting these patterns could go a long way in improving the recall of Coda's dependency analysis in the future.

7.2 Process

7.2.1 Case Study

Immediately after performing our initial analysis of the company's development approach and a discussion with our TU Delft supervisor, we alerted our company supervisor and the rest of the KeyLocker team that we were developing code fundamentally wrong way (in some aspects), as described in chapter 2. However, in chapter 2 there was no room to discuss what actually went well for the team and how well the company was actually doing. Moreover, after discussing the need to alter course and how to solve the problems highlighted by the case study, everyone agreed and measures were taken immediately to improve on the highlighted problems. We are happy to see that, over the course of the project, the development process at KeyLocker (and our own), has greatly improved.

7.2.2 'Full-Stack First'

At the beginning of the project it was determined to implement the full stack of the application first, before starting to perfect every component. This was intended to be able

to quickly validate the utility and usability of the application with the KeyLocker team. However, it was not until six weeks into development that a first full stack application could be presented. Too much time was spent on perfecting the dependency analysis, which took by far the most time to implement correctly. As described before, especially interpreting the multitude of different valid forms of Scala code correctly proved very time consuming to both implement and test - most tests have been written to test exactly that. This caused us to get bogged down on the details of dependency analysis, postponing the delivery of a full stack application and thus postponing the validation of the application. Had we followed this agreed procedure more closely, perhaps different decisions could have been made.

7.2.3 Redesign and Refactor Phases

During the project we did notice a growing buildup of technical debt. The application would not be broken, it was just that we were not satisfied with the elegance and maintainability of the solution. We would then sometimes dedicate a few hours to solve this. However, over the course of the project, technical debt did accumulate and therefore one entire sprint was dedicated to redesigning the some parts of the architecture, before submitting to SIG. This ensured that Coda was well maintainable and consequently sped up our own development process. The good score received from SIG was a confirmation of the quality and served as justification for the redesigning phase. However, the fact that such a dedicated week for redesigning the architecture was needed means something went wrong during the development process. Although caught early enough and dealt with swiftly, our development process did not prevent the buildup of technical debt. Nevertheless, we managed to stay on schedule and keep up the high quality design of Coda.

All in all, a quality product has been developed in a responsible way and challenges have been dealt with efficiently. After having considered what could have gone better, the next chapter will discuss recommendations for future work.

8 Recommendations

In this chapter, recommendations for future work will be discussed. First, some components of the application that could benefit from improvement will be discussed, followed by a description of some areas where useful extensions can be made to expand the functionality of the application.

8.1 Areas for Improvement

Currently, the deployment mechanism of Coda is simply compiling the source code and distributing this to developers who want to use the application. This is far from developer friendly. By far preferable would be to deliver a packaged application with a simple install mechanism that requires very little action from the user. At the moment, an install script is provided to create an alias for Unix based systems, but this is not up to the standard of a production deployment mechanism.

Additionally, Coda is currently unable to distinguish changes to classes when located in the same file, as discussed earlier. Coda currently uses Git commands to find these changes and subsequently parses the changed files suggested by Git. However, if Coda were to parse the diff files provided by git, showing the difference between versions for each line of a changed file, a more fine grained selection of the change set can be performed. Not only could the change set be limited to the changed classes, the change to that class could also be interpreted. This restriction and the additional information could greatly improve the precision of the solution.

Furthermore, the command-line application currently supports very little customization for the developer. The application can be carried out in the default way, analyzing a directory of the user's choosing. However, using command-line arguments, the CLI could offer to execute the analysis tweaked to the user's needs. Some examples include providing a stopping distance for the impact analysis algorithm to restrict runtime, instructing the application to provide more verbose output with e.g. the graph outputted to the terminal, and the results of the analysis could be outputted to a file or database to be stored for later quality assurance. Providing more options to tweak the experience to the developers needs can greatly improve the utility of the application.

Finally, the current parsing engine that uses the Toolbox described earlier suffers from many limitations. As mentioned before, the Toolbox is often unable to parse Scala code that does compile. Therefore, to be able to analyze more Scala projects and improve the quality of the analysis, a different parsing engine could provide great improvement. A possible solution is to develop a custom plugin for the Scala compiler. Although this was too time consuming for the scope of the project, it would provide

Coda with the full parsing power of the Scala compiler. This would eliminate the need for altering code before it can be compiled by the Toolbox, as is currently the case to fix some bugs in the Toolbox. Moreover, this would enable full typechecking, eliminating the need for the somewhat slow process of name resolution currently used.

8.2 Areas for Extension

Currently, Coda will analyze Scala projects on a class level, finding dependencies between classes that use each other or extend/implement each other. However, it does not distinguish between what methods are called. If a class is changed, Coda will suggest other classes that need to be fully checked. If Coda were to analyze on a method level, it could construct a call graph and pinpoint what methods could be impacted, instead of classes. Going even further, if Scala were to analyze on a statement level, it could identify the types of changes made to files and perhaps pinpoint the impacted statements if possible. We recommend expanding the current functionality to the method level for improving the accuracy and limiting the suggested lines that a developer will need to check. Such an approach would need a new implementation of a `TraverseStrategy` and require little changes elsewhere.

Additionally, Coda's functionality need not be limited to mereley parsing Scala code. The current coda project could be expanded to include many other object oriented programming languages, e.g. Java, requiring only minimal changes to the analysis components. Only the input and parsing components would have to be extended or substituted to enable Coda to analyze these other projects.

The same goes for version control systems: currently, Coda anlyzes git repositories, but this can be easily expanded to include other version control systems such as Apache Subversion[18]. Again, this would require substituting a few isolated components and some minor changes elsewhere.

Furthermore, different impact analysis algorithms could be employed to offer a broaded range of options to developers and perhaps increased result quality. This would, as most recommendations for extension, require implementing a new `ImpactAnalysisStrategy` and minor changes elsewhere.

Finally, the current command-line based user interface could be extended to an interactive user interface or, even better, extended to a generated web-based user interface that can run in the browser of the user. This would enable many more options for visualizing output to the user and improving the user's understanding of the code base and Coda's analysis.

[18] `https://subversion.apache.org/`

References

[1] M. Acharya and B. Robinson, “Practical change impact analysis based on static program slicing for industrial software systems”, in *Proceedings of the 33rd international conference on software engineering*, ACM, 2011, pp. 746–755.

[2] K. Ali, M. Rapoport, O. Lhoták, J. Dolby, and F. Tip, “Constructing call graphs of scala programs”, in *ECOOP 2014–Object-Oriented Programming*, Springer, 2014, pp. 54–79.

[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[4] J. M. Armstrong and R. J. Mitchell, “Uses and abuses of inheritance”, *Software Engineering Journal*, vol. 9, no. 1, pp. 19–26, 1994.

[5] L. Badri, M. Badri, and D. St-Yves, “Supporting predictive change impact analysis: A control call graph based technique”, in *Software Engineering Conference, 2005. APSEC’05. 12th Asia-Pacific*, IEEE, 2005, 9–pp.

[6] F. S. de Boer, M. M. Bonsangue, L. Groenewegen, A. Stam, L van der Torre, *et al.*, “Change impact analysis of enterprise architectures”, in *Information Reuse and Integration, Conf, 2005. IRI-2005 IEEE International Conference on.*, IEEE, 2005, pp. 177–181.

[7] S. A. Bohner, “Software change impact analysis”, 1996.

[8] B. Breech, M. Tegtmeyer, and L. Pollock, “Integrating influence mechanisms into impact analysis for increased precision”, in *Software Maintenance, 2006. ICSM’06. 22nd IEEE International Conference on*, IEEE, 2006, pp. 55–65.

[9] S. Brin and L. Page, “Reprint of: The anatomy of a large-scale hypertextual web search engine”, *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[10] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design”, 1994.

[11] J. Davis and M. Goadrich, “The relationship between precision-recall and roc curves”, 2006.

[12] M. Fowler, *Testcoverage*, `http://martinfowler.com/bliki/TestCoverage.html`.

[13] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head first design patterns*. ” O’Reilly Media, Inc.”, 2004.

[14] D. Graham, E. Van Veenendaal, and I. Evans, *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[15] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damásio, "On the precision and accuracy of impact analysis techniques", 2008.

[16] L. Huang and Y.-T. Song, "Dynamic impact analysis using execution profile tracing", in *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, IEEE, 2006, pp. 237–244.

[17] M.-A. Jashki, R. Zafarani, and E. Bagheri, "Towards a more efficient static software change impact analysis method", in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ACM, 2008, pp. 84–90.

[18] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis", in *Software Engineering, 2003. Proceedings. 25th International Conference on*, IEEE, 2003, pp. 308–318.

[19] J. K. Lee, S. J. Jung, S. D. Kim, W. H. Jang, and D. H. Ham, "Component identification method with coupling and cohesion", 2001.

[20] S. Lehnert, "A review of software change impact analysis", *Ilmenau University of Technology, Tech. Rep*, 2011.

[21] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques", *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[22] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. 2008.

[23] A. McNair, D. M. German, and J. Weber-Jahnke, "Visualizing software architecture evolution using change-sets", in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, IEEE, 2007, pp. 130–139.

[24] V. Rajlich, "A model for change propagation based on graph rewriting", in *Software Maintenance, 1997. Proceedings., International Conference on*, IEEE, 1997, pp. 84–91.

[25] M. Sherriff and L. Williams, "Empirical software change impact analysis using singular value decomposition", in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, IEEE, 2008, pp. 268–277.

[26] J. Sutherland and K. Schwaber, *The scrum team*, `http://www.scrumguides.org/scrum-guide.html`.

[27] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes", in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, IEEE, 2009, pp. 145–154.

[28] P. Wolfgang, *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.

Appendix A: Infosheet

Project Title: Coda: A Change Impact Analysis Tool for Scala
Client: KeyLocker BV
Date of the final presentation: 26-08-2015

Description
KeyLocker, a start-up developing cryptographic products, has requested a review of their own software development process. We were tasked with this review as well as developing a software application that could assist in the development process, remedying knowledge sharing and test maintenance using change impact analysis.

Developing this application required research in the field of change impact analysis, the identification of potential consequences of a change to components in software.

Throughout the project we worked in sprints of one week, loosely following the Scrum methodology. Weekly meetings were held with our TU Delft coach and KeyLocker's supervisor separately to discuss our progress and plan the next sprint. Software design was discussed amongst ourselves, sometimes presenting it to the KeyLocker development team or our TU Delft coach if we ran into issues.

The bachelor project resulted in a command-line application, Coda, that performs impact analysis on Scala projects under Git version control and outputs the impact of changes to the user.

The Coda project laid out a fundamental core, which is ready to be picked up by the KeyLocker development team for integration and further development.

Team Roles
Marc Mackenbach - marcmackenbach@gmail.com
Interests: Cryptography, Big Data, Web Services, User Interaction.
Contributions: I/O, Detecting the change set, Impact Analysis, MVC, User Interface.

Aaron Ang - awz.ang@gmail.com
Interests: Software Engineering, Web, User Interface, User Experience
Contributions: Parsing Scala, Dependency graph construction

All team members contributed to preparing the report, the final presentation, developing and testing Coda.

Client: Charlotte Goedmakers, KeyLocker BV
TU Coach: Annibale Panichella, Software Engineering Research Group, TU Delft
Contact: charlotte.goedmakers@keylocker.eu
The final report for this project can be found at: `http://repository.tudelft.nl`

Appendix B: Abstract Syntax Tree Example

Listing 8.1: An example of an AST, parsed from Scala source code.

```
PackageDef(
  Ident(TermName("foo")),
  List(
    ClassDef(
      Modifiers(),
      TypeName("A"),
      List(),
      Template(
        List(Select(Ident(scala), TypeName("AnyRef"))),
        noSelfType,
        List(DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List()),
            TypeTree(), Block(List(pendingSuperCall),
            Literal(Constant(()))))))),
    ClassDef(
      Modifiers(),
      TypeName("B"),
      List(),
      Template(
        List(Select(Ident(scala), TypeName("AnyRef"))),
        noSelfType,
        List(
          DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List()),
              TypeTree(), Block(List(pendingSuperCall), Literal(Constant(())))),
          ValDef(Modifiers(), TermName("a"), TypeTree(),
              Apply(Select(New(Ident(TypeName("A"))), termNames.CONSTRUCTOR),
              List())),
          ValDef(Modifiers(), TermName("s"), TypeTree(),
              Apply(Select(New(Ident(TypeName("Set"))), termNames.CONSTRUCTOR),
              List()))))),
    ClassDef(
      Modifiers(),
      TypeName("Set"),
      List(),
      Template(
```

```
        List(Select(Ident(scala), TypeName("AnyRef"))),
        noSelfType,
        List(DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List()),
            TypeTree(), Block(List(pendingSuperCall),
            Literal(Constant(()))))))))))

}
```

Appendix C: Developer Survey

The developers at KeyLocker were provided with an example project and asked to make changes and use Coda to evaluate the impact of their changes. They were then asked to provide answers to the following survey:

The following questions must be answered from 1 to 5.

1. - very bad
2. - bad
3. - average
4. - good
5. - very good

Questions:

1. How well does Coda integrate into your workflow?
2. How familiar does the use of Coda feel to you? How intuitive do you rate use of Coda?
3. How understandable do you rate the output of Coda?
4. How do you rate the accuracy of the results?
5. How do you rate the usefulness of the results?
6. Would you use Coda when working on Scala projects?

Appendix D: Assignment

In this appendix the original assignment description, found at `bepsys.herokuapp.com`, is shown. Note that the assignment has been fine-tuned during the project.

Designing Development for Growth

The problem to be solved

KeyLocker started with a two-man development team. Both developers were second year Computer Science students from Technical University Delft and relatively inexperienced in both professional software development and the technologies used to build the product. It is safe to say that the learning curve was steep.

Going along, the development team tried to apply some of the best practices of software development, such as agile and test driven development. However, applying these practices correctly and consistently to a process subject to change as experience was gained proved a difficult task. Eventually the dev-team grew to four people, increasing overhead and a need for a systemic on- and off-boarding policy arose. Among other things, documentation of both source code and learned principles, theory, and practice formed a vital part of this on-boarding process. An important aspect of the knowledge sharing is the reasoning involved in development, documenting the what and the why. Currently, the team is being scaled up even further and KeyLocker is in the midst of a process of fine tuning workflow scalability.

Many theories and best practices exist for development in teams of 4-8 people (such as scrum), but little is written for building a software development team from scratch, starting from zero people with zero experience. KeyLocker wants to know where in its history better decisions could have been made and what practices worked well. More importantly, KeyLocker wants to gain insight in how it can do better in the future. Therefore, after evaluating the problem areas of creating a scalable workforce in both experience and number and formulating a strategy accordingly, KeyLocker wants to tackle at least one of the problem areas by developing a tool that will assist a small dev team in the process of developing robust software for now and for the future.

Assignment

Your assignment is use (the history and practice of) KeyLocker as a case to research what the best practices are to scale from zero to something and how best to design

a development process for scalability. Identify the challenges of scaling tiny development teams to regular sized development teams and formulate a development strategy that scales well with a growing company. Build a tool that will assist the early stage company in the development of software and that will tackle one of the signalled challenges.

Usage of your tool by a development team should result in easier growth of the dev team. You are free to determine the problem the tool should solve and what form your solution should take, as long as it is centred on improving the scalability of the dev team. There are no language requirements, as long as it can be used on workstations used by the dev team (running Ubuntu Linux).

Obviously, building a tool that assists in better software engineering, your product should be designed well and according to the best practices of software engineering. You will be strongly evaluated on developer satisfaction and effectiveness of the tool in helping teams to grow.

Company Description

Founded in 2014, KeyLocker is a start-up that is currently developing cryptographic protocols that put the control of encryption keys into the hands of the end-users, instead of service providers. Founded on the philosophy that Internet services should be 'given back'to the consumer, KeyLocker develops crypto products from the ground up that are unparalleled in the market. Employing mainly talented students from the TU Delft that can handle the steep learning curve, knowledge sharing is a crucial component of the company's success. The chief languages used for development are Scala and C, on linux systems.