Master Thesis

April 29, 2014

SQA-Pattern

A Recognition Framework for Violations of Conventions in Software Engineering

Stefan Hiltebrand

of Zürich, Schweiz (08-713-117)

supervised by

Prof. Dr. Harald C. Gall

Martin Brandtner

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

SQA-Pattern

A Recognition Framework for Violations of Conventions in Software Engineering

Stefan Hiltebrand

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Stefan Hiltebrand, stefan.hilti@bluewin.ch

Project period: 29.10.2013 - 29.04.2014

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Prof. Dr. Harald Gall for giving me the opportunity to write this thesis at the software evolution and architecture lab at the University of Zurich. Many thanks also goes to Martin Brandtner for his great assistance. Furthermore, I thank Jens Birchler for the smooth cooperation.

Abstract

In software engineering, different tools have become popular to support the development and management of software projects such as version control repositories or issue trackers. These tools are (web-based) approaches that try to offer best possible conditions to develop software, especially in teams using agile development methods. However, if the tools are not used as intended, it is harder to obtain an overview of the project. To avoid this problem, conventions on the usage of these tools can be defined, as in the source code development, where it is generally accepted that defining conventions on how to structure the code and complying them leads to better results. Thus, there are many applications to analyse the code and find violations of these conventions. Similar analyses are possible to find violations of the conventions on the usage of the above mentioned tools, but there are no applications that provide this data. This gap is tried to be filled by the SQA-Pattern approach presented in this thesis. SQA-Pattern mines the data from the different tools and analyses it for certain patterns that describe violations of the tool usage conventions. The goal thereby is to provide the data as fast as possible such that the approach has the potential to be used in agile software development. The conducted evaluation has proved that the data can be analysed in less then ten minutes even for large-scale software projects.

Zusammenfassung

Im Software-Engineering sind verschiedene Tools zur Unterstützung der Entwicklung und Verwaltung von Software-Projekten populär geworden, wie Version-Control-Repositories oder Bug-Tracker. Diese Tools sind (web-basierte) Ansätze, die versuchen bestmögliche Bedingungen für Software-Entwicklung zu bieten, vor allem für Teams die agile Entwicklungsmethoden verwenden. Werden die Tools allerdings nicht bestimmungsgemäss verwendet, ist es schwieriger das Projekt zu überblicken. Um dieses Problem zu vermeiden können Konventionen für die Verwendung dieser Tools definiert werden. Dies ist ähnlich wie in der Quellcode-Entwicklung, wo es allgemein anerkannt ist, dass das Festlegen und Einhalten von Konventionen über die Strukturierung des Codes zu besseren Ergebnissen führt. Deshalb existieren in diesem Bereich viele Anwendungen, die den Code analysieren und Verstösse gegen die Konventionen aufzeigen. Ähnliche Analysen wären möglich zur Suche nach Verstössen gegen Konventionen, wie die Tools zu benutzen sind. Allerdings existieren keine Anwendungen, die diese durchführen. Die vorliegende Arbeit versucht diese Lücke mit dem SQA-Pattern-Ansatz zu füllen. SQA-Pattern ruft die Daten der verschiedenen Tools ab und analysiert sie auf bestimmte Muster, welche Verletzungen der Toolbenutzungs-Konventionen beschreiben. Das Ziel dabei ist, die Resultate so schnell wie möglich zu erhalten, damit der Ansatz potenziell in der agilen Software-Entwicklung benutzt werden kann. Die durchgeführte Evaluation hat bewiesen, dass die Daten selbst bei grossen Software-Projekten in weniger als 10 Minuten analysiert werden können.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In modern software engineering, continuous integration tools are an important component in the development of software projects. Version control systems allow teams to combine their work on the source code. Through bug tracking systems, the different issues on a project are written down on a centralized platform such that all stakeholders can view them and interact with. Continuous integration servers facilitate the deployment of the source code already in early states of the project and provide information on not properly running unit tests. Source code analysis platforms supply information on code smells. All these tools specially support projects that are built in teams and development processes with short cycles, such as Scrum [RJ00], an agile development method.

1.1 Motivation

In software development, conventions on designing the source code became accepted to have less bug prone code, a better architecture, easier overview of the code, and simplified maintainability. Best practice patterns are one example of code conventions to reach these goals [Bec97] design pattern another [GHJV94]. There are even various tools such as Checkstyle[1] that analyse the current state of a project and deliver information on violations of these conventions.

It is possible to define similar ones for the use of the above presented development supporting tools. In the same way that violations of source code conventions indicate wrong usage of the programming language, these violations indicate misuse of the tools. However, this misuse does not directly imply a bug in the code, but highlights behaviour that can be the reason for upcoming bugs. For example in issue trackers, a resolved issue can be stated as closed, which signals that the code is changed and the tests are running, indicating no more work is needed on this issue. If now a developer commits a source change to the closed issue without reopening it, there is no guarantee that the code is tested properly, and it is therefore bug prone. Another example are commits not linked to an issue. These commits tend to be missed by quality reviews that focus on the issue tracker and thus lead to not reviewed code changes. One can think of building various other conventions. They can be very project specific, or rather general, similar to the source code conventions.

These violations of conventions in the usage of the development tools can not be tracked. They are easily missed, possibly leading to gaps in the code quality. They also can not be technically prevented since multiple tools are involved and because the conventions differ for every project.

[1] http://checkstyle.sourceforge.net/

1.2 Goal

The main goal of this master thesis is to provide an efficient way of analysing the data of a version control system, a bug tracking platform, and continuous integration tools for violations of project conventions. The vision is to combine the data of the different tools, to analyse it for violations of the project specific conventions, and to provide the resulting violations. Therefore, a proof-of-concept implementation of a software quality assurance (SQA) framework called SQA-Pattern is presented, which allows for an automatic analysis of project conventions violations. SQA-Pattern, first of all, has to handle the gathering of the data from the different resource tools. This data the adherence to the conventions is analysed. The focus thereby does not lay on the current state of a software project, but on the detection of violations in its history. To define which violations have to be found, different categories of work patterns are supported that allow for the definition of project specific violations. As a last part, the resulting data is provided via RESTful web services allowing other tools, such as SQA-Timeline [Bir14] to display the information.

Since violations have to be detected as fast as possible to prevent increasing costs of a project and because agile software development processes often include a daily meeting, up-to-date information is highly important. Therefore, the analysis duration has a great impact on the benefit, of the SQA-Pattern approach. Thus in the evaluation of this thesis, the following research question is investigated: *How efficiently can violations of project conventions, such as missing issue keys in commit messages, be automatically detected in software projects?*

The structure of this thesis is as follows: After this introduction the related work is presented. Next, the used resource data is explained, as well as a the insights of a previously implemented project that analyses comparable data. In Chapter 4, the concrete implementation of SQA-Pattern and the considerations behind it are presented. Chapter 5 describes the evaluation to determine the efficiency of the approach. Finally, the conclusions of the thesis as well as the thoughts on future work are located in Chapter 6.

Chapter 2

Related Work

In this chapter, work related to the SQA-Pattern approach is presented. It is divided in three main parts: information needs, large-scale data provision, and violations of code conventions. Literature on information needs covers the stakeholders and their lack of suitable data in software projects. Large scale data provision illustrates what tools are already available to mine data from projects or repositories and provide it to the stakeholders. The studies on code convention violations investigate similar approaches as SQA-Pattern but concentrate on code based violations. Last but not least the connection of the literature to this master thesis is explained.

2.1 Information Needs

Fritz and Murphy [FM10] collected 78 frequently asked questions of developers working on a software project. Many of these questions are code specific but there are also questions about the co-workers, broken builds, test cases, and some other groups of questions. These questions were collected by interviewing eleven professional software developers. Fritz and Murphy found that some of these questions are easy to answer because they focus on one kind of information. On the other hand, questions that require the integration of multiple kinds of information are harder to answer because the information has to be linked. An example is *What have my coworkers been doing?* In this paper an information fragment model is presented. It integrates chosen information automatically and allows to express a large variety of questions. First, the developer chooses the information he is interested in. Then, the prototype links these information fragments and presents a visual graph that can be ordered by the fragments. In the evaluation, 18 developers had to try answering eight sample questions. The participants were able to answer 135 of 144 questions successfully, most of them in less than five minutes.

Another catalogue of questions was collected by the study of Breu et al. [BPSZ10]. They analysed the questions that were asked to different bug reports to identify which information is needed. For every question, meta-information is stored. This indicates to whom the question is addressed or what the focus of the question is. The collected question were grouped in eight different categories. Based on this categories, statistical analyses were executed e.g. on the time these questions were asked in relation to the creation of the bug report or if an answer was provided. As a conclusion, they came up with some ideas for improving bug tracking systems to support the answering of frequently asked questions.

De Alvis and Murphy [DAM08] also focused on the support a developer obtains by combining information from different sources. A software exploration tool named Ferret is presented. It uses a model to support the integration of different sources for software information. The focus of Ferret is to provide support to answer 36 questions (conceptual queries) about the code of a software project. These questions were chosen from the literature, from blogs, or from the

experience of the authors of the paper. In the evaluation some professional software developers used Ferret for two days. The developers used the given conceptual queries and stated the tool as qualified to support their software development.

Ko, DeLine and Venolia [KDV07] observed software developers at their work to see what kind of information they need. 21 types of information needs were identified. Most of these refer to the source code but also information about the work of the co-workers and bug (re-)production are needed. The observation showed that some of these questions can only be answered in unsatisfying way and/or require a long search time. The study discovered that the co-workers are mostly able to provide most of the information the developers needed.

Aranda and Venolia [AV09] analysed bug histories to identify common bug fixing coordination patterns. They arranged a case study on closed bugs from three major product divisions at Microsoft. They analysed the development history of the bugs and the different interactions of actors with the bugs. In total ten bugs and their history where analysed. In addition, a survey with 1500 Microsoft employees (response rate 7.9%) evenly divided between developers, testers and program managers was arranged. They were asked about their last bug, its history and actors, the same data as analysed in the case study. Both studies produced very similar results. These results lead to a list of 28 coordination patterns in the bug histories. Out of these patterns, eight goals for bug fixing were derived as a framework for future work.

2.2 Large-Scale Data Provision

Dyer et al. [DNRN13] focused on the analysis of ultra large-scale code repositories for similarities and differences by creating a domain-specific language called Boa. Boa is focused on retrieving data from ultra large-scale code repositories, especially SourceForge[1] projects, that use SVN, and automatically analyses this data. As the language is focused on this task it only needs some lines of code and is much faster than common programming languages. For example the task *How many revisions are there in all Java projects using SVN?* only requires 5 lines of code and 59 seconds to execute. In Java, the same task needs 60 lines of code and the run time is 4636 seconds.

Another approach focused on projects using a repository is MetricMiner, a web based application described by Sokol et al. [SAG13]. MetricMiner analyses the git or SVN repository of a project to provide code metrics, an interface to analyse the data and statistical tests. It has some predefined metrics that are added on every project but the user can also create specific metrics and tasks for the data to personalize the results. The data is only gathered directly from the version control system and therefore the platform is limited to this data. For example, information on the compiled code is not accessible.

There is another web based approach, called Ohloh[2]. Ohloh does not investigate deeply into a project but rather provides some statistical data. For example the application shows for each project the number of lines of code, the used programming languages, and its activity (commits) on a project and their influence.

A web tool that provides more statistics, even down to code level, is SonarQube[3]. It analyses the source code and the continuous integration data to provide information in so called metrics. For every metric, the actual source code can be viewed to see which impact it has. Additionally, the evolution of the metrics is shown to visualise how the project evolves.

SOFAS (SOFtware Analysis Services) by Ghezzi [Ghe10] and by Ghezzi and Gall [GG11] provides a RESTful web service to run remotely some predefined analysis on a project and fetch the results. The data for the analysis is gathered from version control systems (CVS, SVN and GIT

[1] http://sourceforge.net/
[2] www.ohloh.net
[3] http://www.sonarqube.org/

repositories) and issue tracking history services (Bugzilla, Google Code, Trac and SourceForge). A stakeholder can chose the specific analysis (e.g. code metrics) he wants to run depending on his needs.

Troxler implemented an application called MiningHub [Tro14] that allows to mine data from different social coding sites like GitHub, JIRA or Stack Overflow. Using mining scripts, the information how the different resources have to be handled can be edited by a user. This allows him to add new resources or change the exact information that is mined. Therefore, the application can support different studies on gathering the desired data and avoid the implementation of multiple tools by different researchers for the same purpose. Since the scripts can be shared among different users, an easy replication of data used for studies is provided.

2.3 Violations of Code Conventions

PR-Miner [LZ05], an approach by Li et al., analyses large-scale software code written in C to extract automatically undocumented general programming rules. Afterwards, the code is searched for violations of these rules. The PR-Miner is evaluated with different large-scale software with up to 3 millions lines of code such as Linux. The approach is extremely fast, only needing up to two minutes for the analysis. Additionally, it has been shown that of the top 60 violations found, at least 33 were bugs that had to be fixed in the evaluated code.

Mens et al. [MMW02] developed a similar tool to find code smells. However, instead of creating the rules out of the code, it searches for programming pattern, such as design patterns, in the source code of software projects written in Smalltalk. These pattern are used to develop clean code that benefits from the experience of other software developers that already faced similar tasks. The approach finds violations of these patterns to support software developers in improving their source code.

With SemmleCode, an Eclipse Plugin by Verbaere et al. [VHDM07], not only given violations of predefined patterns can be searched. Using a query language called *.QL*, own project specific conventions can be implemented based on design patterns, Java style rules, or even own rules for a source code of a project. The tool automatically searches for violations of these conventions and displays a warning if any violation is found.

There are various other tools such as Checkstyle[4], FindBugs[5] or Jtest[6] as examples for Java source code to statically analyse the code directly in the development environment and find predefined patterns that indicate violations of code conventions or concrete bugs. The already mentioned SonarQube finds violations of the Java style rules running remotely.

2.4 Link of SQA-Pattern to the Literature

Many different source code conventions are defined for software projects depending on the language of their code. Various tools available in literature support developers in finding violations of these code conventions. These tools are well-engineered and are used by many developers to create cleaner code. For the usage of version control systems, bug tracker, and continuous integration platforms similar conventions exist already or can be formulated. For example, a commit to an already resolved issue is disencouraged. Studies reported in literature, have shown that there are many questions that are hard to answer. They concern continuous integration tools and the general map a stakeholder has on the usage of these tools by the developers. For example

[4] http://checkstyle.sourceforge.net/
[5] http://findbugs.sourceforge.net/
[6] http://www.parasoft.com/jtest

to know what has caused a change of the build result and why that source code was committed. Even though the different tools have some conventions to simplify the handling of a software project, it is not guaranteed that this conventions are met by the developers. SQA-Pattern makes a connection between these two subjects by providing information on violations of conventions for the use of the tools and not the source code itself. To provide this informations, first of all the different resource tools are mined to get the information needed. On mining continuous integration tools, there are also various studies and even though the focus of the analysis in this studies is not the same, some findings can still support the approach of SQA-Pattern.

Chapter 3

Background

In this chapter the background of SQA-Pattern is explained to give insights on the data that is analysed by the application. First, the different kind of tools SQA-Pattern uses as resources are introduced and it is shown which data they provide. Additionally, SQA-Mashup[1] is presented, an already implemented prototype for the combination of project information from different platforms. The experience gained thereby, builds the base for structural decisions in SQA-Pattern.

3.1 Resource Tools

There are multiple tools to support the development process of a software project, often providing the same data. To keep the effort feasible for the SQA-Pattern approach, the number of supported resource tools for the data gathering is limited. Many open source projects of the Apache Software Foundation[2] actively use the same tools, generating a huge amount of development history data. Therefore, these tools are chosen as resources in the approach. However, SQA-Pattern is designed to allow adding further resources that provide the same or similar data.

3.1.1 Git - A Version Control Repository

Git is a "free and open source distributed version control system"[3]. A version control system is a repository of content that provides access to historical editions and records all the changes in a log [LM12]. Thereby distributed means that the entire data of the repository is *cloned* to every users hard drive. An example of a not distributed version control system is SVN[4] where only a *checkout* of the current tip of the repository is downloaded to the users computer. In Git every change event (commit) in a repository is logged and the whole project can be reverted to exactly the same status. This allows to see all the added, modified or deleted lines and/or files of every change event in a project. A commit contains a hash code for identification, the author of the change, the timestamp, and the differences in every file.

[1] http://www.ifi.uzh.ch/seal/research/tools/sqa-mashup.html
[2] http://www.apache.org/
[3] http://git-scm.com/
[4] http://subversion.apache.org/

3.1.2 JIRA - A Bug Tracking System

JIRA[5] is a platform to collect all the changes (issues) in a project that need to be conducted (e.g. bug fixing or adding new features). It is a web application to allow multiple users to track a project, add/change an issue, or comment on an issue. Every issue has to contain a summary, an exact description, a priority, an assigned developer who is responsible for the task, and the reporter of the issue. There are also some optional values e.g. the date an issue has to be resolved on. To follow the evolution of a project, every issue has one of five statuses. When an issue is created the status is set automatically to *open* to show that some work is needed. A developer then can change the status to *in progress* to indicate that he is working on it. When he added his code changes to the repository and thinks the issue is fulfilled he can change the status to *resolved*. To *close* an issue all the tests have to run through successfully. The last status is *reopened*. A reopened issue indicates that the issue was once resolved or even closed but the solution was not correct or the bug occurred again in a later phase of the project and therefore code changes are needed again. Every change to an issue is stored in the change log with the exact timestamp. All this data can be accessed via a web application or RESTful web services.

3.1.3 Jenkins - A Continuous Integration Server

Continuous integration (CI) allows to build the software for quality assurance after small changes even early in the project instead of waiting for fully produced software [Ber12]. This helps to find not only code errors but also errors in the build scripts or in the deploying to the server. Jenkins[6] is a Java based open source CI server. In Jenkins, builds are typically initiated by a user or after a given time period. As with JIRA, the information can be accessed by a web application or RESTful web services. Some important information Jenkins provides is for every build a number to identify, the result (e.g. *SUCCESS* or *FAILURE*), the timestamp, the duration and other build specific information.

3.1.4 SonarQube - A Software Quality Measurement Tool

SonarQube is a web-based open source tool to analyse the code quality. It covers "the seven axes of code quality"[7], that are dublications, unit tests, complexity, potential bugs, coding rules, comments, and architecture & design. SonarQube provides its information as metrics, which are for example lines of code, rules compliance, unit test coverage, and many more. Normally, the shown data is focused on the current state of the development. For the metrics it is possible to drill down into the code view to see exactly which line affects a metric. SonarQube takes the source code of the project and the CI as source for its metrics. Additionally to the current view, there is the time machine to view the history of all metrics and its development. All this information can also be accessed by RESTful web services.

3.2 SQA-Mashup

The SQA-Mashup is a platform to integrate the information of different data sources into one tool. As SQA-Pattern, the SQA-Mashup approach uses a version control repository, a bug tracker and continuous integration platforms as data resources. In addition to just combining the data, some *smart filters* are implemented to highlight possible error sources, e.g. often reopened issues.

[5] https://www.atlassian.com/de/software/jira
[6] http://jenkins-ci.org/
[7] http://www.sonarqube.org/

3.2.1 Implementation Details

In the backend, the tool is built on so called pipes. To simplify the structure, these pipes can be seen as a row of actions to receive the desired information from the different RESTful services, transform them into the target format and if required (especially for the filters) perform some calculations to decide which information has to be shown. Every pipe leads to a Widget in the overview and is called every time the user wants to see the project details. This composition ensures flexibility, adaptability and always up-to-date data.

3.2.2 Impact

An evaluation of the SQA-Mashup platform [BGG14] has shown that if information is needed from different CI tools a user needs significantly less time to find it using the SQA-Mashup tool instead of the different CI tools used as resources by SQA-Mashup. Also the correctness of the received results is higher.

3.2.3 Known Restrictions

However, the approach of the SQA-Mashup has some known restrictions. First of all, the tool is not constructed for large-scale projects. With thousands of commits and issues the calculation needs too much time to use it in a reasonable way especially of the smart filters. A user does not want to wait several minutes on actualized results every time he uses the tool. Even though there is a cache to save already loaded results, active projects need a frequent update to provide information on the newest evolution. With large-scale projects also the other widgets need too much time to gather all the data even though they are not as critical as the filters. Secondly, even tough the platform is built to be flexible in the presented data, the smart filters can not be changed much. If a new information has to be filtered, a software change is needed. Another disadvantage is that a developer concentrated view is not supported. This is informative to see which developer is responsible for which action.

Chapter 4

SQA-Pattern

This chapter describes the implementation of SQA-Pattern. First, the approach is explained. Secondly, the software architecture is shown. Then, the important libraries and their use in the application are introduced. As last part, the three sub projects and their contribution to the project and concrete implementation are presented.

4.1 Approach

The vision of SQA-Pattern is an application that supports software managers and developers in their daily work by finding violations of project conventions and providing this information. The stakeholder therefore can focus on the resulting data and does not need to analyse the whole project history.

The resource tools used are predefined to allow a reliable evaluation and to stay in a feasible size for the prototype. However SQA-Pattern allows the future integration of additional resources without many changes in the existing code. The important condition is that the tools provide similar information: the revisions of a version control repository, the issues of a bug tracking system, the builds of a continuous integration platform and some source code metrics.

The previous work on SQA-Mashup and in the literature (e.g. [Tro14]) indicates that the mining and the analysis can not be performed in real-time when it is needed, if large-scale software projects have to be supported. The information has to be extracted before because it takes too much time especially for the mining. This is implemented by separating SQA-Pattern in three parts and is explained in detail in Section 4.2.

Work Pattern

The different chosen resources and known projects using them are examined for conventions or principles on how the tools have to be used. Based on this conventions and principles different work pattern are built to describe the violations. To fully understand how they are extracted the linking feature of JIRA has to be explained:

JIRA supports an interlinking of commit messages with their issues. Adding some predefined codes to the message allows to illustrate that a change is committed for a specific issue. It is even possible to directly record the time that is used for the work on a commit. Additionally, a new comment to an issue or a status change of an issue can be placed in the commit message. JIRA then parses the message and performs the desired operation.

The implementation of these pattern has to be as generic as possible to allow different use cases without code changes. However, a fully generic identification is not possible in this approach, the work patterns are therefore divided into the following categories by the way the data is analysed:

- **Regular Expression**
  On a chosen sort of text (e.g. commit messages) a search for regular expressions is executed. Therefore, a violation of the convention how commit messages have to be built (e.g. the inclusion of a link to an issue) can easily be found.
- **Time**
  With the time pattern a timespan before and/or after an event can be analysed to find specific other events. A possible usage is to find violations of the convention to commit only on issues that are not resolved.
- **Cluster**
  The cluster pattern searches for accumulations of defined events. For example the closure of multiple issues in a short time span indicates duplicated issues and therefore, a violation of the convention on how to use JIRA.
- **Issue Contribution**
  The issue contribution pattern is a special case of the cluster pattern only searching for commits on the same issue in a given time span. That is a hint on a violation of the convention that only one developer is allowed to work on an issue or that a commit is only performed when the change is complete.
- **Event**
  Special events, for example reopening of issues, can be searched through the event pattern. A reopened issue is a violation of the convention to close only bug-free issues.
- **Metric**
  The metrics pattern allows to find specific values in metrics. A decrease of defined metrics can therefore be highlighted easily. This pattern is a special case since source code conventions have an influence, but the general convention violated is focusing not on the code itself but on the developers committing. For example, test unit success is not allowed to fall beneath a given threshold.

These categories all have defined structures on how they are described. This allows to create patterns for specific tasks not knowing the structure of the data that is used. Hence, it is not possible to create generic patterns, only the different variables can be replaced. For every new pattern a code change is needed. This forces to structure the code in a way that new pattern categories at least need only few code changes.

For every pattern a severity rating can be defined. A high value indicates a strong violation whereas a low value is used for rather insignificant violations. A severity of zero stands for neutral pattern only used to fulfil information needs. For every developer, the severity rating of a pattern is multiplied with the number of instances of this pattern he is involved in. This allows to see which developer does not adhere to the conventions.

4.2 Software Architecture

To meet the main goal of the SQA-Pattern framework, the analysis of the continuous integration data on specific patterns, first of all the data must be collected. Then, the data is analysed and as last step, the application has to provide the data using RESTful services. This set-up allows to split the project directly into these three parts as it can be seen in Figure 4.1. The advantage of creating three independent subprojects is that the whole project is much more flexible and adaptable. One part can be changed or even replaced without disrupting the other two parts. It is only important that the format of the resulting data of one subproject is not changed. To protect the data a user

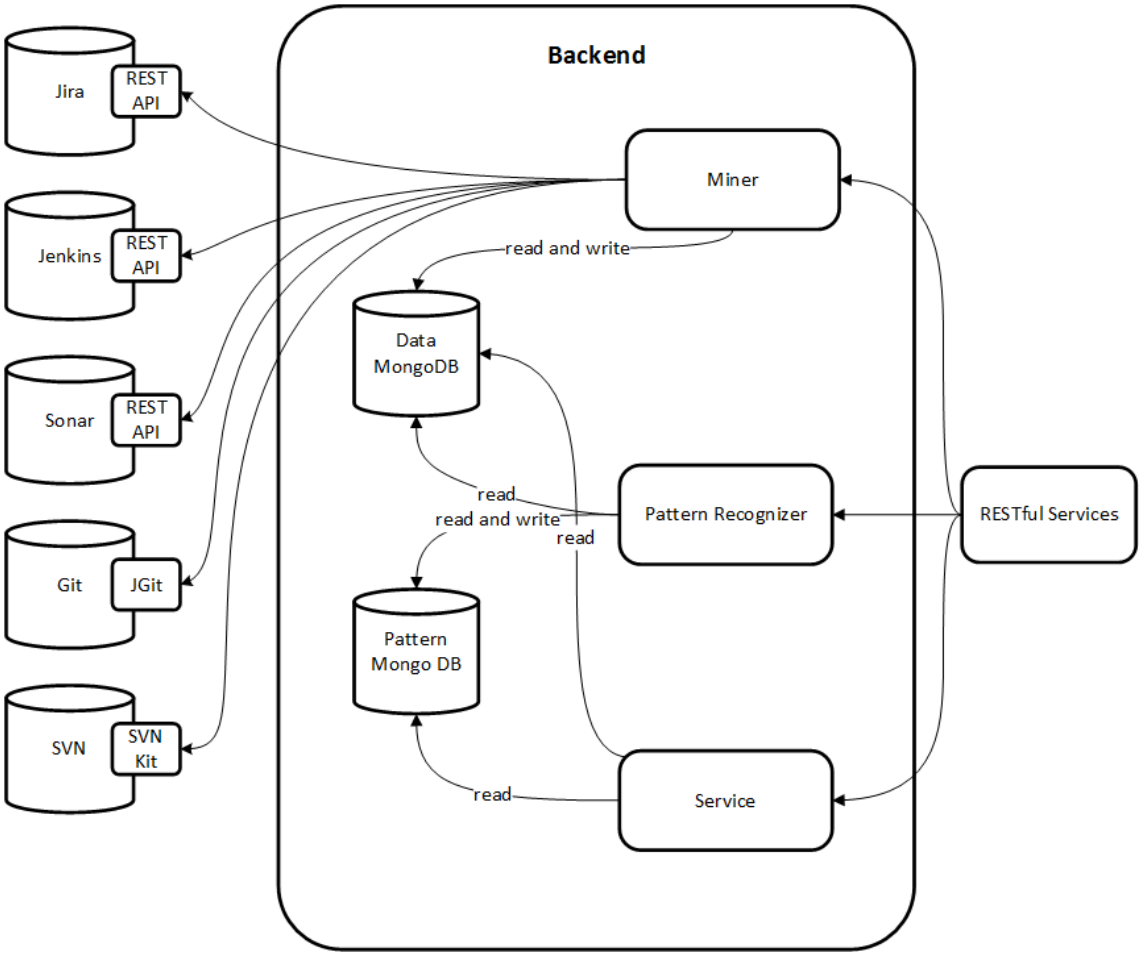


Figure 4.1: Software architecture

login is supported. Through the splitting of the project, it is possible to create users having only access to some parts of the project, e.g. the resulting data. This allows to protect a project for changes but to still provide the data.

4.2.1 Mining

The mining includes the collection of the data for all the predefined resources for one project, convert it to the desired format and store it in the database. The mining of the data has to be as fast as possible. The faster it is, the more often the data can be updated. To guarantee current data that can be used by a software manager, at least an update every day has to be supported. Software developers perhaps prefer even more actualized data to see directly new events without using other tools.

The data for the chosen resource tools is either fetched through RESTful web services or a Java framework that organises the import of the data. After mining the data, it has to be changed to a suitable format to store it in the database. Also the database has to be organized such that the up-to-date data can be provided to the other parts of SQA-Pattern.

To guarantee access to the data all day, the miner is built in a way that the link to the data is only overridden after the successful mining. The process also blocks other attempts to mine the same data to prevent duplicated data, useless computation and connection effort. To minimize unneeded costs a version number is stored to every data that allows to stop the mining process if no new data is available.

The mining process can be started by calling a web service for every resource in a project.

4.2.2 Analysis

To analyse a project on its predefined pattern the data of the mining process is read out and sorted. Afterwards, the data is analysed for every pattern and the resulting violations stored in another database. This split of the databases allows to give the analyser only read access to the mining database. The version control and the bug tracking data as well as the generated pattern results are then analysed to find all the corresponding events for a developer and combine it. This allows to collect a severity rating for every developer to analyse how much he is involved in negative patterns. It also shows if a developer is active in a project or not. This information is generated for the whole duration of the project but also for the recent time span (one week). The resulting developer data is also stored in the second database.

This analysis for pattern and developers always has to be executed after mining the data of a project to have suitable results. Therefore, it is also necessary to have a fast search to allow the data to be updated as much as possible, at least once a day.

Again the data is accessible all the time in the same way as the mining data. Attempts to analyse data an analysis is already running on are blocked. A determination to only analyse if new resource data is available was discussed but too complex, because the pattern can be changed and therefore generate other results. Although this change of a pattern can be stored, it results in the current database structure in the possibility of two services changing the same file and therefore was omitted.

The analysis can be started using a web service and an automated start is possible. In addition, there are web services for saving, deleting and getting the pattern definition to allow project specific patterns.

4.2.3 Data Provision

The last part of the application is the provision of the previously stored data. It only needs read access to the database as the data is not changed any more. The data is fetched out of the database and provided as directly as possible to enable fast web services. Naturally the data still needs some filtering, preparation or combination to be convenient. The individual services are:

- **Summary**
  All the events of the different resources are combined, sorted by date and provided. To prevent an overflow the data is filtered for the important information.
- **Pattern results overview**
  The number of intervals of every pattern over the whole project period as well as for the last week.
- **Pattern results**
  All the resulting intervals of the pattern analysis.
- **Developers**
  The developers and their commits and issues as well as their involvement in analysed patterns.

- **Last commits**
  A list of given number of the last commits.
- **Info**
  This service expects an id of an element and returns all the stored information.
- **Source**
  Expecting a commit-id and a file path, the source file before and after the commit is returned.

4.3 Important Libraries

The SQA-Pattern framework is written in Java. In addition to some well-known Java libraries, there are other frameworks used in the prototype to simplify the tasks and make use of already implemented functions.

4.3.1 Play Framework

The Play Framework[1] is a Java and Scala web application framework that has integrated support of (RESTful) web services. The focus lies on minimal resource consumption to facilitate highly-scalable web applications. Play is highly supporting asynchronous programming for web calls as well as for the actual programming. It supports the actor-based model to handle concurrent systems and long running tasks by integrating Akka[2]. Play also handles the build system using sbt[3]. This allows an easy and flexible way of compiling and deploying the code on a server, integrating other libraries, and package a project as JAR file.

Play is therefore the ideal choice to build SQA-Pattern on. The framework provides a way of doing large-scale web operations already checked by various other developers. The easy way of implementing asynchronous calls allows to use them often and without thinking of the operations that have to be handled in the background. Since all the mining and analysis operations work with asynchronous calls, this is an important point. Using the build script of Play a continuous integration using Jenkins is set up providing all the RESTful web services already during the development period.

4.3.2 JGit

JGit [4] is a Java library that implements the Git version control system. It allows to execute the same calls as with Git on the command line plus direct handling of the resulting information in Java. In SQA-Pattern it is used to clone the Git repositories and analyse all the commits and their corresponding data. With JGit it is possible to perform many other actions such as commit new code but this functionality is not needed in the application.

4.3.3 SVNKit

SVNKit[5] provides the functionality to use SVN in a Java application . Basically SVNKit provides the same functionality for SVN as JGit for Git but due to the different composition of the repositories in a varied way. In SVN the whole project can not be cloned but a checkout of the current

[1] http://www.playframework.com/
[2] http://akka.io/
[3] http://www.scala-sbt.org/
[4] http://eclipse.org/jgit/
[5] http://svnkit.com/

state is performed. Therefore for every commit, a new connection to the repository has to be established to obtain the files at this state of the development history.

4.4 Miner

The main part of the miner is, naturally, the mining of the different resources. But before the data can be gathered, a project configuration has to be uploaded. This can be performed through a PUT request on the web services. The body must include the JSON formatted project configuration.

4.4.1 Project Configuration

As an example the document for Apache Camel[6] is shown in Listing 4.1. The *key* value can be set at will, the only restriction is that using an already existing key with case insensitivity, will overwrite the existing project configuration. The *name* has no influence on the data, it is just for human recognition of the project. Setting the *startdate* in milliseconds since 1970 allows to ignore all the data before this timestamp. The *vcsSources* array includes all the relevant information for the version control repository. Theoretically, it is possible to add more than one repository but in the current state of the prototype additional repositories will be ignored. In the *type* is stored if it is a Git or a SVN repository and the url links to it. For protected repositories a user name and a password can be set. To specify which branch has to be analysed it can be set in the *name* variable. In the *wsSource* array all the tools can be specified that are accessible through a web service. Again the *type* value defines what sort of resource it is. In the current project state it can be Jenkins, JIRA or SonarQube. The *url* links to the home directory of the resource specific RESTful service and the *name* value has to be the specific key that is used to identify the project. For every resource a user name and password for authentication can be set if needed. Additionally, for SonarQube, the metrics to mine have to be set in the resource configuration.

```
{
   "key": "CML12",
   "name": "Apache Camel 01.01.2012",
   "startdate": 1325376000000,
   "vcsSources": [
      {
         "name": "master",
         "password": "",
         "type": "git",
         "url": "git://git.apache.org/camel.git",
         "username": ""
      }
   ],
   "wsSources": [
      {
         "name": "Camel.trunk.fulltest",
         "password": "",
         "type": "jenkins",
         "url": "https://builds.apache.org/job",
         "username": ""
```

[6] https://camel.apache.org/

```
        },
        {
            "metrics": "violations_density,violations,ncloc,test_errors,tests,
            coverage,test_success_density,test_failures",
            "name": "org.apache.camel:camel",
            "password": "",
            "type": "sonar",
            "url": "https://analysis.apache.org",
            "username": ""
        },
        {
            "name": "CAMEL",
            "password": "",
            "type": "jira",
            "url": "https://issues.apache.org/jira/rest/api",
            "username": ""
        }
    ]
}
```

Listing 4.1: Example project configuration.

When the web service is called, first of all the request body is analysed to determine if the JSON syntax is correct. As a next step it is inspected if the main values (key, name, startdate, vcsSources and wsSources) in the JSON exist. Further analyses are not executed. If an incorrect document is uploaded either an error will be thrown during the PUT request or the mining just won't work. The same happens, when the data in the document is not linking to a correct resource API. If there was no exception in the inspection the document is stored in the database in a collection of all the configuration files. Thereby, the key is changed to upper case letters and a possibly already stored file with the same key is overridden. Using a lower case key in any part of SQA-Pattern will always result in the upper case key. If everything worked correct a HTTP status 201 - *created* will be returned.

A project can also be deleted through a web service. If this service is called the configuration file gets removed from the database. Also all the collections of the project will be dropped, deleting all the stored data.

To get an already existing project configuration a web service can be called using the predefined key of the project. There exists also a web service to receive the key and name of all the stored projects. The detailed links to the services can be seen in the Appendix A.

4.4.2 Database structure

Every project has a collection in the database that stores the status information of the data for all resources and some collections that store the mined data. An example of a status information document can be seen in Listing 4.2. The type value is to identify the correct document for a resource. In the version field the actual version number is stored. This has two purposes, first it allows to compare the stored number with the actual version of the resource and second the number is used to find the collection with the resource data. These collections are named with the key of the project, the resource of the data, and the actual version number. If new data is mined, it is always stored in a new collection allowing to provide the old data until the new one is fully uploaded. Since the old data is not deleted directly another part of SQA-Pattern that is

perhaps using the old collection can still continue working. If new data is ready to be used the version number in the status information just has to be updated and the actualised data can be accessed. Starting the first mining the version is set to null until the operation is finished. In *timeOfVersion* the timestamp when this version was uploaded is stored. To prevent more than one running mining operations on the same resource of a project the *inUse* value exists. At the start of the mining it has to be set to *true* and at the end back to *false*. Since it is always possible that a mining operation is interrupted, for example because of a server crash, the start timestamp of the mining is stored in *startOfUse*. This allows to check if the mining takes to much time and release the data even though the *inUse* value is still set to *true*. In the example in Listing 4.2 the value is null because no mining is active and therefore no start time can be set.

```
{
   "type": "Jenkins",
   "version": 1768,
   "timeOfVersion": 1396621302312,
   "inUse": false,
   "startOfUse": 0
}
```

Listing 4.2: Example resource status information.

4.4.3 The Mining

To mine some data a resource specific web service has to be called. As input the key of the project is needed. All the web services have in common that first of all it has to be checked if the project configuration for the given key exists. If this is not the case a HTTP status *404 - not found* is returned. If the project configuration exists the status information is checked to detect if there is already running a mining operation on the specific resource of the project. If the *inUse* value of the status document is set to *false* or the *startOfUse* timestamp is older than an hour the resource is taken as unused. In this case the *inUse* variable is directly set to *true* and the actual timestamp is stored as the beginning of the usage. Having set the usage information the actual mining can begin. This is different for every resource but it is always one or more asynchronous calls running in the background. The user who called the web service already gets a HTTP status *202 - accepted* to indicate that the mining is now running but a correct result can not already be guaranteed. A HTTP status *423 - locked* indicates that the resource is not free for mining. Since other parts of SQA-Pattern have to access the data stored in the database collections the old data gets not directly deleted because it can interrupt a running program using the data. Therefore every time a new mining starts it is analysed if the old version was older than a day. If this is the case all other existing versions of the data are deleted to reduce the space that is needed.

Git Mining

To gather the data of a git repository, first of all a project specific directory is set up, where the repository will be cloned to. If the directory is not already empty, it has to be cleaned up. Having an empty directory, the clone operation can be started. This is performed by getting all the information of the project configuration such as the URL of the repository. The clone is only executed on the predefined branch to save time by skipping all the irrelevant commits. When this operation is finished, the local repository can be analysed. First, the application takes the timestamp of the newest commit as version number and compares it with the stored one. If the version is the same, the whole operation is aborted and resource set to unused. Otherwise, the miner searches

through all the commits and writes out the commit identification of the secure hash algorithm (SHA), the author, the email of the author, the date and the changed files if the commit is newer than the defined start date in the configuration, otherwise ignoring the commit. To directly get the exact change of a file in a revision without analysing the commits before, the *diff* file and the exact version of the resource file after the change have to be stored. Since a large-scale project can have 10000 and more commits every commit has to be stored directly in the database. Not storing them, sooner or later leads to an overflow in the Java heap space. When all the commits are stored in the database the directory of the local Git clone is deleted. Then, the status information of the resource is updated storing the new version and setting the resource to *not in use*.

SVN Mining

Since a project in SVN can not be cloned, a local repository is not needed for the SVN mining. Instead the connection to the repository is established and the number of the latest revision fetched. This is than used as version number to compare with the already stored data and decide if a new mining is needed. In this case all the log entries of the project are fetched and handled one after the other. This log entry includes the number of the revision, the author, the message and the timestamp. The email of the author can not be retrieved. Unfortunately also the changed files are not directly included in the log entry, only the path of the file before and after the revision. Therefore, two request to the remote repository have to be executed for every changed file. Having all the information needed, the data is stored in the database and the next log entry gets analysed. After the last one, the status information of the resource is updated. Naturally, the revision is only analysed if the date is newer than the start date in the configuration.

Because this analysis needs many independent connections to the remote repository the mining of the data takes much more time than using a Git repository. SVN offers the possibility to first copy the whole repository to a local path and then analyse it locally. This version is probably faster if only one project is stored in the SVN repository. But unfortunately, more than one project can be stored in a SVN repository and a copy operation can only be executed on all of them together. The Apache Software Foundation for example stores all their projects in one repository and therefore, a huge amount of data has to be transferred. Hence, this approach is even worse.

Nevertheless, there is a possible solution. If all the additional requests for the files are separated and performed using asynchronous calls, the time consumption is probably reduced significantly. But since the Git mining was already working pretty fast and all the needed repositories are available using Git, the focus was not laid on this task. However, if in the future SVN data is needed, this approach is definitely worth trying.

JIRA Mining

The mining of the JIRA data works through RESTful web service calls that return JSON formatted information. The needed data are all the issues with their change log. There is a web service to get all this data, but the maximal number of issues in one call is restricted to 100. Therefore for large-scale projects 30 calls or more is no rarity which shows the importance of asynchronous calls. At the beginning the first 100 issues are fetched. With the results also the number of existing issues is delivered. The total number of issues serves as version number and is compared the same way as the Git and SVN version. If a new version of the data is available, it is calculated how many calls are needed and a counter is set. Additionally a method that buffers the first 100 results in an array and decrements the counter by one is called. Only the issues that are updated in the timespan between the start date in the configuration and the actual time are buffered, the rest is dropped. This leads to possible issues in the list that are created before the start date, but ignoring them does not make sense because events in the timespan after the start date then also

lack. As a next step, the asynchronous calls to receive all the issues are performed. All of this calls buffer their result in the same way as the first one. When the response of the last call is handled the counter is reaches zero and indicates completion of the task.

As a next step the linking of the commits with the issues is performed. Therefore, it is necessary to have the mining of the version control repository finished before the start of the JIRA mining to use actualized data. For every issue all the commits fetched from the database are investigated to see if the issue specific regular expression is true. For the first issue of the Apache Camel project the regular expression is *.*CAMEL-1[^0-9].**. In this case *CAMEL* is the JIRA key of the project and *1* is the number of the issue, combined by a hyphen building the key of the issue. .* at the beginning and the end of the expression means that every character irrelevant how many times used (also no character at all) can be there. *[^0-9]* implies that after the number of the issue no digit is allowed, what protects to find for example *CAMEL-12* as a result of the regular expression of *CAMEL-1*. This analysis however needs much time and is only needed to avoid false positive results. Therefore every commit is first checked if the issue key is contained in the message to filter most of the commits in a fast way. Then only the positive values are really checked on the complete regular expression. Having thousands of commits that all have to be checked for every issue, this approach saves much time. All the resulting linked commits are stored in an array and added to the specific issue.

Beside the commits no data is added to the issues. Also no information is deleted to allow the use in the other parts of SQA-Pattern even though in the current approach most of it is not needed. The issues are then stored in the JIRA database collection creating a document for every issue. As a last part, the status information of the project for JIRA is updated which finishes the mining.

Jenkins Mining

Mining the Jenkins data is also performed through RESTful web services returning JSON formatted documents. The information of every build has to be gathered. For this reason first of all, a web service on the overview of the project has to be called, getting the number of the first build that is not deleted and the number of the next build that is used as version number. Unfortunately when a build is deleted, also all the information associated are cleared and can not be fetched any more. All the build numbers from the first not deleted to the next build number minus one are used for asynchronous calls on this builds. To analyse if all service calls have responded a counter is created. When these calls get a response they call a method to buffer the result and decrement the counter. All the deleted builds in between will return a HTTP status *404 - not found* indicating that they have to be ignored but still decrement of the counter is needed. Unfortunately there is a bug in the Play Framework not permitting to have a longer idle time than 120 seconds. Normally this is not a problem but the Jenkins server of Apache is badly slow and therefore this limit can sometimes be reached. As a workaround, the builds that use more than 120 seconds to response are also ignored. However, in the next version of Play this bug will be fixed and the workaround will not be needed any more.

The responded builds are analysed for their timestamp as the data of the other resources and only the desired data is buffered. Since the data of the builds needs no adjustment, it is stored in the database as soon as all the build responses are fetched. Finally, the status information for Jenkins is updated and the mining ends.

SonarQube Mining

Only three RESTful service calls of SonarQube are needed to gather the relevant data for SQA-Pattern. The resources service provides some general information on the project, the time machine

service returns the different metrics over time and the metrics service offers information about these metrics. First of all, the resources service has to be called for a JSON response with the SonarQube project key. Analysing the responded JSON document the date of the actual version can be gathered and is then used as version number. The whole operation is aborted if the date is the same as the one stored in the database.

The next step is the fetching of the time machine data. As additional parameters to the ones already used for the resource, the metrics and the start date that are defined in the project configuration are assigned. For some reason SonarQube, sometimes does not deliver the newest values if the end date is not defined and therefore the current date is transferred. As soon as a response is received the data is buffered for further revision. The metrics are fetched asynchronously at the same time and and the result also buffered.

When both requests are returned successfully, the data of the two calls is combined. The time machine data only has the SonarQube specific identifier of a metric and the list of values. The metrics data includes additionally the correct name of the metric and some information if the number is a percentage or a concrete value. Combining these datasets allows to interpret the data correctly.

In the end the combined data is stored in the database and as for every other resource the status information is updated.

4.5 Recognizer

In the recognition part of SQA-Pattern, the patterns for searching violations can be defined, modified or removed. With these patterns the stored data of the miner is analysed. Last, the developer data is generated.

4.5.1 Pattern Definition Format

To analyse a project for violations, first of all some patterns have to be defined. The JSON format is used to transfer these pattern definitions. Therefore, the structure of the JSON document has to be defined to parse the data. In Listing 4.3, an example pattern definition is shown. The values *name* and *description* thereby only have informative character. The *severity* sets how fatal a found violation is and the *id* is used to clearly identify every pattern. The *type* is used to set the pattern category. This has an influence on the object in *definition*. The format of the definition for every category is different since the information needed depends on the category. An example for every category is shown in Listing 4.4 to 4.9.

```
{
   "name": "Example Name",
   "severity": "5",
   "description": "This is an example description",
   "type": "time",
   "definition": {
      ...
   },
   "id": "pat5335ad20e4b0f6e907f8de23"
}
```

Listing 4.3: Example pattern definition.

Regular Expression Pattern. The specific structure of a regular expression pattern is shown in Listing 4.4. The *type* inside the definition object defines which sort of events it is searched for. The value can either be *commit* or *issue*. The variable *field* is used to concretise in which text the search has to be performed. It can be *message* or *author* in a commit event or *summary* or *description* in an issue. The regular expression searched for, is stored under *regex*. The field *ruletype* specifies if a violation occurs when the expression is found or the very reverse, by setting the value to *regex* or *complementregex*. It is possible to have multiple objects in the *rules* array. That allows to search in different fields by adding another rule, the event then must meet both of the rules to count as violation. The pattern in Listing 4.4, for example, searches for all the commits not containing *CAMEL-* or *Merge* in their message.

```
"type": "regex",
"definition": {
   "type": "commit",
   "rules": [
      {
         "field": "message",
         "ruletype": "complementregex",
         "regex": ".*CAMEL-.*|.*Merge.*"
      }
   ]
}
```

Listing 4.4: A regular expression pattern to search for commit messages not containing *CAMEL-* or *Merge*.

Time Pattern. The pattern of the category *time* in Listing 4.5 searches for a resolved issue and all its linked commits 86400000 ms (1 day) after. Thereby the value in *type* is the anchor event. Supported are all the keys in Table 4.1 expect *commit_sameauthor*, *commit_sameissue*, and *commit_sameauthorandissue* since they make no sense as origin. The two interval variables define the timespan in ms from the anchor event in which the events listed in the *filters* array is searched. The value of *negativeInterval* thereby refers to the time before the event and *positiveInterval* stands for the time afterwards. All the keys in Table 4.1 can be chosen in the *filters* list. The only constraint is that if *commit_sameauthor* is used the anchor event has to be an issue or a commit and if *commit_sameissue* or *commit_sameauthorandissue* is used the anchor has to be an issue.

```
"type": "time",
"definition": {
   "type": "issue_resolved",
   "negativeInterval": 0,
   "positiveInterval": 86400000,
   "filters": [
      "commit_sameissue"
   ]
}
```

Listing 4.5: A time pattern searching for commits to an issue one day after its resolve.

Cluster Pattern. A maximal *interval* in ms the events have to be in is defined in a cluster pattern definition. Every rule in the *rules* array sets for one event type the minimal number of occurrences

Table 4.1: The keys for the different events used in the pattern description.

Key	Event-type	Explanation
commit	Commit	Any commit.
commit_sameauthor	Commit	Commits of the same author.
commit_sameissue	Commit	Commits that are linked to the same issue.
commit_sameauthorandissue	Commit	The combination of *sameauthor* and *sameissue*.
build	Build	Any build.
build_successful	Build	A successful build.
build_failure	Build	A failed build.
issue_created	Issue	The creation of an issue.
issue_resolved	Issue	The resolve of an issue.
issue_closed	Issue	The closure of an issue.
issue_reopened	Issue	The reopening of an issue.
issue_updated	Issue	Any update of an issue that is no status update

to be marked as a violation. The individual rules are linked using an AND operation. In the example of Listing 4.6, the pattern hits if there are more than 4 issue closures in 5 minutes (300000 ms). The *type* of the rule has to be a value of the Table 4.1. However, the keys *commit_sameissue* and *commit_sameauthorandissue* are not supported since for this purpose a different analysis is needed that is offered in the issue contribution pattern. In the current version of SQA-Pattern only the greater than *operator* represented by the string *more* is provided. Last the *operand* defines the limit of events.

```
"type": "cluster",
"definition": {
   "interval": 300000,
   "rules": [
      {
         "type": "issue_closed",
         "operator": "more",
         "operand": 4
      }
   ]
}
```

Listing 4.6: A pattern of the category cluster searching for more than 4 closures of issues in 5 minutes.

Issue Contribution Pattern. Specifying an issue contribution pattern includes the definition of an *interval* timespan in milliseconds that is the maximal time that can lay in between the single events to still count as violation. To ignore the value, it can be set to -1. Under the key *sameAuthor* a boolean is expected to mark if only commits of one author count for the pattern. As for the

cluster pattern only *more* is supported as *operator*, meaning greater than related to the limit in the variable *operand*. In Listing 4.7, an example issue contribution pattern is presented that searches for all issues that have more than one commit linked to it.

```
"type": "issue contributions",
"definition": {
   "interval": -1,
   "sameAuthor": false,
   "operator": "more",
   "operand": 1
}
```

Listing 4.7: An issue contribution pattern finding every issue with more than one linked commit.

Event Pattern. To define an event pattern only the value of the variable *type* has to be set. Possible are all the keys of the Table 4.1 expect *commit_sameauthor*, *commit_sameissue* and *commit_sameauthorandissue*. An example definition is shown in Listing 4.8.

```
"type": "event",
"definition": {
   "type": "issue_reopened"
}
```

Listing 4.8: An event pattern for all reopenings of issues.

Metric Pattern. A metric pattern is specified by setting an arbitrary number of *rules*. In Listing 4.9 an example is shown with the only rule that if the metric *Rules Compliance* falls below 80% the pattern hits. The value of *metric* thereby has to be the key of a SonarQube metric with the term *met* in front. The meaning of this additional term will be explained in Section 4.6.4. The *operator* is set using *less*, *equal* or *more* to define how the number of *operand* has to be interpreted.

```
"type": "metric",
"definition": {
   "rules": [
      {
         "metric": "metviolations_density",
         "operator": "less",
         "operand": 80
      }
   ]
}
```

Listing 4.9: A Metric pattern searching for a rules compliance below 80%.

4.5.2 Pattern Configuration

These pattern definitions are uploaded in the body of a PUT web service call. Since every pattern is project specific the call needs the key of the project as a parameter. On uploading the pattern, it is only checked if the JSON document includes the values *name*, *description*, *defintion*, *type* and *severity*. If the provided data is really in the correct format, is checked during the analysis (Section

4.5.4). An *id* does not need to be included in the JSON document, it is generated. However, the id must be provided to update an already existing pattern. In this case the existing pattern with the same id is overridden.

There are also RESTful web services to return the existing pattern definitions. All pattern of a project can be requested by only handing over the project key. By also including the id of a pattern, only this specific pattern will be returned. In the same way an existing pattern can be deleted with the only difference of setting the HTTP method to *DELETE*.

4.5.3 Database Structure

The database of the recognizer is structured in the same way as the database of the miner. There is a collection to store the status information for the patterns and the developers of a project. The collections storing the resulting data can be reached with this documents. In Listing 4.10, an example pattern status information is shown. Thereby, the different *versionOf* variables show the versions of the resources that was used for the analysis. All the pattern ids that were checked during the last run are stored in the array *checkedPatterns*. The remaining variables are used in the same way as in the mining part explained in Section 4.4.2. This guarantees access to the up-to-date data without directly deleting the old one and prevents running two analyses at the same time. The structure of the status information for the developers looks the same except that there is no *checkedPatterns* array. Also, the versions of Jenkins and SonarQube are not stored because that information is not used for the developers, instead the version of the pattern analyses is stored.

```
{
   "versionOfJenkins": 1758,
   "versionOfJira": 7307,
   "versionOfSonar": 1394506322000,
   "versionOfGit": 1395903889000,
   "checkedPatterns": [
      "5328902be4b0e93355709fb9",
      "531cf2bce4b099a5856c56f9"
   ],
   "type": "Pattern",
   "version": 1396612812891,
   "timeOfVersion": 1396612820448,
   "inUse": false,
   "startOfUse": 0
}
```

Listing 4.10: An example status information document for the pattern analysis.

4.5.4 The Analyses

The analyses for the patterns and developers are started through a RESTful web service. As in the miner first of all it is checked if another instance is running the analysis and in this case a HTTP status 423 - *locked* is returned. Otherwise, the data is locked itself and the analysis is started through an asynchronous method call, a HTTP status 202 *accepted* is returned. To save space at this point, also a method is called to delete all the results older than a day except for the current version.

Pattern Analysis

The firsts step of the analysis consists of caching the data of the different resources and adapting the format to allow a pattern analysis. The version control and the Jenkins data, does not have to be adapted because a document is stored for every event. However, the JIRA data is stored depending on the issue, not the actual event. Therefore, the creation and every event of the change log have to be stored separately with the corresponding issue information, e.g. the issue key. Additionally, the author of every event has to be set, which is not necessarily the author of the whole issue. Since the SonarQube data is sorted according to the metrics, the values of every timestamp have to be combined to an event.

Having all the data in the correct format, the analysis for every stored pattern can begin. Since only the structure of the pattern description document is checked, this process can throw an error. Thus, the errors are caught and a pattern whose analysis has thrown an error is marked in the description document and not checked any more in further analyses until the mark is removed.

All patterns have in common that the results are stored as intervals having a start and end timestamp. An example is shown in Listing 4.11. These timestamps are set by analysing the events contained in a result and taking the first and the last one as the borders. The events of a pattern result are all the events that are part of the violation. If the pattern result only contains one event, the start and end timestamps are the same. Results of the same pattern definition with overlapping borders are combined into one result if the focus is the same. A focus is for example used in the issue contribution pattern because a combination between results concerning different issues makes no sense. The *weight* of a result shows how many different results are combined in it. Therefore, the weight is first always set to 1 and summed up if the results are combined. The field *patternId* contains the id of the pattern definition that is used to find the interval. The *id* uniquely identifies the instance over all results and is built by adding *ins* to the pattern id at the beginning and a count of all the results of this pattern definition at the end.

```
{
   "endTimestamp": 1203120575000,
   "events": [
      {
         ...
      }
   ],
   "focus": "issDOXIA-56",
   "id": "ins5335ad6ce4b0f6e907f8de241",
   "patternId": "5335ad6ce4b0f6e907f8de24",
   "startTimestamp": 1142211142000,
   "weight": 7
}
```

Listing 4.11: An example pattern result interval.

In the following the specific analysis for every pattern category is described. Thereby the pattern definition structures of Section 4.5.1 are referred.

Regular Expression Pattern. To analyse a regular expression pattern, first of all it is checked in the definition if the type is a commit or an issue. Then, the regular expression is searched on the defined text for every convenient event and every rule. Since the rules are associated with an AND operation the analysis for one event is aborted if a rule is not met. If all rules are fulfilled an interval is created.

Time Pattern. To analyse a time pattern for every event that is matching the anchor type, the defined interval around it is scanned for the wanted events. If there are any, the interval is built with all the events, setting the anchor event as the focus.

Cluster Pattern. The cluster pattern results are found by generating for every event an interval with the given maximal timespan. All the events inside the borders are added. For every rule it is then checked if the minimal number of specific events fall into the interval. If this is the case, all the events that are not part of the cluster are removed. If one of the rules is *commit_sameauthor*, the author of the first event is used for the analysis and as the focus of the result.

Issue Contribution Pattern. For the issue contribution pattern analysis, all the linked commits of an issue are searched for clusters exactly in the same way as the cluster pattern. The only reason these two patterns are not combined is the following: The information which commits are linked to an issue is stored at a different location, thus making the search differ. The creation event is stored in the events of the interval. However, it is ignored while setting the borders of the interval and a field *responsible* is created, where the assigned developer is set, because the one who created the issue is not responsible for the violation of this pattern.

Event Pattern. The analysis of an event pattern is relatively easy. The events are just checked for the defined event type and the matching ones yield the result. The only slightly special case is the search for reopened issues because there the developer that resolved the issue is taken as responsible and not the one that actually reopened the issue.

Metric Pattern. For the metric pattern analysis, all the rules of the definition are checked with the corresponding metrics at one timestamp. If all rules are fulfilled, the event is violating the convention. Combining all the neighbouring events that matched leads to the resulting interval.

Developer Analysis

The developer analysis connects the authors of commits and issues and calculates the involvement of these developers in the pattern results. The analysis is directly started after the pattern analysis since an update in the pattern results always needs an update of the developers analysis. The analysis can also be started using a web service. However, this is mainly for testing purpose it does not really make sense to use it otherwise.

First of all, the version control data is analysed for all the developers that committed a revision. For every developer all the ids of his commits are stored. Additionally, the number of recent commits is saved, referring to all the commits in the week before the last update of the version control data.

Next, the bug tracker data is analysed to find all the users and the issues they are involved in. The involvement is counted for every issue a developer created, initiated a change to, or is assigned to. The keys of these issues are stored in the document of every developer. Additionally the developers are directly combined with the ones from the version control data by comparing the names. It was thought of using the email as connection but SVN does not provide an email address. Also for Git, the name leads to better results because some developers used multiple email addresses.

As the last part, all the events listed in the patterns are investigated to find the patterns every developer is involved in. Normally, the author of an event is taken as the involved developer, however sometimes, e.g. in the issue contribution pattern, the previously as responsible stored author is the one who counts. For every developer, all the instances of the different patterns plus

the recent ones he is involved in are stored. How often a developer is involved in one instance of a pattern is ignored. The number of instances for every pattern allows to calculate the summed-up severity rating as well as the recent severity rating for a developer.

In Listing 4.12, the structure of the resulting information for a developer is shown with some fake data.

```
{
   "commits": [
      "comcommitsha",
      ...
   ],
   "displayName": "Firstname Lastname",
   "id": "devfirstnamelastname",
   "issues": [
      "issEXAMPLE-1"
   ],
   "numberOfRecentCommits": 14,
   "overallSeverity": 22,
   "patternMatches": [
      {
         "name": "Patternname",
         "numberOfMatches": 11,
         "patternId": "pat5335ad6ce4b0f6e907f8de24",
         "severity": 2
      }
   ],
   "recentOverallSeverity": 10,
   "recentPatternMatches": [
      {
         ...
      }
   ],
   "usedEmails": [
      "example@example.com"
   ]
}
```

Listing 4.12: The structure of a stored developer information.

4.5.5 Adding Pattern Categories

New pattern categories can be added with only one simple change in the existing code. The key of the new category has to be added into the if-statements of the class *PatternFactory* to instantiate the new class that performs the analysis for the category. This class has to be a subclass of the abstract class *PatternChecker* and therefore, has to implement the method *checkPattern()*. Afterwards, pattern definitions of the new category can be added.

4.6 Service

The service part of SQA-Pattern is responsible for the provision of the resulting data of the minings and analyses. However, it is not just the forwarding of the information stored in the database since some has to be combined, restructured or filtered to deliver useful information. The data is delivering through RESTful web services. A summary how the different services can be accessed is listed in Appendix A.

4.6.1 Summary

The summary service lists all the data mined from the different resources. The data for every resource is sorted in ascending order depending on their timestamp. The only exception is the data about the metrics, which is categorised by the different metrics. Since, depending on the project, there is too much data to deliver, only the really important information is provided. The commits include only the *sha*, the *author* and the *timestamp*, the builds consist of the *number*, *result*, and *timestamp*. The filtering is a bit more complicated for the issues, because in addition to the *key*, *type*, *priority*, and *summary* also all the timestamps of the changes and the type of every change has to be summarised. The metric data is already reduced to the important information and therefore no more filtering is needed. In addition, the defined *starttimestamp* is provided to show if the data is cut. Listing 4.13 shows the structure of the summary of the mined data for illustration.


```
{
   "data": {
      "builds": [
         {
            "id": "bui157",
            "result": "SUCCESS",
            "timestamp": 1388643875000
         }
      ],
      "commits": [
         {
            "author": "Firstname Lastname",
            "id": "com8138f2fd6f358d973c62be9e57aacdf47927d541",
            "timestamp": 1130808153000
         }
      ],
      "issues": [
         {
            "id": "issEXAMPLE-1",
            "priority": "Blocker",
            "summary": "Issue summary",
            "timestamps": [
               {
                  "timestamp": 1113525993000,
                  "type": "Created"
               }
            ],
            "type": "Bug"
```

```
            }
        ],
        "metrics": [
            {
                "data": [
                    {
                        "timestamp": 1296674161000,
                        "value": 100
                    }
                ],
                "id": "mettest_success_density",
                "name": "Unit tests success (%)",
                "percent": true
            }
        ]
    },
    "options": {
        "starttimestamp": 0
    }
}
```

Listing 4.13: The structure of the summary of the mined data.

4.6.2 Pattern Results

The pattern results provided in this service are a combination of the pattern definition and the actual results. In Listing 4.14, the structure is illustrated. The information on the pattern definition is delivered, e.g. *name* or *type*. Additionally in the *instances* array the intervals are shown, without the events of every instance to save time and space. In *lastRun* the time of the version is provided if the pattern is already checked and has no error, else -1 is delivered.

```
{
    "name": "Patternname",
    "severity": "5",
    "description": "A description",
    "type": "time",
    "instances": [
        {
            "endTimestamp": 1132797259000,
            "focus": "",
            "id": "ins5335ad6ce4b0f6e907f8de250",
            "startTimestamp": 1132794164000,
            "weight": 1
        }
    ],
    "lastRun": 1397747074034,
    "id": "pat5335ad6fe4b0f6e907f8de25"
}
```

Listing 4.14: The structure of the one pattern result of the returned array.

4.6.3 Developers

The service to deliver the results of the developer analysis is relatively simple. The data on the different developers is just fetched from the database and handed on. Therefore the resulting information is just an array of the data presented in Listing 4.12.

4.6.4 Detailed Information

Since not all information is provided in the summary and pattern results services, an additional service is needed to deliver this data. To recognize which information has to be delivered, the id is needed as a parameter. This id is built of two parts, the first three characters are two determine what sort of information it is and the rest is the resource specific id. Thereby *com* stands for a commits, *iss* for an issue, *bui* refers to a build, *met* to a metric, *dev* to a developer, *pat* to a pattern, and *ins* stands for an instance of a pattern. The id is analysed upon a call to know which information is needed. Because the metrics data is already fully provided, a request is not supported in this service. A developer request, however, is supported even though all information is provided in the developer service to support requests on the data of just one developer. On a request of build or developer data, the information is just handed on from the database. If an issue is desired also the whole data of the database is returned, but the summary of the timestamps is added, generated in the same way as in the summary service. For a pattern request, the results and descriptions are combined as in the pattern results service. In the provision of a specific instance of a pattern result are now all the belonging events. Only the returned information on commits is still not complete. Since there can be many different rather big files associated, only the paths before the change and after the change are delivered as Base64 encoded strings. The Source code service is used to receive these files.

4.6.5 Source Code

The source code service delivers a specific file before and after a commit. Therefore, the id of the commit and the path of the file (Base64 encoded) have to be given as parameters. Since the file can be generated or deleted or the path can be modified, additionally the information if the specified path is the one before or after the commit has to be provided as a boolean, where zero points to the old path and one to the new one. If the specific file the path links to is found the whole file before and after the commit is returned. In the database of Git, however, only the new file and a *diff* information file are stored. The old file is generated based on this *diff* file using DiffUtils[7], a Java library to calculate differences between two files.

4.6.6 Other Services

There are two additional small services. The first one just returns a list of the newest commits to a project. How many commits that are returned maximally, is passed as parameter of the call. The other service is an overview on the pattern results. It is quite similar to the pattern results services but instead of the detailed instances just the number of instances as well as the number of recent instances for every pattern is returned.

[7] https://code.google.com/p/java-diff-utils/

4.7 Security

In SQA-Pattern a security system is implemented. Every service call is accepted only if a correct session token is delivered in the header. The required header name is *X-AUTH-TOKEN* with the session token as value. First of all, the login service has to be called using POST as HTTP method to get this token. The body of the call has to be a JSON formatted document with the user name and password such as in Listing 4.15.

```
{
   "username": "ExampleUsername",
   "password": "ExamplePassword"
}
```

Listing 4.15: The login format.

All three parts of SQA-Pattern store their own login information to give the possibility of users that can not change anything on a project setup or can change only the pattern definitions. However using the same database to store the users and their session token, also a common login is possible.

Chapter 5

Evaluation of SQA-Pattern

In this chapter the evaluation of the SQA-Pattern approach is described. Since there is no comparable data of violations to project conventions and on how much time the analysis takes without supporting software, a comparison can not be done. However, what can be performed is an analysis of rather small, medium and large-scale software projects to answer the research question: *How efficiently can violations of project conventions, such as missing issue keys in commit messages, be automatically detected in software projects?* The projects have to be mined and analysed for patterns as fast as possible. At least a nightly analysis has to be performed to have actualized pattern results that can support software managers or developers on their daily work. The resource data has to be updated before the pattern recognition can be executed. This implies that the time used for the mining and the pattern analysis have to be summed up. The combined duration, especially for the large-scale projects, must not exceed six hours such that it can be run over night. However, since developers probably prefer real-time data, it is ideal to have the complete data analysed at least in around ten minutes.

Additionally, large-scale software projects must not need exponentially more calculation effort than small scale projects. A maximally linear increase is desired. This is to avoid that growing projects once gain a size where the analysis cannot be performed in reasonable time any more.

The evaluation chapter includes the set-up of SQA-Pattern including the decisions which resources to evaluate and the technical circumstances. Then, the execution times are presented. The chapter closes with the discussion on how the results can be interpreted and whether the goals of the evaluation are met.

5.1 Study Setting

In this section, it is discussed which projects were chosen to mine the data and for which reason. Furthermore, the analysed work patterns are described and the consideration behind these patterns are explained. Finally, the hardware of the system is described to gain insight on how the effective results can be classified.

5.1.1 Source Projects

Nine projects of different sizes were chosen for the evaluation (Table 5.1). It is not that simple to judge whether one project is bigger than another, especially because it depends on the focus of the analysis. A project with many lines of code does not automatically indicate more functionality than another with less lines. Often the size of a project is calculated in combination with the cost for a project. However, the crucial point of this evaluation is neither the functionality of the

software nor the development costs. The critical point is that the SQA-Pattern approach still has to perform with projects having much data. For the Git repository, for example, three variables influence the performance: the number of commits, the sizes of them, and the size of the repository. Also not only the version control repository indicates the amount of analysis but it depends also on the number of issues, builds, and the size of the time machine data of SonarQube.

Table 5.1: The projects used for the evaluation sorted by their size (lines of code).

Project	Lines of Code	Lines	Commits	Issues	Builds	Metrics Data
Tomcat Maven Plugin	6530	12264	548	259	1	9
Maven Plugin Tools	7372	13765	797	245	5	15
Directmemory	7629	12546	640	137	1	13
Doxia	40037	71748	1527	463	4	29
Empire-db	41775	70479	804	204	15	30
Abdera	49228	79695	1491	299	3	15
Sling	149100	266683	9333	3453	4	26
ActiveMQ	169198	304112	7823	4918	2	64
Jackrabbit	229255	454734	7926	3663	2	42

However, the number of commits and issues is irrelevant for the current size of the project. Of course, the size of the software project was not chosen to exactly meet the requirements of the analysis, otherwise the result of the evaluation is already evident. Therefore, the probably most trivial sizing measurement was chosen: the non commented lines of code. Nevertheless, to avoid unusable data due to big differences in the data, projects were chosen that have at least a similar number of commits and issues. Because if, in the extreme example, an already finished project is committed the history is lacking and therefore the result is falsified due to the strong focus on the history data. Anyway, if the tools are used during the development, larger projects normally also have more data in the issue tracker or in the continuous integration platforms.

Another important point is the connection to the different tools. For the evaluation only projects of the Apache Software Foundation were used to have comparable results. Even though the connection is relatively slow, it is the same for all the projects. The software at Apache ranges from projects with only 3000 lines of code to bigger ones with more than 400'000. However, not all the projects were suitable because sometimes other tools than the ones supported in SQA-Pattern are used. Nevertheless, three small projects with circa 7000 lines of code were found relatively easy because there are many projects in this range. The medium sized projects were already harder to find, three projects were chosen with a range of lines of code from 40'000 to 50'000. The large-scale projects were the hardest to find. The size of the chosen projects ranges from 150'000 to 230'000 lines of code. In Table 5.1, the different project sizes are shown as well as the size of the available resource history. It can be seen that mostly at maximum five builds are available. This is because the Jenkins server always deletes older builds which makes it impossible to get more results. The *Metrics Data* shows the number of data points available for the SonarQube metrics. Another thing that must be pointed out is that although the medium sized projects have six times more lines of code than the small sized, the number of commits and issues is only two times bigger or in the case of Empire-db even in the same range as the small projects. Probably also the

number of lines have more influence on the result than the lines of code because for the mining and pattern analysis the effective code is not important. Nevertheless, the number of lines are just listed for information purposes since using this value as project size makes no sense.

5.1.2 Pattern Analysis

In the evaluation, not only the mining of the data from the resources was measured but also the time that was needed to analyse this data for the work pattern and assign the activities to a developer. Therefore, some reasonable patterns had to be defined. As shown in section 4.5.4 there are many possibilities to create specific patterns to find violations in projects. For all the evaluated projects, three conventions were formed, and violations of this conventions were searched.

Missing Issue Keys

The first convention is: *All commits have to be linked to an issue using an issue key*. The projects of Apache all use this linking between JIRA and Git. They do not use further commands in the commit messages such as directly writing comments or changing the status of an issue through the commit message, but for this convention the linking using an issue key is sufficient.

```
{
   "name": "Commits without issue key",
   "severity": "0",
   "description": "none",
   "type": "regex",
   "definition": {
      "type": "commit",
      "rules": [
         {
            "field": "message",
            "ruletype": "complementregex",
            "regex": ".*DOXIA-.*"
         }
      ]
   },
   "id": "pat5335ad6fe4b0f6e907f8de25"
}
```

Listing 5.1: The regular expression pattern for the project *Doxia*.

The regular expression pattern was used to find the violations to this convention. Thereby, the definition had to be configured that the commit messages is searched for every occurrence of the regular expression *.*JIRAID-.** where *JIRAID* stands for the project specific identification of JIRA. All the commits where this expression is not found are fulfilling the pattern. As example the pattern used for the search in the Doxia project is shown in Listing 5.1. For the other projects, the same pattern was used but the *regex* value was changed to the JIRA identification of the particular project. The severity was set to zero because it has no influence on the calculation effort.

Frequent Code Changes

Through this linking of commits to issues, it is also possible to analyse how many commits to an issue are available. Issues with several commits can be an indicator for complex code problems

that can not be solved easily and for an issue that is probably held to general. Several commits in a short time period point to possible intersecting code changes. Besides it alludes that a code change did not really solve the problem.

```
{
   "name": "Issues with frequent code changes",
   "severity": "0",
   "description": "none",
   "type": "issue contributions",
   "definition": {
      "interval": -1,
      "sameAuthor": false,
      "operator": "more",
      "operand": 1
   },
   "id": "pat5335ad6ce4b0f6e907f8de24"
}
```

Listing 5.2: The issue contribution pattern of the evaluation.

Therefore, the convention was formed: *An issue is only allowed to have maximally one linked commit*. This convention is rather extreme, but appropriate for the evaluation, because the longer the maximal timespan is (in this case infinite) and the more results to a pattern are found the longer takes the analysis. Therefore, a strict convention guarantees that having much search effort and many violations is not slowing down the algorithm too much. To find these frequent code changes the issue contributions pattern was used, as seen in Listing 5.2.

Inappropriate Issue Resolve

In JIRA a resolved issue means that the code changes on this issue are finished and the problem is resolved or that the new feature is implemented. Therefore, all the commits on this issue have to be performed before the status changes. However, it is of course possible to commit changes afterwards and assign them to the issue, for example when one hour later it is noticed that there is a bug in the new code. When this bug is remarked theoretically the issue had to be reopened, only then the bug could be fixed and committed and at last, the issue status changed to resolved again. Independent of whether this reopening was conducted or not the resolve of the issue was to early, the code has to be controlled first to avoid changes directly after a resolve.

This leaded to the convention: *A resolved issue must neither be reopened in the next ten day nor have a linked commit*. Using the time pattern as seen in Listing 5.3 the occurred violations were found. Again ten days is a quite high value that prevents from underestimating the calculation effort.

```
{
   "name": "Commits on issues after resolve",
   "severity": "0",
   "description": "none",
   "type": "time",
   "definition": {
      "type": "issue_resolved",
      "negativeInterval": 0,
      "positiveInterval": 864000000,
      "filters": [
```

```
            "commit_sameissue"
        ]
    },
    "id": "pat5335ad20e4b0f6e907f8de23"
}
```

Listing 5.3: The time pattern of the evaluation.

5.1.3 Configuration

To obtain adequate results, some configurations for the resources and for the evaluation had to be set. First of all, Git was used as version control repository for all the evaluated projects. The reason is that for large-scale projects it was already evident that the SVN mining takes much more time. The gathering of the data for a large-scale Apache project needed up to six hours where the same process using the Git repository needed less than ten minutes. The reason for this difference can be seen in Section 4.4.3. Even though there is a possible solution to reduce the time consumption, the SVN mining was not needed due all the Apache projects are hosted on both repositories.

To have comparable results, the configuration for all the projects was defined the same way. This means the Git data was not filtered for a special branch (the master respectively the trunk branch with all the commits was used). Also no timestamp to start from was set (meaning all the available data is gathered for all the resources) and for all projects the same SonarQube metrics were used.

Unfortunately, the Apache projects do not always have the same servers for their JIRA and SonarQube instances. Therefore, also the time used for the connections to these tools depends on the destination server. The JIRA instance is hosted on the Apache Servers for seven projects, the two others are on the one of Codehaus. For SonarQube, only one project is not hosted by Apache but by SonarQube in their examples database called Nemo.

The code was separated as much as possible to have details on how much time is consumed in the application and at the cutting points the timestamps are saved. However, due to the asynchronous composition of the mining it was not possible to fully separate the different actions. Especially the web service calls cannot be separated from the analysis and buffering of the data. Therefore an application monitoring tool called New Relic[1] was installed and loaded within the Play framework. New Relic monitors the entire application showing response time, error rates, recent events, information on the Java virtual machine, and much more. The information was tracked during the evaluation and automatically uploaded to a web user interface where the data can be interpreted. Unfortunately, this tracking probably slows down the application itself but since it was the same for different projects it does not falsify the results. For the evaluation, only the features to analyse the web service calls had to be used. There, it can be seen how much time was used to get a response for every call.

With this information, it can be calculated exactly how much time was used for which operation. In detail for the Git repository, the start-up, clone, analysis and saving, deletion, and end-up time spans were stored. For JIRA the fetch, linking and saving time spans were calculated, additionally to the duration of the first and the maximal length of the following connections. Only the longest duration was stored because the calls were executed asynchronous and therefore the longest-lasting connection is the one influencing the complete duration. The Jenkins analysis included the fetch and saving timespan as well as the duration of the connection to the overview service and the maximal length for a response of the build details. Also for the last resource, SonarQube, the execution time of the three service calls was measured. For the second and the

[1] http://newrelic.com/

third type of calls again only the maximal length influences the complete duration. Furthermore, the fetching, combining, and saving timespan was stored.

These divided time spans did not all have direct influence on the evaluation results because of the asynchronous composition. However, it can be shown what needs most of the time of the mining and therefore, perhaps support a conclusion. Also outliers can be identified.

For the pattern analysis only the duration of the different patterns was measured and the time used to run them all. Even though only the time used for the whole run is important for the complete duration, it is interesting to see which pattern analysis needs which amount of time to run through. The developer analysis was performed on the results of all three patterns considering only the combined results are significant.

5.1.4 Hardware

The analysis was computed on a notebook with Windows 7 64-bit operating system. It was equipped with an Intel Core i7-3520M CPU with 2.9 GHz and 8 GB RAM. As a data storage device, a HDD was used. This set-up has less computing power than today's servers, guaranteeing that the result is not just reached through high-tech hardware. The internet connection used had an upload rate of around 75 Mb per second and a download rate of approximately 95 Mb per second. However this connections were probably not fully used since the Apache servers are rather slow.

5.2 Results

In this section the results of the evaluation are presented. The detailed outcomes for every part of the operations can be found in Appendix B. For every resource and pattern a LOESS (LOcal regrESSion) model [CD88] is added to show if the analysis running time rises linear or exponentially with respect to the number of inputs. LOESS is a locally weighted regression approach providing information on how data evolves in a scatter plot. Additionally, the combined results are analysed and it is shown how much each part of the application influences the overall outcome.

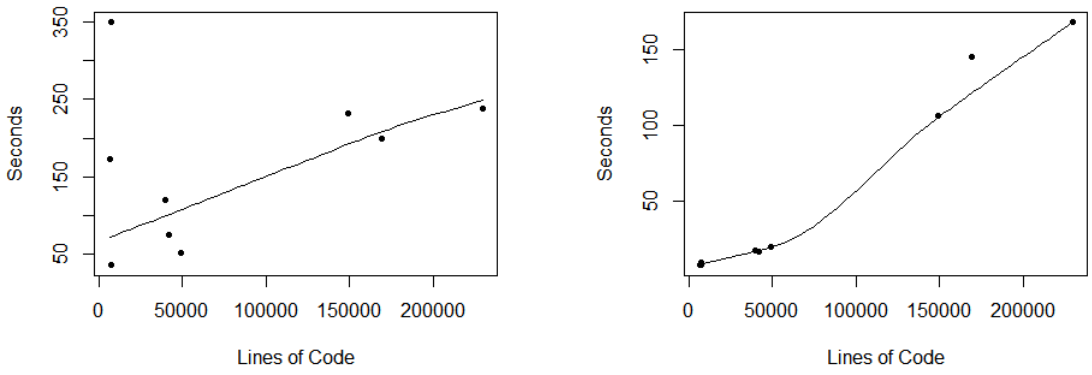
5.2.1 Git Results

The mining of the Git data lasted for the chosen projects maximally 5 minutes and 49 seconds (349 seconds) as seen in Table 5.2. The fastest mining was finished in only 36 seconds. Most of the time was used for the clone operations of the repositories, namely 66% in total. The results are presented in the scatter plots from Figure 5.1a to 5.2b in relation to the lines of code and the number of commits. The project that needed most time for cloning, Maven Plugin Tools, is one of the smallest. It took 5 minutes and 42 seconds (342 seconds) to clone its repository. The further analysis only lasted around 7 seconds which is comparable to the other small projects. Out of the data there can not be found a reason for this behaviour. The project has neither many commits nor lines of code. It is possible that Apache uses slower servers for some repositories or the data is in a difficult format to store and save, e.g. many very small files. Also the smallest project, Tomcat Maven Plugin, needed nearly as much time as the large-scale repositories due to a very long clone duration without an obvious reason, but this outlier is not that extreme.

To see the analysis time without possible effects of the server connection in Figure 5.1b and 5.2b the execution time is calculated without the time that is used for cloning. This has also the advantage that only code that is written as part of SQA-Pattern is taken into account since the

Table 5.2: The execution times of the Git minings.

Project	Git Mining (s)	Cloning (s)
Tomcat Maven Plugin	172.563	165.198
Maven Plugin Tools	349.490	342.271
Directmemory	35.930	26.698
Doxia	120.876	103.371
Empire-db	76.118	59.847
Abdera	52.136	32.750
Sling	232.411	126.656
ActiveMQ	199.481	54.902
Jackrabbit	238.087	70.470



(a) With the clone operations.

(b) Without the clone operations.

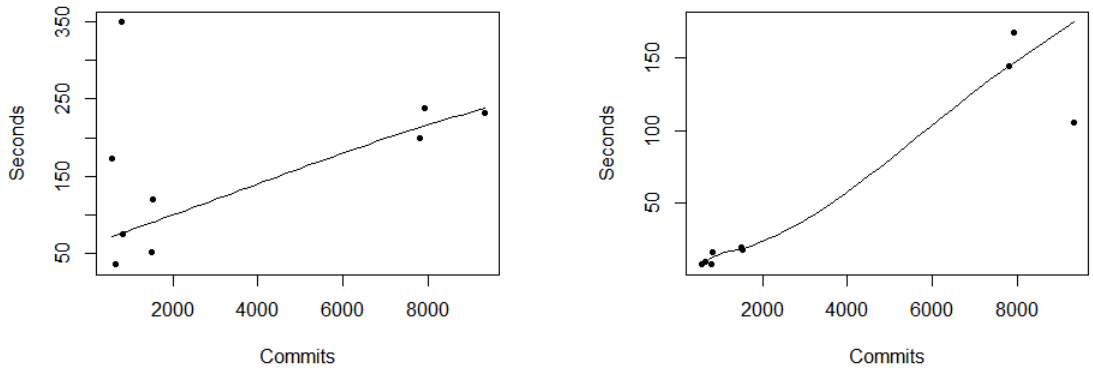
Figure 5.1: The execution times of the Git minings in relation to the lines of code.

cloning is just a functionality of JGit that is used unchanged. However, the cloning is a part of the mining, it can thus not be fully ignored.

The LOESS curve for all four diagrams shows that the calculation effort rises mostly linear compared with the lines of code as well as the number of commits. Since the data with the full time span has some outliers the curve is more accurate for the data without the cloning.

5.2.2 JIRA Results

Mining JIRA data is theoretically independent of the project size, however, bigger projects also tend to have more issues. But a direct analysis to calculate in which way the effort rises only makes sense based on the number of issues. The scatter plot is shown in Figure 5.3a. Unfortunately there are two servers used to host the JIRA instances. As it can be seen in Table 5.3, the two



(a) With the clone operationss

(b) Without the clone operationss

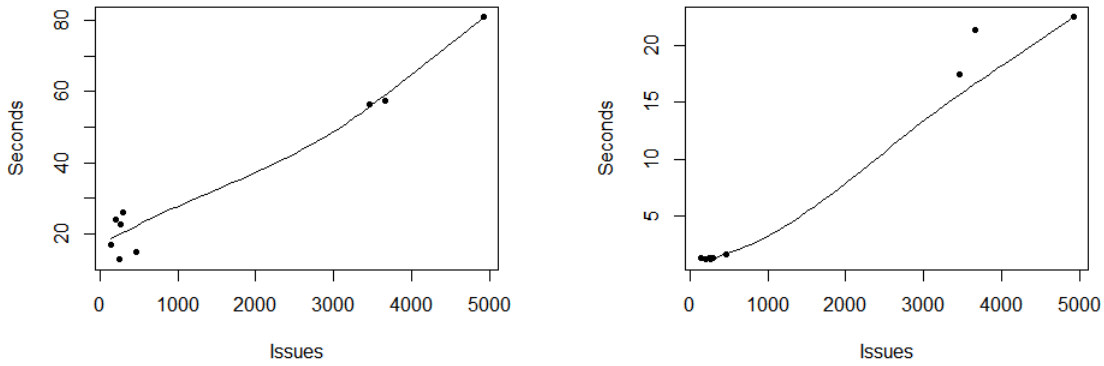
Figure 5.2: The execution times of the Git minings in relation to the number of commits.

fastest projects were Doxia and Maven Plugin Tools. They are both hosted by Codehaus, the rest by Apache itself. The responses from Codehaus lasted approximately half the time of the Apache responses. In Figure 5.3b the values are shown without the contamination of the server response time. The difference in the execution times is this big because, for example for ActiveMQ, the project with the biggest number of issues, 49 calls on the same time were executed resulting in 46 seconds waiting time for the last response.

Table 5.3: The execution times of the JIRA minings.

Project	JIRA Mining (s)	Waiting for Responses (s)
Tomcat Maven Plugin	22.727	21.60
Maven Plugin Tools	12.774	11.47
Directmemory	16.834	15.54
Doxia	15.055	13.40
Empire-db	24.185	23.00
Abdera	26.178	24.90
Sling	56.424	38.90
ActiveMQ	80.937	58.40
Jackrabbit	57.535	36.14

Again the LOESS curve is provided in both graphs. It can be observed that the curve looks quite linear, but it is hard to say because the small and the middle sized projects have nearly the same number of issues and therefore there is a large gap without data in between.



(a) With the connection times.

(b) Without the connection times.

Figure 5.3: The execution times of the JIRA minings in relation to the number of issues.

5.2.3 Jenkins Results

The results of the Jenkins mining are reported in Table 5.4. On average 99.5% of the time used was just the waiting for a response of the Apache Jenkins Servers because some of them are really slow. Some servers even needed up to 104 seconds for a response.

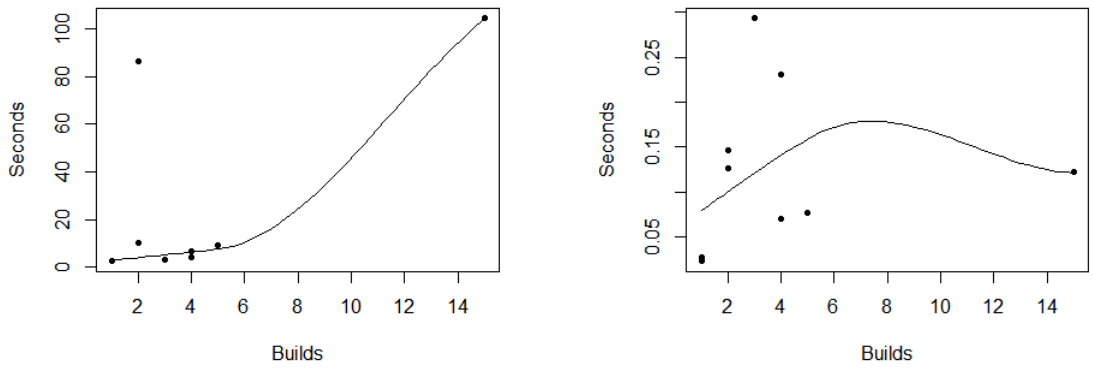
Table 5.4: The execution times of the Jenkins minings.

Project	Jenkins Mining (s)	Waiting for Responses (s)
Tomcat Maven Plugin	2.437	2.414
Maven Plugin Tools	9.447	9.370
Directmemory	2.721	2.694
Doxia	4.120	4.050
Empire-db	104.423	104.300
Abdera	2.943	2.650
Sling	6.770	6.540
ActiveMQ	86.207	86.060
Jackrabbit	10.427	10.300

In Figure 5.4a the complete execution times of the mining for Jenkins data is displayed compared with the number of builds. There are mostly only 1-5 builds for the projects because the older ones were deleted and are therefore not accessible any more. Empire-db, the project with 15 builds, needed more than 100 seconds to finish the mining. However, this was not influenced mainly by the number of builds but by the servers, since only one connection needed that long.

Therefore, the value must be treated as an outlier same as the result of ActiveMQ with a connection time of more than 80 seconds.

To avoid this server problem the duration is also calculated without the connections, illustrated in Figure 5.4b. Unfortunately this diagram is also not ideal because some of the projects with a fast connection to Jenkins are now outliers due to a longer calculation. This is probably because for the other projects the application can overlap some calculations with the waiting for the responses, what is not possible with a fast connection.



(a) With the connection times.

(b) Without the connection times.

Figure 5.4: The execution times of the Jenkins minings in relation to the number of available builds.

The LOESS curve in Figure 5.4a shows some exponential elements but if the highest values is omitted as outlier the result looks different. The other values raise in a straight line. The running times without the connections show no indication for a quadratic growth. Due to the few number of builds, the result has to be interpreted with care.

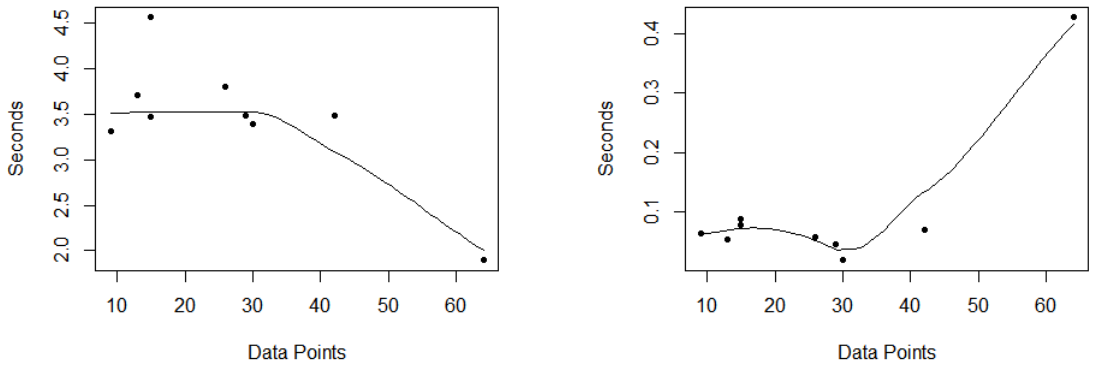
5.2.4 SonarQube Results

The results of the SonarQube mining are presented in Table 5.5. As it can be seen, the execution times are all between 3 and 5 seconds except for ActiveMQ. This is also the only project not hosted by Apache and therefore has a much faster connection. In the scatter plot in Figure 5.5a between the number of data points in SonarQube and the mining duration it can be seen relatively easy that it does not fit to the other results. Figure 5.5b depicts the durations without the connection times. There it is shown that the value is again an outlier this time in the other direction. This has, as for the JIRA results, nothing to do with a longer calculation, but with the work executed in the background during the waiting for a response, which can not be performed on a fast connection.

Considering the imprecision, the value of ActiveMQ has to be ignored in the LOESS calculation. Usually, LOESS tries to eliminate the impact of outliers itself but since the outlier is also the project with more data points than the others it is not adapted automatically. The other values in both cases add up to a straight line implying linear calculation effort.

Table 5.5: The durations of the SonarQube minings.

Project	SonarQube Mining (s)	Waiting for Responses (s)
Tomcat Maven Plugin	3.313	3.250
Maven Plugin Tools	4.567	4.490
Directmemory	3.714	3.660
Doxia	3.486	3.440
Empire-db	3.388	3.370
Abdera	3.477	3.390
Sling	3.808	3.750
ActiveMQ	1.892	1.465
Jackrabbit	3.490	3.420



(a) With the connection times.

(b) Without the connection times.

Figure 5.5: The execution times of the SonarQube minings in relation to the number of data points.

5.2.5 Analysis Results

In Table 5.6 the running times of the different pattern analyses and the search for all developers are presented. The first pattern is the analysis for missing issue keys, the second finds frequent code changes for an issue, and the third one searches for inappropriate issue resolves. The exact pattern definitions and their purposes are described in Section 5.1.2. The results under the label *Combined* are not the combination of the results for the three patterns, but the result of running them all in one go. Therefore, the result is lower than just the combination of the other ones since some commonalities were used. Overall, one can observe that the pattern recognition was much faster than the mining. Analysing the data for the three pattern and creating the developer data together maximally needed around 20 seconds whereas just the Git mining already needed up to 5 minutes. In Table 5.7 the resulting violations of the defined conventions are displayed. The table

Table 5.6: The execution times of the pattern and developer analyses.

Project	First (s)	Second (s)	Third (s)	Combined (s)	Developers (s)
Tomcat Maven Plugin	0.261	0.583	0.165	0.652	0.876
Maven Plugin Tools	0.456	0.752	0.164	1.008	1.163
Directmemory	0.188	0.645	0.113	0.732	0.567
Doxia	0.896	1.613	0.278	1.629	1.396
Empire-db	0.833	0.666	0.370	1.313	0.763
Abdera	0.523	1.628	0.238	1.753	1.321
Sling	7.533	7.849	3.518	11.248	8.955
ActiveMQ	5.226	8.216	4.435	9.857	9.153
Jackrabbit	7.648	5.727	3.577	11.693	8.672

Missing issue keys pattern as *First*, frequent code changes pattern as *Second* and inappropriate issue resolve pattern as *Third*.

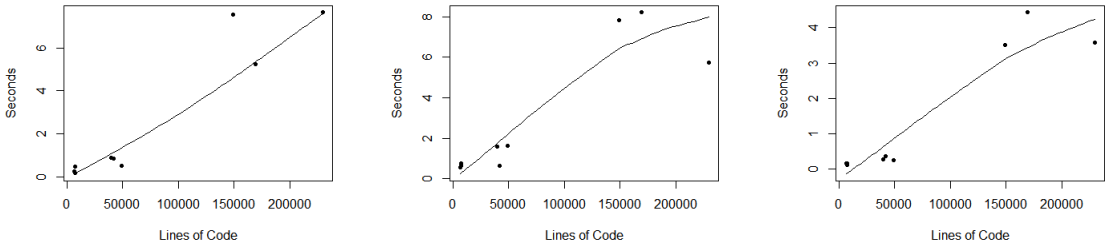
Table 5.7: The number of violations for every pattern and project.

Project	First	Second	Third
Tomcat Maven Plugin	404	23	0
Maven Plugin Tools	561	30	0
Directmemory	514	18	8
Doxia	1183	57	0
Empire-db	386	67	18
Abdera	1322	17	16
Sling	4107	940	367
ActiveMQ	4193	645	468
Jackrabbit	2858	719	454

Missing issue keys pattern as *First*, frequent code changes pattern as *Second* and inappropriate issue resolve pattern as *Third*.

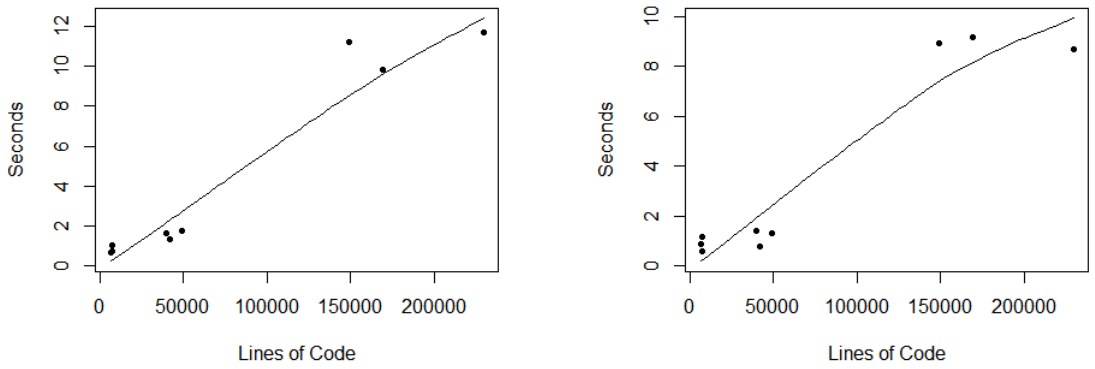
is provided for information purposes, in the efficiency evaluation these results have only a minor influence. However, it shows that there are pattern results, otherwise the developer analyses would be faster.

The LOESS regressions for the specific patterns in Figure 5.6 show quite linear results. The execution times of the pattern to find missing issue keys evolve linear. The execution times of the other two pattern even show logarithmic evolution.



(a) Missing issue keys. **(b)** Frequent code changes. **(c)** Inappropriate issue resolves.

Figure 5.6: The execution times of the analyses for the single patterns in relation to the lines of code.



(a) For all three patterns. **(b)** For the developers.

Figure 5.7: The execution times of the combined analyses in relation to the lines of code.

The running times of the combined pattern analysis in Figure 5.7a also tend to rise rather linear or logarithmic than exponentially. The same can be observed for the scatter plot of the results from the developers search in Figure 5.7b.

5.2.6 Combined Results

In Table 5.8 the complete execution times and the ones without the connection times and the cloning of the version control repositories are presented. Tomcat Maven Plugin and the Maven Plugin Tools still need much more time than expected. This is because the Git mining used roughly 70% of the whole evaluation time, presented in Figure 5.8. Therefore, the Git mining has a large impact on the overall results. This relation is not only because the cloning takes this much time: If the cloning and connection times to the resources are subtracted, only the execution time of the analysis remains where the Git mining even needed 77%. The bar chart also shows that the analysis of SonarQube and Jenkins is so fast that it has practically no influence. However, since the Jenkins results were always deleted and the SonarQube instances were not often

Table 5.8: The execution times of the complete minings and analyses.

Project	Complete Duration (s)	Without Connections and Clone (s)
Tomcat Maven Plugin	202.568	2.754
Maven Plugin Tools	378.449	3.638
Directmemory	60.498	2.688
Doxia	146.562	4.807
Empire-db	210.190	3.422
Abdera	87.808	4.745
Sling	319.616	38.026
ActiveMQ	387.527	42.139
Jackrabbit	329.904	41.978

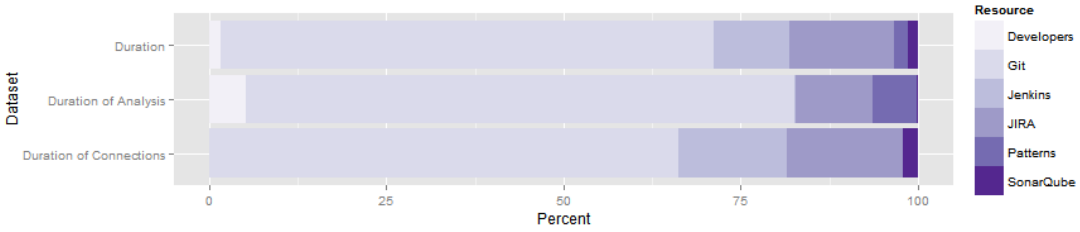


Figure 5.8: The percentage execution times of the operations.

updated this relation is not fixed. Some projects have 1000 Jenkins builds and more, having them not deleted results in much more effort.

Figure 5.9 reveals the ratio between the running times of the connections or the cloning to the running times of the analysis. At least 66% of the time was spent waiting only for the resource information. This problem can possibly be resolved by running the application on the same server as the resource information is on. An alternative would be to use at least resources with a faster connection than the ones of Apache.

Doing the LOESS regression on the resulting datasets shows no indices for an exponential raise of the durations for the complete mining in Figure 5.10a as well as for the values without the connections and cloning in Figure 5.10b.

5.3 Discussion

The evaluation has shown that with the SQA-Pattern approach the whole mining and analysis for patterns needs less than ten minutes even for large-scale projects. This is a quite good result because it definitely allows nightly updates of the data. It is even possible to provide nearly real-time data by checking the data, for example, every hour. Since there is no exponential growth in the running times, as far as it can be said due to the small sample size, it is possible to support even bigger projects without too much effort. The result is also rather fast compared with other

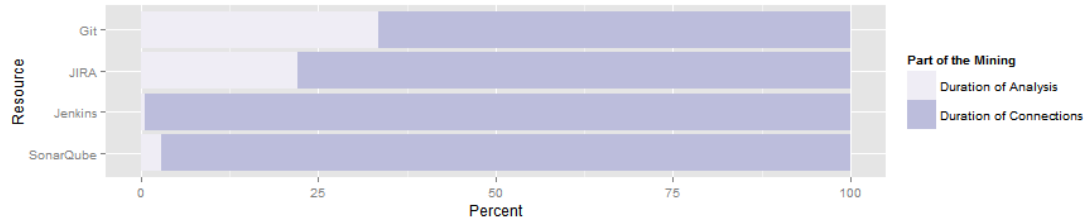
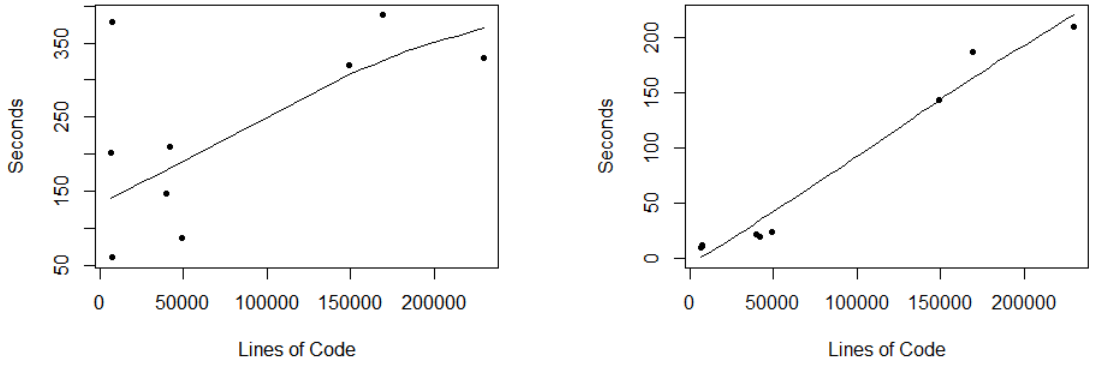


Figure 5.9: The percentage execution times of the analysis versus the connection times for each resource.



(a) With the time used for connections and cloning.

(b) Without the time used for connections and cloning.

Figure 5.10: The execution times of the whole minings in relation to the lines of code.

data mining in the literature. Troxler [Tro14], for example, needed up to one and a half hour to gather the data from JIRA or GitHub.

The evaluation has also shown where improvements are possible. Since much time is lost by waiting for a response, time can be saved by starting all the mining operations at the same time. This approach was not followed in this evaluation to generate more accurate results for the single resources. Additionally, in the current state of SQA-Pattern, the JIRA mining can not be performed before the Git mining is finished because the linking of the issues with the commits is executed in the same process. If the linking of the JIRA mining is uncoupled and performed separately afterwards, all the mining operations can be started at the same time, resulting in a better overall performance.

The Git mining needed approximately 70% of the whole time thus an improvement of the code is best located there. At the current state, each time the data gathering starts, the whole repository is cloned and deleted at the end. This eases the implementation of the prototype. Furthermore, at the beginning of the project, it was unclear which operations require which amount of time. The evaluation, however, has shown that 66% of the time for obtaining the Git data is only used for the cloning. Additionally, by deleting the repository after every usage some more time is lost. If the repository is cloned at the first mining and afterwards only updated much time is saved.

Generally, the approach of always taking the current data and ignoring the already mined probably has to be reconsidered. In the case of JIRA, the approach is supposedly correct because every issue can change and therefore have new data. However, the old data provided by Git, Jenkins, and SonarQube is still valid; it only needs to be supplemented. Therefore, a complete mining is only needed in the beginning. This also allows to store data of already deleted builds in Jenkins, which is possibly desirable.

The analysis of the pattern and developer data was rather fast in comparison with the mining since each of them needed around 1-2% of the whole time. The only concern is that the more pattern are analysed the higher the effort will be. However, the evaluation has shown that analysing all patterns together needed much less time than the combination of the single analysing runs. Therefore, the costs do not rise too strongly by adding new patterns.

Summarising, the research question of this thesis asking for the efficiency of a violations analysis can be answered as follows: The SQA-Pattern approach proofs that the mining and analysis to find violations of conventions in a project does not need too much time to allow real-time data even for large-scale projects. Although there is still room for improvements, the running time is already below ten minutes without indications for exponential growth of the duration in relation to the project size.

Chapter 6

Final Remarks

This chapter summarizes how the SQA-Pattern approach is integrated in the different fields of studies and which conclusions can be drawn from the evaluation. Additionally, it is discussed where still some drawbacks are and which future work can improve the approach.

6.1 Conclusion

In this thesis the proof-of-concept implementation *SQA-Pattern* is presented and evaluated. It is an application to mine the data of different continuous integration tools used in software projects and analyse it for violations of conventions about the usage of the tools. Six different pattern categories are defined in SQA-Pattern to find various types of violations. Additionally, the parameters of these patterns can be changed for every project to support project specific conventions. The challenges of implementing SQA-Pattern were, on the one hand, to allow flexible pattern adaption to support different conventions. On the other hand, the goal was to mine and analyse the data as fast as possible to provide up-to-date results even for large-scale software projects. In the evaluation, it was measured how long the SQA-Pattern approach needs to provide results for projects of different sizes. The focus was to answer the following research question: *How efficiently can violations of project conventions, such as missing issue keys in commit messages, be automatically detected in software projects?* Thereby, it was shown that the whole passage needs less than ten minutes even for large-scale software projects of more than 200'000 lines of code. Additionally, the evaluation reveals no evidence for an exponential growth of the execution time compared to the project size. This indicates that the approach delivers the violations also for larger projects in a comparable time range. Most of the time in the evaluation, namely 95.6%, was spent for the mining of the different resources. This shows where the focus has to be set for further work on the efficiency. Summarising, the SQA-Pattern approach has shown that a daily analysis is possible for violations of project conventions on the usage of different development tools. Even real-time updates are possible, however, this target can be met more accurate with some improvements of the algorithms.

6.2 Future Work

There are two directions in which the focus of future work can lay. The first is to improve the SQA-Pattern approach by speeding up the algorithm, adding new resources, or enabling the search of other kinds of violations. Secondly, it can be evaluated if the generated information on project violations improves the development of software projects in teams.

6.2.1 Improvement of SQA-Pattern

To advance SQA-Pattern, the development best focuses primary on the mining of the Git data, since this part of the approach needed around 70% of the whole execution time in the evaluation. Of this 70% of the whole time, the cloning of the repository required approximately 67%. However, the already stored data of Git stays valid because only additional commits are added. Therefore, a clone operation is only needed when a new project configuration is added, afterwards an update of the current state is sufficient to have the complete data. This update is probably much faster then a whole cloning and therefore a lot of time can be saved. Additionally, the mining of Jenkins data that takes around 11% of the time can be improved in the same manner. The already stored builds do not need an actualisation, only the new ones have to be requested. Therefore as a side effect, the data of builds that are deleted on the Jenkins platform still can be displayed.

Another possible improvement is the separation of the algorithm that links the commits and the issues. This allows to run all the minings at the same time and afterwards link the data. Thereby, the execution time is reduced since many threads are just waiting for responses.

Additionally, adding algorithms for the mining of similar resources as the ones already supported allows to analyse more software projects. With the current version, the combination of exactly the four supported tools is needed to fully analyse the data but in reality there are various other tools resulting in multiple different combinations. A first step in this direction can be to extend the support of SVN. This requires to improve the algorithm of the corresponding mining by asynchronously requesting the files of the different revisions. This will reduce the time needed for the SVN mining significantly, especially combined with mining the whole SVN data only on the addition of a project configuration as mentioned above.

To improve the search for violations, additional pattern categories can be implemented as explained in Section 4.5.5. For example, a sequence pattern was designed too late to include in the current approach but is promising to find further violations. It was thought of searching defined consecutive events to find actions that are performed in a wrong order. An underlying convention can be, for example, that after an issue is resolved, a build has to run successfully before the issue is closed. Furthermore, the approach can be improved by allowing to combine the already existing pattern categories.

6.2.2 Additional Evaluation

In the evaluation of the SQA-Pattern approach, the focus was set on how fast the data can be gathered and analysed. An additional evaluation where developers use the patterns to find violations can show if the data provided helps the stakeholders in their daily work, for example, with the SQA-Timeline [Bir14] user interface.

Appendix A

Web Services

Table A.1: The web services paths of the miner part.

Purpose	Path	HTTP method
User login	/login	POST
User logout	/logout	POST
Projects overview	/projects	GET
Addition of a Project Configuration	/projects	PUT
Get a project	/projects/:key	GET
Delete a project	/projects/:key	DELETE
Get the status information	/projects/:key/status	GET
Mine the Git data	/projects/:key/git/mine	GET
Get the unformatted Git data	/projects/:key/git/result	GET
Mine the SVN data	/projects/:key/svn/mine	GET
Get the unformatted SVN data	/projects/:key/svn/result	GET
Mine the JIRA data	/projects/:key/jira/mine	GET
Get the unformatted JIRA data	/projects/:key/jira/result	GET
Mine the Jenkins data	/projects/:key/jenkins/mine	GET
Get the unformatted Jenkins data	/projects/:key/jenkins/result	GET
Mine the SonarQube data	/projects/:key/sonar/mine	GET
Get the unformatted SonarQube data	/projects/:key/sonar/result	GET

:key has to be replaced by the project key.

Table A.2: The web services paths of the recognizer part.

Purpose	Path	HTTP method
User login	/login	POST
User logout	/logout	POST
Get the status information	/projects/:key/status	GET
Update the pattern data	/pattern/update/:key	GET
Get the unformatted pattern data	/pattern/result/:key	GET
Update the developer data	/developer/update/:key	GET
Get the unformatted developer data	/developer/result/:key	GET
Add/modify a pattern definition	/:key/pattern	PUT
Get all the pattern definitions	/:key/pattern	GET
Get a pattern definition	/:key/pattern/:id	GET
Delete a pattern definition	/:key/pattern/:id	DELETE

:key has to be replaced by the project key, *:id* by the pattern id.

Table A.3: The web services paths of the service part.

Purpose	Path	HTTP method
User login	/login	POST
User logout	/logout	POST
Get the summary	/:key/timeline	GET
Get the pattern results	/:key/patternresults	GET
Get the developers	/:key/developers	GET
Get the data of an item	/:key/info/:id	GET
Get a source file	/:key/source/:id/:filePath/:searchNewFiles	GET
Get the newest commits	/:key/commits/:numberOfCommits	GET
Get a pattern results overview	/:key/overviewpatternresults	GET

:key has to be replaced by the project key, *:id* by the id of an item, *:filePath* by the path to the questioned file (Base64 encoded), *:searchNewFiles* by 0 for the old path or 1 for the new one, and *:numberOfCommits* by the maximal number of commits.

Appendix B

Evaluation Results

		Tomcat Maven Plugin	Maven Plugin Tools	Directmemory	Doxia	Empire-db	Abdera	Sling	ActiveMQ	Jackrabbit
Data	Lines of Code	6530	7372	7629	40037	41775	49228	149100	169198	229255
	Lines	12264	13765	12546	71748	70479	79695	266683	304112	454734
	Commits	548	797	640	1527	804	1491	9333	7823	7926
	Issues	259	245	137	463	204	299	3453	4918	3663
	Builds	1	5	1	4	15	3	4	2	2
	Metrics datapoints	9	15	13	29	30	15	26	64	42
SonarQube	Fetch	3295	4536	3710	3478	3383	3471	3795	1863	3481
	Analysis	1	1	0	1	0	0	0	0	1
	Saving	17	30	4	7	5	6	13	29	8
	Complete	3313	4567	3714	3486	3388	3477	3808	1892	3490
	Resource request	1730	2720	1800	1530	1510	1610	1570	465	1550
	Timemachine request	888	1540	1370	1480	1660	1090	1570	1000	1290
	Metrics request	1520	1770	1860	1910	1860	1780	2180	434	1870
	Complete without requests	63	77	54	46	18	87	58	427	70
Jenkins	Fetch	2430	9388	2717	4109	104402	2898	6658	86119	10353
	Saving	7	59	4	11	21	45	112	88	74
	Complete	2437	9447	2721	4120	104423	2943	6770	86207	10427
	Overview request	2040	7950	2260	2740	1300	1570	2780	8460	8400
	Build request (max)	374	1420	434	1310	103000	1080	3760	77600	1900
	Complete without requests	23	77	27	70	123	293	230	147	127
Git	Startup	8	10	88	8	79	51	43	63	51
	Cloning	165198	342271	26698	103371	59847	32750	126656	54902	70470
	Analysis and saving	7158	6898	8837	16992	15727	18257	99412	140102	162282
	Deletion	190	300	293	488	453	1051	6200	4327	4052
	Endup	9	11	14	17	12	27	100	87	1232
	Complete	172563	349490	35930	120876	76118	52136	232411	199481	238087
	Complete without cloning	7365	7219	9232	17505	16271	19386	105755	144579	167617
JIRA	Fetch	22097	12102	16388	13909	23567	25373	39382	59120	37735
	Linking	91	158	107	240	211	266	8816	10376	11269
	Saving	539	514	339	906	407	539	8226	11441	8531
	Complete	22727	12774	16834	15055	24185	26178	56424	80937	57535
	First request	11300	5710	11600	5400	11500	12000	12500	12200	9540
	Following requests (max)	10300	5760	3940	8000	11500	12900	26400	46200	26600
	Number of following requests	2	2	1	4	2	2	34	49	36
	Complete without requests	1127	1304	1294	1655	1185	1278	17524	22537	21395
Analysis	Issue contribution pattern	261	456	188	896	833	523	7533	5226	7648
	Regular expression pattern	583	752	645	1613	666	1628	7849	8216	5727
	Time pattern	165	164	113	278	370	238	3518	4435	3577
	Combined patterns	652	1008	732	1629	1313	1753	11248	9857	11693
	Developers	876	1163	567	1396	763	1321	8955	9153	8672
Complete	Sum	202568	378449	60498	146562	210190	87808	319616	387527	329904
	Sum without cloning & requests	10106	10848	11906	22301	19673	24118	143770	186700	209574

Appendix C

Contents of the CD-ROM

The following files are stored on the CD-ROM:

- **Abstract.txt**
  Abstract
- **Zusfsg.txt**
  German abstract
- **Masterarbeit.pdf**
  Copy of this thesis
- **SQA-Pattern.zip**
  Source code
- **Evaluation-Results.xlsx**
  Evaluation results

Bibliography

[AV09] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, pages 298–308. IEEE Computer Society, 2009.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ, 1997.

[Ber12] Alan Berg. *Jenkins Continuous Integration Cookbook*. Packt Publishing Ltd, 2012.

[BGG14] Martin Brandtner, Emanuel Giger, and Harald Gall. Supporting continuous integration by mashing-up software quality information. In *IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE)*, pages 109–118, Antwerp, Belgium, FEB 2014. IEEE.

[Bir14] Jens Birchler. Sqa-timeline. Master's thesis, University of Zurich, 2014.

[BPSZ10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301–310. ACM, 2010.

[CD88] William S Cleveland and Susan J Devlin. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988.

[DAM08] Brian De Alwis and Gail C Murphy. Answering conceptual queries with ferret. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 21–30. IEEE, 2008.

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[FM10] Thomas Fritz and Gail C Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 175–184. ACM, 2010.

[GG11] Giacomo Ghezzi and Harald C Gall. Sofas: A lightweight architecture for software analysis as a service. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 93–102. IEEE, 2011.

[Ghe10] Giacomo Ghezzi. Sofas: software analysis services. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 381–384. ACM, 2010.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[KDV07] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.

[LM12] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[LZ05] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[MMW02] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Systems with Applications*, 23(4):405–413, 2002.

[RJ00] Linda Rising and Norman S Janoff. The scrum software development process for small teams. *IEEE software*, 17(4):26–32, 2000.

[SAG13] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, and Marco Aurelio Gerosa. Metricminer: Supporting researchers in mining software repositories. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 142–146. IEEE, 2013.

[Tro14] Silvan Troxler. Mininghub - a social mining and data sharing platform. Master's thesis, University of Zurich, 2014.

[VHDM07] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with semmlecode: an eclipse plugin for semantic code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 880–881. ACM, 2007.