

Research Project in Computer Science

Analyzing Dependencies between Software Architectural Degradation and Code Complexity Trends

Lukas Moritz Schulte, 17.01.2021

Department of Mathematics and Computer Science, Karlstad University, l.schulte@mailbox.org

Issues like software architecture degradation and technical debt manifest as source code of low quality. Many source code metrics exist to quantify the quality of code from different angles. Previous studies have shown that these metrics' ability to indicate architectural degradation is very limited. The goal of this research project is to shed some light on the question whether the trends of source code metrics over time can be better indicators. For this purpose, a tool for gathering data about metric trends from the version management system of software repositories has been created and evaluated. Initial answers to the question above are presented and discussed.

1 INTRODUCTION

The creation of a software system that fulfills commonly aspired quality attributes like flexible reusability, effective maintainability, and conceptual integrity, requires life-long attention and prominence to the system's architecture (Medvidovic and Taylor 2010). This naturally includes the documentation of the intended architecture of the system.

In this research project, conducted in the context of Karlstad Universities Research Project on Computer Science course for master's level students, the result of reflexion modelling, a specific form of such architecture documentation, will be used to validate a newly created tool for gathering data about metric trends from a version control system.

1.1 Motivation

Software systems that divert from their intended architecture, through shifted responsibilities among classes or unintended communication between components, suffer from architectural degradation. This is a common symptom of large-scale projects that have been under development for some time, without revising the architectural decisions made and without the investment of time into maintaining them (Murphy et al. 2001), leading to architectural erosion and drift (Perry and Wolf 1992).

To detect architectural degradation the codebases version of interest needs to be compared against up-to-date architectural documentation. Such a comparison can be achieved supported by static architectural-conformity-analysis techniques like reflexion modelling (Murphy et al. 2001; Passos et al. 2010). The creation of the reflexion model in a graphical form as well as the process of comparing it to the source code, can be performed with tools

like the eclipse¹ plugin JITTAC². It provides live feedback on how many dependencies exist between components and where they violate the model (Buckley et al. 2013). In an optimal scenario this is where refactoring can begin. Unfortunately, most projects that fit the description in the previous paragraph do not have up-to-date documentation that is detailed enough to be the basis for a reflexion model. Therefore, papers previously published have been investigating other ways to achieve the detection of architectural degradation.

One paper in particular has been the initial motivation for the execution of this research project: “Exploring the suitability of source code metrics for indicating architectural inconsistencies” (Lenhard et al. 2018), in which the authors attempt to identify the same cases of architectural degradation as a reflexion model, purely based on source code metrics. Findings in this area would bypass the need for complete documentation and allow developers to find starting points for refactoring tasks even in undocumented systems. The codebases investigated in the paper were distinct versions of the open-source projects Apache Lucene³ and Ant⁴ as well as JabRef⁵, which are all written in the Java⁶ programming language. These specific systems were chosen since they are sufficiently large and since they were under active development at the time. Although 49 metrics were calculated by seven different tools, the results of the study remain mostly inconclusive with more research to be done and more data sources to consider (Lenhard et al. 2018). In this research project that additional data source is going to be the codebase history.

1.2 Research question

As mentioned in the previous section, identifying architectural degradation based on source code metrics of one distinct version of the codebase does only deliver results that are mostly inconclusive. From the setup of this previous research the hypothesis can be made that more data points lead to improved accuracy in classification results. It will therefore be the goal of this research project to add metric data from the source codes history into a similar analysis.

(RQ1): *Can the sequence of source code metric changes be used to identify components that are likely to cause architectural inconsistencies?*

This way the development made in the codebase that led to the inconsistencies detectable by reflexion modelling may be the factor allowing for a more accurate detection of sources and targets of degradation.

For the work of this project to fit the margins set by the study regulations of this course, the research question was applied to one predefined system only. This system is the open-source bibliography management application JabRef, which was also used in the earlier paper (Lenhard et al. 2018) and therefore already has results of a valid reflexion model available. Its analysis will be described in the evaluation of chapter 4.

1.3 Contributions

Many commercial and free tools exist that enable developers and researchers to generate source code metrics of their codebases to measure code quality and other related factors. Some are integrated into IDEs (Microsoft 2020), others can be run standalone (SonarSource SA 2020b) or are accessible over a command-line interface (Aldanial 2020). They can also be distinguished into those tools that run on compiled source files and those that

¹ Homepage of the eclipse IDE: <https://www.eclipse.org/>

² Homepage of JITTAC: <https://arc.lero.ie/jittac/>

³ Project homepage of the Lucene search engine library: <https://lucene.apache.org/>

⁴ Project homepage of the Ant build system: <https://ant.apache.org/>

⁵ Project homepage of the JabRef reference manager: <https://www.jabref.org/>

⁶ https://www.java.com/en/download/help/whatis_java.html

analyze purely the plain code in a repository. Although when it comes to analyzing more than one version of the source code, few solutions exist that are also open for extension.

To investigate the research question asked in the previous chapter it was therefore deemed necessary to create a tool that (1) gathers the source code history data from a repositories version control system, (2) makes it accessible through open and common interfaces, and (3) is designed to interface with existing source code metric tools. The implementation of this tool was a major part of this research project and will therefore be the topic of chapter 3 where the focus lies on the architecture of the tool and its extensibility for further research.

The tool's core function is inspired by the works of the author Adam Tornhill and his books "Your Code as a Crime Scene" (Tornhill 2015) and "Software Design X-Rays" (Tornhill 2018). It relies on Git repositories as its primary source of historic data.

Beyond the editing history of the codebase read from the version control system Git, the data gathered by this version of the tool will include source code metrics provided by the static source code analyzer SonarQube. Those metrics will be calculated for every commit in the Git repository, providing this analysis with a timeline of change in code quality.

2 FOUNDATIONS

2.1 Codebase history data

The term codebase history data introduced in this research project describes data that is extracted from a version control system like Git. The history of a codebase contains information on who made changes, where and when changes were made, and what those changes mean for the quality of the source code found in the codebase. To have this data available in a way that machines and humans can read it alike, it needs to be extracted, aggregated, and enriched.

For extraction, the output of the Git repository can be read and interpreted via regular expression. During aggregation the data is filtered, with relevant information being stored in a database for later use. Finally, to enrich the data, third-party tools can be used to generate source code metrics for each of the versions registered in the database and the version control system.

2.2 Git

Git is a free and open-source distributed version control system. It is commonly used in software projects to organize the development of multiple programmers on the same codebase. For this it keeps track of all changes made in the codebase, allowing programmers to merge their changes with those made by others.

Git repositories are usually publicly or privately hosted through a provider for version control systems, like GitLab or GitHub, and locally operated through the Git CLI or an integration into a development environment.

To obtain a copy of the codebase a programmer can execute a *clone* command on the remote URL. After changes to the code were made, the *add* and *commit* commands add those changes locally, with the *push* command integrating the changes into the remote repository (given sufficient access right and no merge conflicts). Once this is done other programmers can obtain a copy of the newest version with the *pull* command, synchronizing the changes with their own version of the codebase.

In order to view the history of changes made to the codebase the *log* command can be executed, providing a chronological list detailing who made changes to which file, how many lines were added or removed, and whether there were changes to the files path. An extract of such a log is shown below.

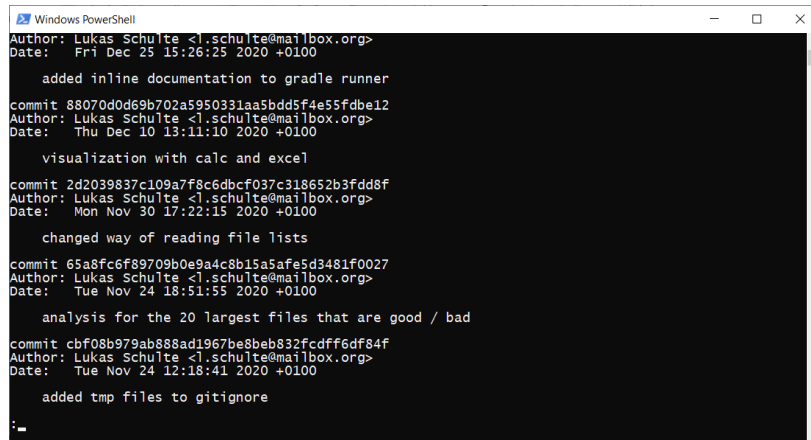
A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window displays the output of a Git log command. The output is a list of commits, each with a commit ID, author, date, and description. The commits are listed in reverse chronological order. The first commit is "added inline documentation to gradle runner" with commit ID 88070d0d69b702a5950331aa5bdd5f4e55fdbel12, authored by Lukas Schulte on Fri Dec 25 15:26:25 2020 +0100. The second commit is "visualization with calc and excel" with commit ID 2d2039837c109a7f8c6dbcf037c318652b3fdd8f, authored by Lukas Schulte on Thu Dec 10 13:11:10 2020 +0100. The third commit is "changed way of reading file lists" with commit ID 65a8fc6f89709b0e9a4c8b15a5afe5d3481f0027, authored by Lukas Schulte on Tue Nov 24 18:51:55 2020 +0100. The fourth commit is "analysis for the 20 largest files that are good / bad" with commit ID cbf08b979ab888ad1967be8beb832fcdff6df84f, authored by Lukas Schulte on Tue Nov 24 12:18:41 2020 +0100. The fifth commit is "added tmp files to gitignore" with commit ID -, authored by Lukas Schulte on Tue Nov 24 12:18:41 2020 +0100. The window has a standard Windows title bar with minimize, maximize, and close buttons.

Figure 1: Example for a Git log

Using the *checkout* command, it is thereafter possible to obtain an older version by its unique commit id. This can be used to inspect those previous versions, which will be of great importance for this research project.

2.3 Gradle

Gradle is one of many build automation-tools that supports multiple programming languages, amongst which are C++, JavaScript, and Java. The process of building with Gradle is based on tasks that can be combined to match the requirements of the build. This can include basic steps, like build or test, as well as plugins which can be integrated through the `build.gradle` file. Besides specifying available plugins, the `build.gradle` file, which is located at the root of the project, also defines further build configurations.

Gradle builds can be executed in two ways: Either through a local installation of Gradle or through a Gradle wrapper that uses the locally installed Java virtual machine. The file containing the Gradle wrapper is commonly called `gradlew` and accessible through a `gradlew.bat` file.

2.4 CLOC

CLOC is one of many tools that “counts blank lines, comment lines, and physical lines of source code in many programming languages” (AlDanial 2020). It is a simple source code metric generator that can be operated through a command line interface, returning its result in many formats, including JSON. It was used in the book “Your Code as a Crime Scene” (Tornhill 2015) and also integrated into the tool described in chapter 3 as an extension. Its basic commands include the measurement of single files and directories while supporting 270 languages / file types (AlDanial 2020).

2.5 SonarQube

SonarQube is a complete code quality measurement tool that allows its users to generate several metrics which go far beyond those found with CLOC. It gathers its metrics through an integration into common build systems for

27 programming languages like Java, C#, C and C++. This enables it to create metrics that count not only what is visible inside the plain code files, but also those that measure logical coupling between classes distributed over multiple files. Some metrics available when analyzing Java projects are listed below:

Metric	Description
Lines of code	Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.
Number of statements	Number of statements.
Number of functions	Number of functions. Depending on the language, a function is either a function or a method or a paragraph. In Java methods in anonymous classes are ignored.
Complexity	It is the Cyclomatic Complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1.
Violations	Total count of issues in all states.
Code smells	Total count of Code Smell issues.
Sqale index	Effort to fix all Code Smells. The measure is stored in minutes in the database. An 8-hour day is assumed when values are shown in days.
Bugs	Number of bug issues.
Duplicated lines	Number of duplicated blocks of line
Vulnerabilities	Number of vulnerability issues.

Table 1: Metrics of SonarQube (SonarSource S.A 2020a)

Analyses are performed within projects where each can, over time, contain metrics for multiple versions of the same codebase. They can be accessed either through the web interface or through a REST API.

2.6 Docker

Docker allows its user to virtualize operating systems and to deliver them packaged as so-called containers. With it, operating systems like Linux distributions can be started from a host system while also executing further commands that install software packages or run programs. A container is specified in a file with the extension `.dockerfile`.

Its functionality can be extended by docker-compose, which simplifies the handling of multiple containers that may be dependent on each other. A group of containers is managed in `docker-compose.yaml` files.

3 TOOL DESIGN

This chapter documents the development of the tool for gathering codebase history data which was implemented as a practical part of the research project. It is structured into sub-chapters with chapter 3.1 giving an introduction and describing the goals of the project, chapter 3.2 giving an overview over the systems components, chapter 3.3 describing the form and function of the implementation, and chapter 3.4 reasoning about the decisions that were made leading to this solution.

The implementation work of the tool is delimited from the scripts that were used to gather the results discussed in chapter 4. It describes the state of the tool including the optional integration of two extensions for gathering data from CLOC and SonarQube, which were both introduced in chapter 2.

3.1 Introduction and Goals

3.1.1 Requirements Overview

The driving force behind the development of the tool for gathering codebase history data is its prospected usefulness in this and further research projects. Its intended users, in the context of research, are expected to possess an advanced knowledge in computer science. Its purpose is to gather a software system's history from the system's codebase for a given timeframe. Since the research in which the tool may be helpful covers broad use cases, the focus of development lays not on providing a complete suite of functionalities, but rather on allowing for an easy extension on demand. Extensibility is for this reason one of the following three main requirements:

- (R1): The tool must gather source code history data from any git repository.
- (R2): The tool must store the gathered data persistent and accessible through third-party tools.
- (R3): The tool's data gathering and storing logic must be extendable, allowing the integration of other data providers.

Due to the nature of the environment in which the tool is intended to be used, and due to the type of projects it is supposed to be used on, the following delimiting requirements were formulated:

- (R4): Data visualization is not a responsibility of the tool itself.
- (R5): Starting the tool from the command line and changing start parameters in the source code is acceptable.

In the context of this research project further requirements need to be met. Those concern the additional data required to find answers to the research questions beyond the core functionality of the tool.

- (R6): Extensions required for the research questions must follow (R3).
- (R7): Extensions must cover sufficient number of common metrics used in earlier paper (Lenhard et al. 2018).
- (R8): One of the extensions must integrate a complete source code analysis tool like SourceMonitor⁷, CKJM⁸, or SonarQube.

3.1.2 Quality goals

The three main quality goals and corresponding scenarios have been determined in consideration of the tool's intended use. Hence, they all focus on allowing the application to be easily modified and used in varying environments. The goals are listed in Table 2 below.

Goal	Scenario
(G1) Extensibility through the integration of third-party data providers	Potential research questions that require codebase history data can be diverse, requiring at this point unpredictable metrics and metadata on the codebase's content. Easy extensibility of the tool and integration of other analysis tools is crucial for the usefulness of the project.

⁷ Homepage of SourceMonitor: <http://www.campwoodsw.com/sourcemonitor.html>

⁸ Homepage of CKJM: <https://www.spinellis.gr/sw/ckjm/>

(G2) Open access to the gathered data	The architecture of the application must provide an interface to all the gathered data for it to be used in other analysis scripts and applications.
(G3) Containerization	Installation of the tool should not depend on the operating system of the user. The only requirement should be an up-to-date installation of the containerization software Docker.

Table 2: Three main quality goals

3.2 Context and Scope

3.2.1 Business Context

The diagram below displays the business context of the tool as it is implemented in version 1.0.

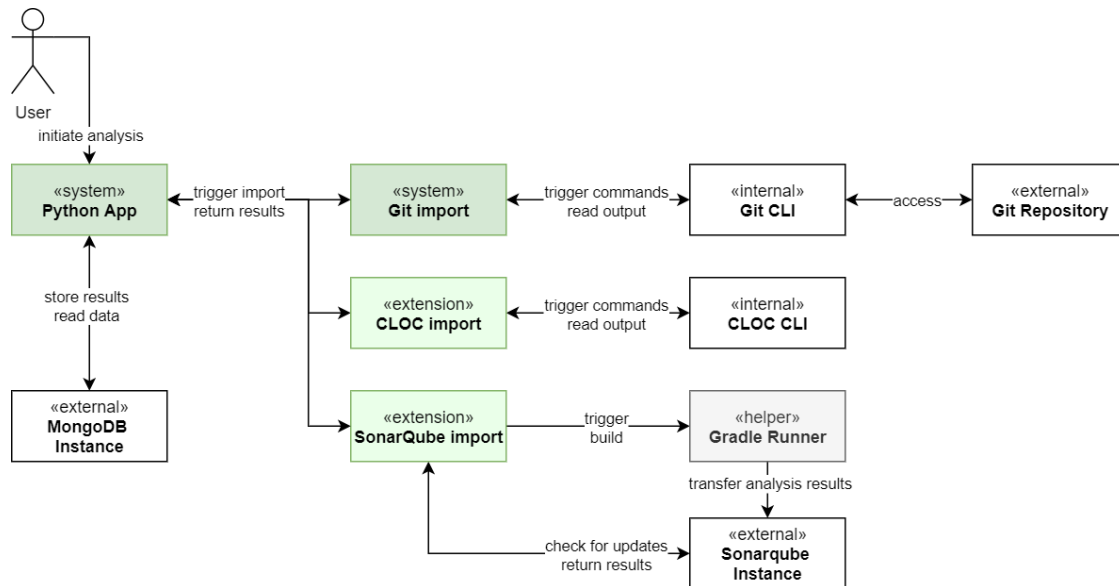


Figure 2: Business context diagram of the tool for gathering codebase history data

The core system consists of the three components *Python App*, *MongoDB*, and *Git import* (green), which provide functionality that is necessary to satisfy the requirements defined in chapter 3.1. The functionality is extended by *CLOC import* and *SonarQube import* (light green), which interface with additional data sources supporting the investigation of the previously formulated research questions. They are an example for the extensibility of the tool as demanded by the quality goal (G1). Further internal and external systems exist, which are either third-party tools hosted in- or outside the applications scope (white), or helper systems created to support the integration of such tools (gray). The latter is also maintained in the same repository as the core system but can be used independently. Since there is currently only one such helper system, the *Gradle runner*, and since it or a derivation of it may be of interest for the integration of further data sources, its business context is later documented in Figure 3.

It is noteworthy, that the complete tool described does not display any of the data itself. It merely stores it in the *MongoDB* where it later can be accessed by scripts written for data analysis. This conforms with the quality goal of open access (G2). Table 3 gives an overview over the intended function of each of the main systems components.

Component	Description
Python App	Starting point of the application. Also deals as communication hub for other components and extensions.
MongoDB	Instance of document-oriented database MongoDB.
Git import	System component responsible for communication with the Git CLI for the import of source code history data.
Git CLI	Instance of the Git CLI, installed on the host operating system.
Git Repository	Git-compatible repository publicly hosted on the internet or local network.
CLOC import	Extension component responsible for communicating with the CLOC CLI for calculating metrics from plaintext source code.
CLOC CLI	Instance of the CLOC CLI installed on the host operating system.
SonarQube import	Extension component responsible for communicating with the Gradle runner and the SonarQube instance for static source code analysis during the Gradle build process.
Gradle runner	A helper component implemented to spawn in multiple instances executing Gradle builds reporting metrics to SonarQube through integration of the sonarqube-gradle-plugin ⁹ .
SonarQube instance	Instance of the software analysis tool SonarQube.

Table 3: Description of components in the business context

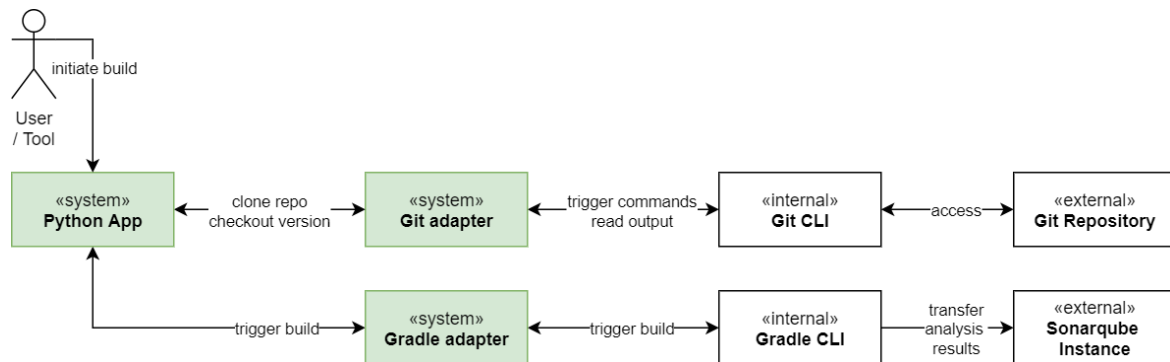


Figure 3: business context diagram of the Gradle runner

As stated before, the *Gradle runner* is its own, independent system, able to operate with a dedicated instance of a Git repository which it must analyze. It therefore has its own version of a *Git adapter*, similar to the *Git import* in the main system, also relies on a *Gradle CLI*, and can be operated by a user or any tool through REST-API calls.

⁹ Website of the SonarQube Gradle integration: <https://docs.sonarqube.org/latest/analysis/scan/sonarscanner-for-gradle/>

3.2.2 Technical Context

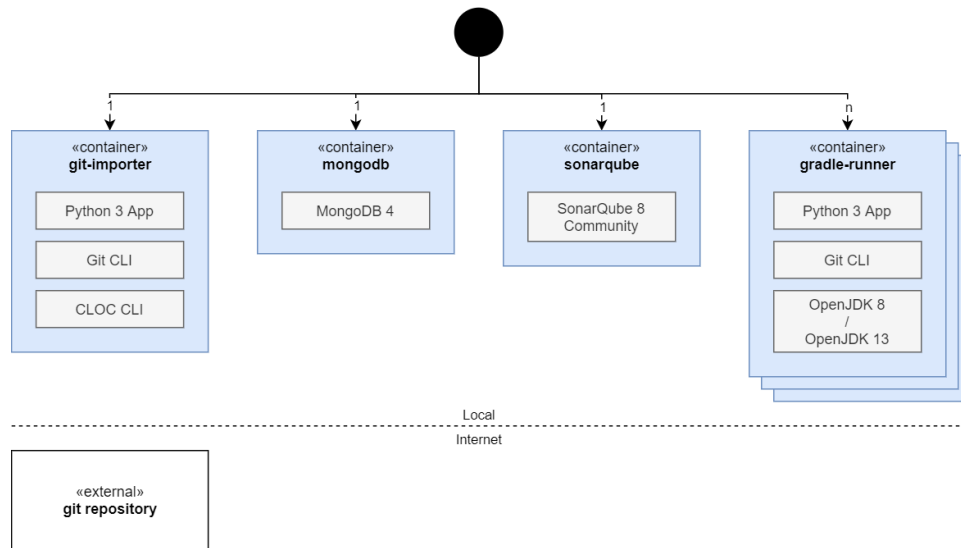


Figure 4: Technical context diagram of the tool for gathering codebase history data

The diagram in Figure 4 serves as the technical context of the application, highlighting the containerization with docker (blue), the high-level packaging of systems and subsystems inside each of the containers (gray) and listing the separately hosted systems components depend on (white). Startup of the application is handled by docker-compose, creating one instance each for the *git-importer*, *mongodb* and *sonarqube* containers, with the *gradle-runner* requiring manual scaling through input to the startup script as displayed below.

```

PS C:\VCS\cdbs> .\run.bat
Enter number of gradle runners to start: 3
docker-compose up --scale gradle-runner=3 --build -d
  
```

Scaling the *gradle-runner* container to more than one instance is required for analyzing projects with SonarQube that have more than a few commits and classes. If the hardware on which the execution is performed allows for multiple high-demand processes to run at once, parallelization can decrease the overall time required to gather the data. With this, the tool fulfills the quality goal of containerization (G3).

3.3 Building Blocks

The logical structure of the project was already documented in the previous chapters. In this chapter the specifics of the implementation will be discussed in building blocks which are displayed in Figure 5 below.

The first level only distinguishes between *apps* and *database*. On the second level two blocks in form of separate applications exist within *apps*: the *git-importer* and the *gradle-runner* apps. They form separate entities that can be executed independently, yet the *git-importer* relies on the *gradle-runner* for the execution of Gradle builds required by one of the extensions that generates metrics for SonarQube. Implementation of the applications was done using the Python programming language. Within the *database* block only one database by the building block name

mongodb exists. It is, as the name suggests, an instance of MongoDB which will also be described in the following sub-chapters.

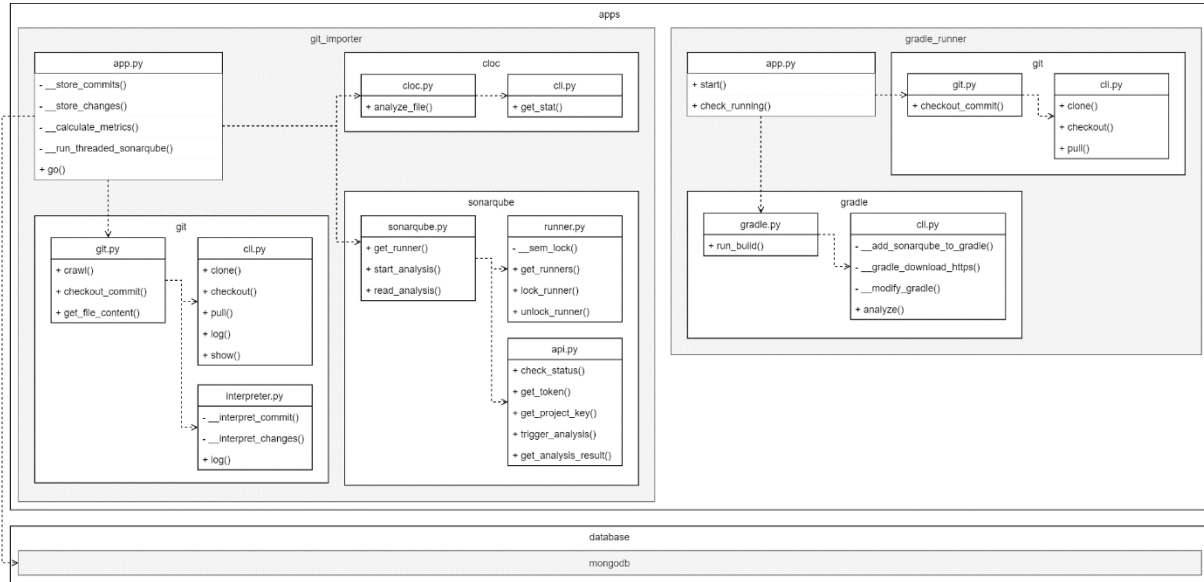


Figure 5: Building blocks

3.3.1 Git importer

The aforementioned main application - *git-importer* - itself is decomposed into its entry script called *app.py*, one integral component called *git* for handling the codebase as well as its history and the application's extension components. Those two are currently *cloc* and *sonarqube*, integrating the corresponding metric generators CLOC and SonarQube into the application.

To archive modularity of the solution, responsibilities between the components are clearly separated. The entry script *app.py* is the central hub for communication between *git* as well as current and future extensions. It is also the only component that is supposed to read and write in the database attached to the solution, allowing for extensions to be added and removed without interfering with the basic functionalities.

Within extensions and the *git* component, the composition is also strictly defined. A central script by the name of the component deals as an entry point and communication hub for specialized scripts, which for example access a CLI or use regular expressions to interpret input routed there from other components over the central script.

git consist of two such scripts, one interacting with the local Git CLI, providing functionality to clone, checkout and pull a repository, as well as reading the repository's history through the log output and returning the contents of a specific version of a file via the `git show` command. The command used for the log output is listed as an example to show what kind of data is processed gathering the codebase history data:

```
git log --numstat --no-merges --date=unix --after=YYYY-MM-DD.
```

cloc works in a similar fashion, although besides its *cli* component there is no need for an *interpreter* since the CLOC CLI offers JSON as a return format that can be parsed by Python itself. Its file-by-file analysis is executed through the command `cloc -json filename.ext`.

In contrast, the setup of *sonarqube* with its integration of the SonarQube API is more complex since it involves the *gradle-runner*. It consists of a *runner* class handling the allocation of runner instances and an *api* component. The latter is responsible for interacting with the REST interfaces of both SonarQube and the one from the variable number of *gradle-runner* instances. The implementation details of the features provided by the *gradle-runner* will be described in the following sub-chapter. For the functionality of the main application, it is only important to know that there can be multiple *gradle-runner* executing Gradle builds and that those are configured to include a *sonarqube*¹⁰ task which reports source code metrics to a given SonarQube instance. This process was already visualized in Figure 3 of chapter 3.2.1 and is again shown below as an extract of that figure.

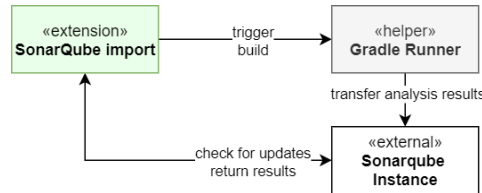


Figure 6: Business context extract

Once triggered via a HTTP call the build can take a few minutes. When it finishes successfully a “200 — OK” HTTP response is returned, together with the timestamp at which the build started. This is vital because the SonarQube instance may take another few minutes to process the metrics it received and with the timestamp the list of finished analysis can be checked to know if the results are ready to be read.

The process of a full import is then managed by the script found in *app.py*. It is visualized in the graph below.



Figure 7: Visualization of the import process

Within the step of *calculate metrics* previously stored commits and changes may be accessed again, depending on the requirements of the extension used for the calculation. To run analysis on specific versions of the code, those can be checked-out by corresponding functions of the *git* component. Specific versions of the contents of specific files can also be returned by the that component. It is intended that the integration of extensions is handled in this step. The other steps provide the codebase history and can be skipped once the anticipated timeframe has been stored in the database. For details concerning the data that is stored in the database see chapter 3.3.3.

3.3.2 Gradle runner

The decomposition of the *gradle-runner* is similar to the one found in the main application. The *gradle-runner* contains one entry script called *app.py* and two components called *git* and *gradle*. While *git* is a copy of the *git* component found in the main application that has been limited in functionality, the *gradle* component provides the functionality of triggering a time intensive Gradle build.

To run the Gradle build successfully, some requirements need to be fulfilled:

¹⁰ Documentation of SonarQube’s Gradle integration: <https://docs.sonarqube.org/latest/analysis/scan/sonarscanner-for-gradle/>

For once, the machine on which the *gradle-runner* is hosted needs to have a version of Java installed, that is compatible with the application that is to be build. When using the *gradle-runner* in the technical context of the *git-importer*, this can be archived by selecting the correct Dockerfile in the systems docker-compose config.

Then the application needs to be configured to use the Gradle build system for all the versions that are to be build. Specifically, all versions need to include a `gradlew` file in its root directory and a `build.gradle` file specifying the properties and plugins used for the build. Only the SonarQube Gradle-plugin does not need to be installed yet, since the *gradle-runner* can add it before execution. Here both ways of integrating the plugin, Gradle's DLS language as well as the legacy integration, are supported. It is also possible to remove broken dependencies from the build file within the *gradle-runner*, but those must be found manually. Broken dependencies may surface when attempting to build old versions of the codebase.

If the *gradle-runner* is used outside of the *git-importer*'s context, it may be necessary to define the URL under which the SonarQube instance is hosted by modifying the `__init__.py` script of the *gradle* component.

Running the *app.py* script starts the *gradle-runner*'s integrated webserver, through which it can be accessed by applications depending on it. This webservice has the following endpoints:

`http://cdbs_gradle-runner_1:5000/check`

Figure 8: Gradle runner check endpoint, GET

`http://cdbs_gradle-runner_1:5000/?commit=5f32&project_key=sq_key&api_key=5aa2`

Figure 9: Gradle runner root endpoint, POST

The *check* endpoint in Figure 8 was implemented so that other applications can check whether the webserver of the Gradle runner is up and running. Upon a HTTP Get request it returns a "200 – OK" HTTP response.

The *root* endpoint in Figure 9 can be used with a HTTP Post request, starting a Gradle build specified by the HTTP parameters and the JSON body of the request. Expected parameters and data schemas are defined below in Figure 10.

JSON body	Expects repository information as a JSON document. Required attributes are: title: String The title of the repository, must be equal to the folder name Git will clone the repository into. url: String The URL used to clone the Git repository.
commit	Expects a valid Git commit id of the repository
project_key	Expects a valid SonarQube project key.
api_key	Expects a valid API key for the SonarQube instance used.

Figure 10: Gradle runner root endpoint parameters and schemas

When accessing the *root* endpoint with the *git-importer*, its *sonarqube* extension will generate a valid project name and an API key automatically in coordination with the SonarQube instance. Otherwise, that information can be read manually from the SonarQube web interface.

3.3.3 MongoDB

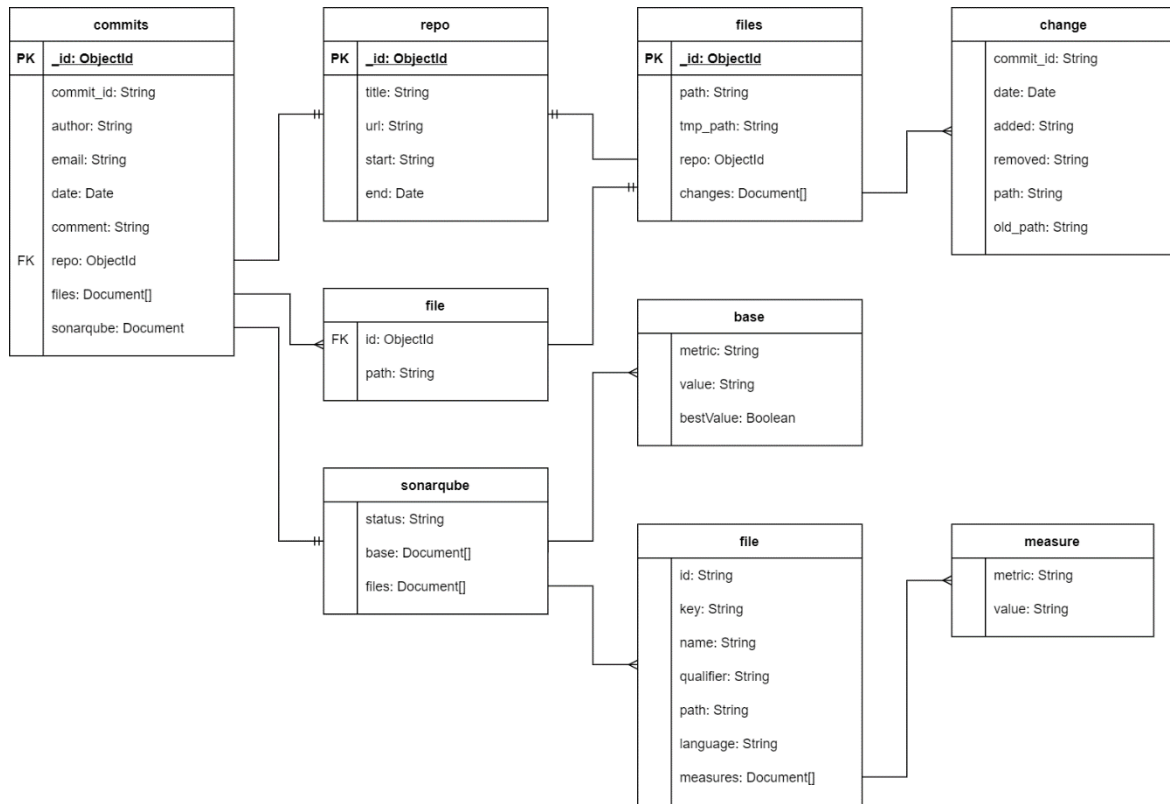


Figure 11: Database diagram

The data collected by the tool is stored in three MongoDB collections inside the *mongodb* component. Those collections listed below.

repos contains information on the repository that is to be analyzed. The collection needs to be created before starting the tool and filled with one entry conforming to the schema in Figure 11.

The collection *files* contains one entry per file that was found in the Git log interpreted by the *git-importer*. Tracing of renamed files is implemented and guarantees that one file does not have two entries in the database. Inside each file entry changes of the file itself are tracked by through the core functionality of the *git-importer*. That includes commit ids, old and new paths, as well as added and removed lines of code.

The largest collection, *commits*, is also created by the core functionality. It is later completed by the extensions of the tool providing more complex metrics calculated per commit. Currently only the SonarQube extension is activated and therefore only its data structures are visible in the schema of Figure 11.

3.4 Architectural decisions

3.4.1 Technology decisions

After defining the requirements of the tool, the first architectural decision made was the choice of programming language used. Whilst many programming languages are suitable for the task, Python was chosen since it meets

requirements set in chapter 6 and due to the authors personal preference for and experience with the language in the context of data gathering and analysis.

Since JSON is a common document type used in Python and by interfaces of various tools that might serve as extensions to this project in the future, MongoDB was chosen as a database of the tool. The MongoDB instance can be accessed from outside the application itself and started independently, therefore fulfilling the requirement for allowing access by third-party tools and custom analysis scripts for research.

3.4.2 Project compatibility

While the core functionality of the codebase data gathering tool is not language specific and works with all public Git repositories, the requirements set by the extensions necessary to answer the research question severely limit the universality of projects supported.

As described in previous sub-chapters, the integration of SonarQube relies on the Gradle build system executed by a Gradle wrapper file. If a project does not include such a file, and valid configurations to use it, its data cannot be imported by the tool. Depending on the SonarQube edition used, supported programming languages are also more or less limited. In the current version of the tool only Java projects have been tested.

3.4.3 Extensions

To answer the research question and satisfy the requirements (R6) to (R8) at least one complete source code analysis tool had to be integrated into the tool. Between SourceMonitor, CKJM and SonarQube, which were the three tools providing the highest number of metrics in the research paper (Lenhard et al. 2018) that served as a motivation for this projects research question, SonarQube was chosen since it has the highest number of metrics available and since its documentation was the most complete.

Before SonarQube was integrated though, an exemplary integration for CLOC was created to test the concept of expanding the tool with a simpler and less complex source code analysis tool. Since the metric calculated by CLOC, lines of code, is also available through SonarQube, the integration of CLOC is currently not activated and its data does therefore not appear in the database diagrams.

3.4.4 Helper tools

One decision visible in previous chapters was to design the *gradle-runner* as an external helper tool instead of as a component that is part of the main application *git-importer*. The main reason behind this choice is related to parallelization. During the testing phase of the SonarQube integration it was determined that running Gradle builds in sequence for all versions of a codebase was not feasible in terms of time required. Therefore, concepts for parallelization were explored, leading to the current solution which allows the user of the tool to run a variable number of *gradle-runner* instances in independent docker containers. In contrast to a previous idea which relied on the use of threading within Python, this approach proved to be less complex and easier to debug during development since the concern and logic of running Gradle builds was separated from the main application.

4 EVALUATION

To answer the research question of chapter 1.2, the data of a codebase's history is required. This chapter will on the one hand be used to evaluate the functionality provided by the tool described in the previous chapter as well as to analyze the data it gathered with the goal of answering the research question.

4.1 Object of the analysis

As stated in chapter 1.2, in the context of the research question, the project that will be the object of this analysis is the bibliography management application JabRef. It is one of the systems analyzed in the paper “Exploring the suitability of source code metrics for indicating architectural inconsistencies” (Lenhard et al. 2018). It therefore already has a validated reflexion model available, which is crucial for further analysis of the research question. Among the other projects analyzed in the aforementioned paper, it was chosen because it is a more traditional desktop application that, on a high level, separates into GUI, model and logic. The other projects, Apache Ant and Lucerne, are a CLI tool and a library.

4.1.1 JabRef

The open-source project JabRef exists since 2003 and provides a standalone interface for editing BibTeX files as well as integrations for multiple text editors like Word and LibreOffice. With those, the management of references in documents like research papers can be organized and partly automated.

In order to collaborate on the development of JabRef the developers use a Git version control system hosted on GitHub. In January 2021¹¹ it has 15.793 commits from a total of 339 contributors, contains a total of 1901 Java source files and 150308 lines of Java code.

Yet, since a valid reflexion model is required for the analysis of this chapter, an older version was chosen instead. This version is the same used in the previous research paper (Lenhard et al. 2018) and was released on the 13 July 2016 (v3.5). For this version the researchers created and validated a reflexion model that reflected the intended architecture. This was achieved consulting the development team of that time, and using a tool called JITTAC that functions as a plugin into the integrated development environment eclipse (Lenhard et al. 2018).

4.1.2 Data pool

The JabRef project in version 3.5 conforms with the requirements set by the tool for gathering codebase history data and its extensions. It is implemented in Java, uses the Gradle build system through a Gradle wrapper, and has its own Gradle specification file to integrate plugins. Its reliance on multiple Java versions can be addressed by executing multiple rounds of analyzing the code with SonarQube and changing the version the *gradle-runner* is installing. Still one factor limits the amount of historic data that can be imported into the data pool: JabRef versions older than July 2014 use the Ant build system instead of Gradle. Since the *gradle-runner* in its current form cannot adapt to this, the data pool is limited to the timeframe between the 13th of July 2014 and the same day in 2016, covering two years of data in total. It was determined that this timeframe must be sufficient to perform the analysis.

In that timeframe 3723 commits were created, with a total of 27022 files changed. Together with the SonarQube integration spread over eight *gradle-runner* instances, gathering that data took the tool approximately 5 days.

As it is common while developing software, not all commits are finished versions of the application. Therefore, not all build processes completed successfully, which led to missing SonarQube metrics for 1055 commits. The distribution of successful and unsuccessful builds over time is displayed below.

¹¹ Version 5.2, analyzed with CLOC

4.2 Visual analysis

The first step that was taken to determine whether the data gathered can be used to identify architectural inconsistencies was to visualize it, with the goal of identifying visual characteristics in the graphs. Therefore, two types of graphs were created for a select amount of data: graphs comparing “good” files with “bad” ones and graphs comparing files that are *targets* of architectural degradation with those that are the *sources* of degradation.

In the context of this analysis, files were labeled “good” if they are neither *source* nor *target* of architectural degradation, and they were labeled “bad” if they are. This labeling is based on the data found through the reflexion model of the paper (Lenhard et al. 2018).

4.2.1 Comparing good and bad files

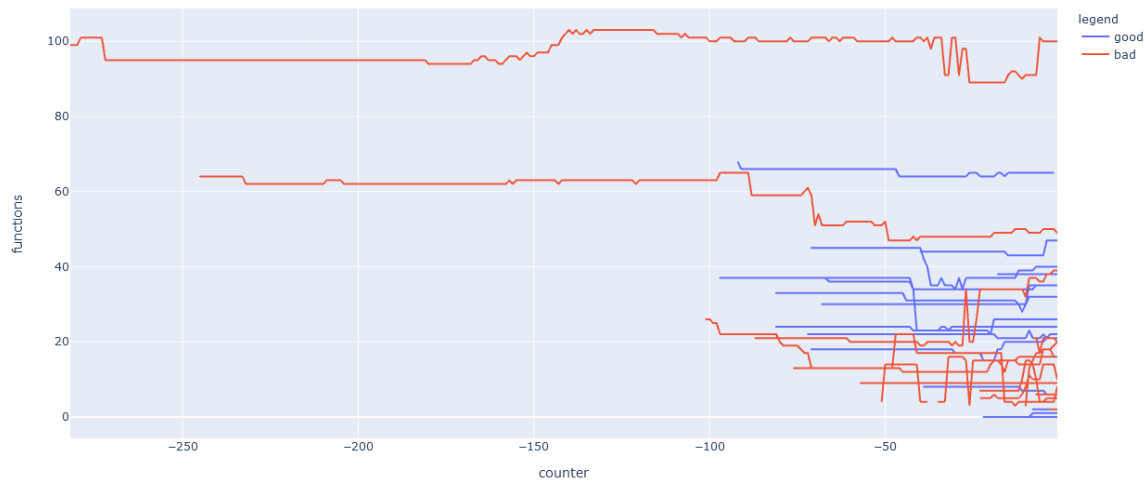


Figure 14: Number of functions between good and bad files

While the y-axis of the following graphs shows a file’s metric value, the x-axis acts as a counter for the number of changes that a file is behind on change zero, with change zero being the last change of the file before release v3.5 on the 13th of July 2016.

A first look on the graphs which show the evolution per metric of the 15 *bad* files with the most architectural degradation and the 15 largest *good* files suggests, that on the one hand stability is a feature of *good* files, while a less stable metric value is one of *bad* files. This applies specifically to the ones showing the *number of functions* in Figure 14 and the ones showing the *complexity* in appendix A 4. Still, these are no distinct features that would allow a solid classification since there are also *good* files that are unstable.

What stands out is that files with many commits and high metric values seem to be *bad* ones, yet the general result of this visual analysis remains inconclusive.

4.2.2 Comparing sources and targets of architectural degradation

For this comparison files that are *sources* of architectural degradation were classified as those with five or less incoming and more than five outgoing violations to the intended architecture as defined in the reflexion model of (Lenhard et al. 2018). Files with five or less outgoing and more than five incoming violations were in return classified as *targets* of degradation. For the following graphs, the lesser number of files in those two categories (29)

was set as a maximum number of lines displayed in the following figures to prevent an imbalance. Although colors may be used more than once each line represents a different file.



Figure 15: Metric functions for targets (top and left) and sources (right) of architectural degradation

Again, the features of the graphs seem similar with no visual distinction between targets and sources, yet sources of architectural degradation seem to have a slightly lower number of commits.

The visual analysis of the data does not appear to lead to a method of solving the research question.

4.3 Timeseries Classification

Since the visual analysis of the metric history remained inconclusive, the idea of training a neural network to classify *good* and *bad* files was briefly explored. For this a timeseries classification model was created using the open-source library Keras. It was then trained with the codebase history data and a truth extracted from the results of the reflexion model.

While it was an interesting experiment, the resulting classifier could not differentiate between the classes, as visualized in the confusion matrix of Figure 16.

This does not say though, that neural networks are not suitable for the task. With more time to adjust the parameters of the network as well as more data, for example from other projects analyzed by the codebase data gathering tool, it may be possible to archive better results.

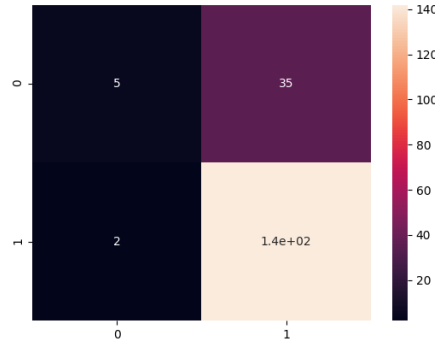


Figure 16: Keras confusion matrix (0: bad file, 1: good file)

4.4 Clustering

Since the visual and the automated classification of classes based on characteristics of the graph's course have so far been inconclusive, clustering based on metrics cumulated in the dimension of time was explored. If it proves to be efficient, the clustering configuration could be used to automate classification as it was the idea behind the training of a neural network in the previous chapter. Even if a binary classification into *good* and *bad* files or *sources* and *targets* is not possible, the clustering results can still be analyzed to spot characteristics for sub-groups.

4.4.1 Methods

To display the changes in metric values per file, two formulas have been applied on the source code history data: mean square deviation and mean sum of change.

$$MSD = \frac{1}{N} \sum_{i=1}^N |\bar{x} - x_i|^2, \text{ with } \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

Figure 17: Mean square deviation

$$MSC = \frac{1}{N} \sum_{i=1}^{N-1} |x_i - x_{i+1}|$$

Figure 18: Mean sum of change

With those cumulated values multiple configurations for the clustering functions of the Python library SciPy¹² were used to group similar files. Configurations were executed via the code in

```
def get_cluster(data, threshold, criterion='distance'):
```

```

    Z = ward(data)
    res = fcluster(Z, threshold, criterion)
    data.insert(0, 'cluster', res)

    return data
```

Figure 19 and defined through its input parameters. For the input parameter *data* varying combinations of cumulated metric values were chosen to create broad but concise datasets. They were combined with the available

¹² Homepage of the Python library SciPy: <https://www.scipy.org/>

clustering *criteria* “inconsistent”, “distance”, “maxclust”, “monocrit” and “maxclust_monocrit”. The clustering threshold was afterwards manually varied to limit the number of clusters generated to around 10 for the results to be comprehensible. All results were then exported to CSV files and inspected through a spreadsheet application.

```
def get_cluster(data, threshold, criterion='distance'):

    Z = ward(data)
    res = fcluster(Z, threshold, criterion)
    data.insert(0, 'cluster', res)

    return data
```

Figure 19: Python code for creating clusters with SciPy

During the visual inspection of links between the cumulated metrics and the number of inconsistencies, the percentiles in which those numbers are located were calculated to support the observations. Calculation of the percentiles was performed by the default functionality of the spreadsheet application¹³.

4.4.2 Results

The automatic clustering produced mostly inconclusive results¹⁴ which did not lead to rules for classification. Still the best results where, on a small scale, clusters of three or four files when applying “distance” criterion to data containing only one metric and the number of commits per file.

cluster	function	count	class	source	target	total
5	0,22	245	JabRefPreferences	161	452	613
5	0,44	282	BasePanel	0	95	95
5	0,42	333	JabRefFrame	0	30	30
4	0,29	144	BibtexParser	3	0	3
4	0,28	151	ImportInspectionDialog	0	2	2
4	0,18	196	EntryEditor	0	2	2
4	1,68	166	WarnAssignmentSideEffects	0	0	0

Figure 20: Example for small clusters found with the MSC of the number of functions, the commit count, the distance criterion, and a threshold of 100 (Schulte 2021).

Most other clusters contained both *good* and *bad* files and overall examples like the ones in Figure 20 were rare.

A manual identification of clusters on the other hand resulted in some findings useful for classification on a small scale. The manual clustering was conducted in a spreadsheet application by calculating the metrics percentiles and grouping them in multiple configurations. Yet here the limited number of files available results in small clusters which may not be representable for other projects. An overview of the data is available at (Schulte 2021).

¹³ LibreOffice Calc documentation for PERCENTILE:

https://help.libreoffice.org/Calc/Statistical_Functions_Part_Four#PERCENTILE

¹⁴ Results can be reproduced through the *data-dashboard* scripts that are available on the *analysis* branch of

<https://git.cse.kau.se/lukaschu100/cdbs/-/tree/analysis/> / https://github.com/l-schulte/cdbs_hist_kau/tree/analysis

When looking at the number of commits on a file and the medium square distance between the values of the function count metric, the following observations were made:

If the function MSD is in the 98th percentile, 10 out of 15 files are bad. And if the commit count is in the 98th percentile, 12 out of 16 files are bad.

Although this first observation is not very precise in spotting files that contain architectural inconsistencies, it may provide a quick overview over files to start with when investigating a codebase.

If the function MSD is in the 98th percentile and the commit count outside the 90th percentile, 4 out of 5 files are bad.

The files *GroupTreeNodeViewModel*, *CustomImportList*, *GroupTreeNode*, and *CleanupPreset* all contain architectural inconsistencies with no separation between *sources* and *targets* while matching the observation above. The exception in this is *CustomImporter*. It does not contain any inconsistencies according to the reflexion model.

If the function MSD and the commit count are in the 98th percentile they are either themselves in the 98th percentile of classes in terms of the number of inconsistencies or have no inconsistencies at all.

This observation is relevant despite its low hit rate of only 4 out of 7 because with it both files with the highest number of inconsistencies in the codebase can be classified. Those files, *JabRefPreferences* and *Globals*, have 613 and 244 inconsistencies, respectively. The other two hits, *MetaData* and *JabRefFrame*, have 38 and 30 inconsistencies and thereby still classify as being in the 98th percentile of files.

If the commit count is in the 98th percentile and the function MSD outside the 95th percentile, 8 out of 9 files are bad.

Curiously, delimitations for bad files can also be made by selecting only those outside of the high metric percentiles. For this observation out of nine files only *BibDatabase* is a false positive.

As long as the medium square distance was used as an accumulator, similar metrics gave mostly comparable results to those described above, yet the connections between files were the clearest when looking at the MSD of the number of functions.

In contrast to the mean square displacement (MSD) the mean sum of change (MSC) shows no visible correlations with the file's architectural inconsistencies. Therefore, no observations could be made with it. The data is available at (Schulte 2021).

The results of this analysis, in the form of observations, require validation. Such could be archived through analysis of further projects, yet it is not part of this paper. Those further analysis would require reflexion models of those projects and time for the codebase data gathering tool to run its import processes.

5 CONCLUSION

In this paper two major topics were discussed: The implementation of a tool for gathering metric trends from a version control system, and the analysis of whether this data can be used as indicators for architectural degradation in the codebase.

The tool design followed the requirements R1 to R8 set in chapter 3.1. It therefor allows the user to gather codebase history data from any public Git repository via the command line (R1, R4, R5), storing the data in an instance of MongoDB (R2) while being designed to support extensions (R3) to enrich the data for broader usefulness. It also includes one such extension (R6), an integration of the complete code quality measurement tool SonarQube (R7, R8), which provides data vital to the investigation of the research question.

The architectural design also defines the quality goals G1, G2, and G3, which were all met. The extensibility of the solution was documented and proven through the integration of SonarQube (G1), the open access to the gathered data was made possible using the common non-relational database MongoDB (G2) and the containerization for easy installation was documented and realized with Docker (G3).

The functional operability of the tool was then tested by executing it on the open-source project JabRef and its Git version control system, gathering enriched codebase history data of a two-year timeframe. While minor adjustments had to be made to circumvent problems that were mostly project specific, the overall process was successful. At this point the major obstacle of gathering data from a large codebase is time. The Gradle builds required by SonarQube are time-consuming and can extend the total running time of the tool up to several days. Compatibility issues which may occur due to changing dependencies during the project's development history can therefor lead to unnoticed failures and leave timeframes without proper metrics. At this point further improvements may be required.

Finally, the data pool created by the tool for gathering codebase history data was deemed sufficient to start the process of finding an answer to the research question.

Out of three attempts to classify files into the categories of those that are free of architectural inconsistencies ("good" files) and those that contain inconsistencies ("bad" files), as well as sub-categories of *targets* and *sources* of inconsistencies, no clear pattern was established.

The visual analysis of graphs displaying the change of metric values per file did not show consistent characteristics for either category. The only observation possible was that files with many commits seem to be likely *bad* and a *target* of inconsistencies.

The timeseries classification based on a neural network trained by the codebase history data did not result in a classification mechanism either. It is at this point noteworthy though, that more data and more time to optimize the networks parameters might lead to better results.

Lastly, the application of (multi-) dimensional clustering on cumulated metric values, generated for example with the mean square deviation, led to a few partly correct classifications. Still, the results were far away from the goal of creating a classification strategy. Yet, a manual analysis of the cumulated values and their connection to each other resulted in some observations that might be useful as clues towards which files are likely to contain architectural inconsistencies. Since the observations are based on the percentiles in which the cumulated metrics lie, they may be transferable to other projects. As such further analyses are required to validate them.

In regard to answering the research question, "*Can the sequence of source code metric changes be used to identify components that are likely to cause architectural inconsistencies?*", the answer supported by this research project cannot be final. The data analyzed supports assumptions towards the existence of some rules after which *good* and *bad* files can be classified. Still a validation of those assumptions through further research is required.

The tool for gathering codebase history data has the potential to allow those analyses. Projects that are similar to JabRef can be analyzed and used to gather a larger data pool, on which clustering or machine learning may produce more conclusive results.

A. APPENDIX

A 1 References

- AlDanial (2020) cloc: Count Lines of Code. <https://github.com/AlDanial/cloc>. Accessed 20 December 2020
- Buckley J, Mooney S, Rosik J, Ali N (2013) JITTAC: a just-in-time tool for architectural consistency. ICSE '13: Proceedings of the 2013 International Conference on Software Engineering:1291–1294
- Lenhard J (2018) lenhard/arch-metrics-replication. <https://github.com/lenhard/arch-metrics-replication>. Accessed 8 January 2021
- Lenhard J, Blom M, Herold S (2018) Exploring the suitability of source code metrics for indicating architectural inconsistencies. *Software quality journal*
- Medvidovic N, Taylor RN (2010) Software architecture. In: Kramer J, Bishop J, Devanbu P, Uchitel S (eds) ACM/IEEE 32nd International Conference on Software Engineering, 2010: ICSE '10 ; 2 - 8 May 2010, Cape Town, South Africa. IEEE, Piscataway, NJ, p 471
- Microsoft (2020) Code analyzers - Visual Studio. <https://docs.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2019>. Accessed 20 December 2020
- Murphy GC, Notkin D, Sullivan KJ (2001) Software reflexion models: bridging the gap between design and implementation. *IEEE Trans. Software Eng.* 27:364–380. <https://doi.org/10.1109/32.917525>
- Passos L, Terra R, Valente MT, Diniz R, das Chagas Mendonca N (2010) Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Softw.* 27:82–89. <https://doi.org/10.1109/MS.2009.117>
- Perry DE, Wolf AL (1992) Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes* 17:40–52. <https://doi.org/10.1145/141874.141884>
- Schulte L (2021) Tables. https://git.cse.kau.se/lukaschu100/cdb/-/tree/paper_14-01-21/apps/data-dashboard/tables / https://github.com/l-schulte/cdb_hist_kau/tree/analysis. Accessed 14 January 2021
- SonarSource S.A (2020a) Metric Definitions: SonarQube Docs. <https://docs.sonarqube.org/latest/user-guide/metric-definitions/>. Accessed 6 January 2021
- SonarSource SA (2020b) SonarQube: Code Quality and Security. <https://www.sonarqube.org/>. Accessed 20 December 2020
- Tornhill A (2015) Your code as a crime scene: Use forensic techniques to arrest defects, bottlenecks, and bad design in your programs / Adam Tornhill. The Pragmatic Bookshelf, Dallas
- Tornhill A (2018) Software design x-rays: Fix technical debt with behavioral code analysis / Adam Tornhill. The Pragmatic Bookshelf, Raleigh

A 2 Tool setup guidelines

What follows is a short setup description for the codebase data gathering tool. This is necessary since some preparation steps are currently not automated.

Installation

Due to containerization, the only external software that needs to be installed is Docker. Docker can be downloaded online (<https://www.docker.com/products/docker-desktop>).

The tool itself can be downloaded from the master branch of its Git repository. It is located on the GitLab instance of Karlstad University (<https://git.cse.kau.se/lukaschu100/cdbs>), with a public copy available on GitHub (https://github.com/l-schulte/cdbs_hist_kau).

Database setup

The setup of the database requires the creation of a “repos” collection, containing at least the information of one repository.

Start the MongoDB instance from the command line of the tool’s root directory:

```
docker-compose start mongodb
```

Then connect to the MongoDB shell of the container:

```
docker exec -it mongodb mongo --username "root" --password "localdontuseglobal"
```

And insert the repository information following the schema below:

```
use cdbs_db
db.repos.insert(
  {
    title: "jabref",
    url: "https://github.com/JabRef/jabref.git",
    start: "fd695d24782266018ee90407cf1210959c238bb2",
    end: "2014-07-13"
  }
)
```

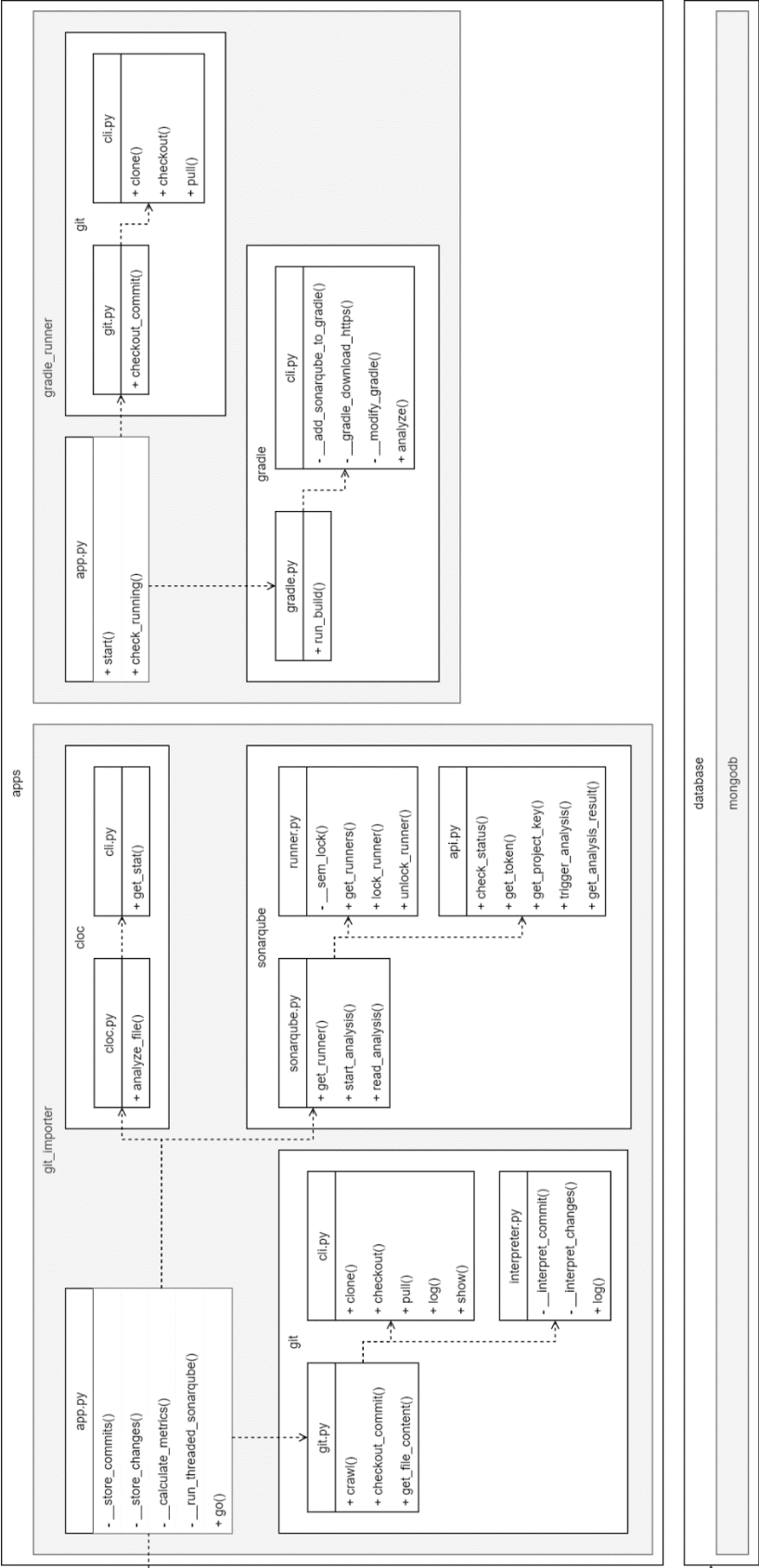
Afterwards the setup is complete, and the tool can be started. Note that the *start* point of the import is defined by the commit of the newest version that is to be imported and that the *end* of the analysis is defined by a date that must lie further in the past than the *start*.

Tool execution

When the setup is completed, the tool can be executed through the file “run.bat” in the projects root folder. It will ask for the number of gradle-runners to start and then execute all necessary components also starting the gathering process.

```
.\run.bat
```


A 3 Building blocks (full size)



A 4 Graphs comparing *good* and *bad* files

