Developing a git Repository Mining Tool for Change Coupling Analysis

Bachelor's Thesis

Karim Issa

March 26, 2021

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Priv.-Doz. Dr. Henning Schnoor
Dr.-Ing. Reiner Jung

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 26. März 2021

Abstract

This thesis aims to explore the possibilities of analyzing the content of a software projects version control system to gain insights about dependencies, module encapsulation and other data implied by the changes made to a software system over a period of time. In fact the presented analysis methods operate on a file system level and therefore work with any programming language, framework and any other project specific feature. The main work of this thesis in the development of a ready-to-use, highly customizable, self-contained analysis tool written in python. It discusses the problems that occurred during the development and the algorithmic solutions to these problems. This thesis also explains the code structure, software usage instructions and other information needed for further development. The tool utilizes the git version control system for data collections and delivers the results of the analysis in the form of a rendered, node based graph, visualizing the coupling strength between different files. These results are also exported to standardized text file formats and can be used for further computations.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Introduction to Coupling Analysis . . . 2
1.2.1 Change Coupling Metric . . . 3
1.3 Goals . . . 5
1.4 Undirected Raw Counting . . . 6
1.5 Directed Raw Counting . . . 7

2 Tool Development **9**
2.1 Requirements . . . 9
2.2 External Libraries . . . 9
2.2.1 PyDriller . . . 9
2.2.2 click . . . 10
2.3 Project Structure . . . 12
2.3.1 Python Project . . . 13
2.3.2 Rendering Module . . . 13
2.3.3 Analysis Module . . . 13
2.4 Modularity & Extendability . . . 14
2.4.1 Initialization . . . 14
2.4.2 Preprocessing . . . 15
2.4.3 Coupling Algorithms . . . 18
2.4.4 Input/Output Formatter . . . 20
2.5 Rendering Module . . . 20
2.6 Interesting Algorithms . . . 21
2.6.1 ArtifactStore . . . 21
2.6.2 Preprocessing Algorithms . . . 24
2.7 Performance . . . 30

3 Documentation **33**
3.1 Usage Explanation . . . 33
3.1.1 Quick Start . . . 33
3.1.2 Advanced Usage . . . 33
3.2 File Format Specification . . . 38

4 Examples **41**
4.1 Kieker . . . 41
4.2 ExplorViz . . . 45
4.3 Titan . . . 48
4.4 Spring Framework . . . 51

5 Conclusion **53**

6 Future Work **55**

Bibliography **57**

Chapter 1

Introduction

1.1 Motivation

In the ever more diversifying landscape of software engineering, developers get confronted with a restlessly growing number of code frameworks, programming languages and end-user platforms. With this increasing complexity, it gets harder to keep an overview of a software projects inner workings, code dependencies and structural details. When such a project evolves over time, these missing insights naturally lead to the degradation of a projects code quality and structural consistency, therefore further lowering the maintainability of the software project. This can also lead to worse code performance, negatively affecting the user.

To counteract these problems the technological progression has also lead to more capable software development tools, so developers can keep up with the increased software complexity. One of the most important tools of modern software development being the git version control system. It supports developers in organizing their code, automating workflows and collaboration with other developers. But the git version control system can also be utilized on a completely different level, in the standoff against the aforementioned problems: It contains valuable metadata. This metadata consists of valuable contextual information [Ball et al. 1997], with its primary purpose being the representation of the version history itself, consisting of commits, branches, tags oder releases. That contextual information can also be interpreted from a different perspective: It shows how the different project files changed over time and most importantly, how these file changes are related to each other.

This thesis aims to extract and analyze the git metadata from this file-based point of view. The analysis takes the entire version history into account, by connecting the information contained in each of the commits. This creates a file based data model that is different from the branch-commit-tree model, usually used to represent the git repository information. The main goal is for developers to learn about new properties of their software projects, by looking at their code from this new perspective. Ultimately, this newly gained knowledge can help developers learn about and approch structural problems of their code base. And as this analysis technique is based on the version control system itself, it is completely independent from any programming language and framework used in the software project and thus gives results for any possible combination of technologies used, which nicely defuses the preliminarily presented problems.

1.2 Introduction to Coupling Analysis

To begin with, this chapter will provide some background information about the broader research field of coupling metrics. The analysis technique dealt with in this theses is called change coupling analysis and is one of many strategies of implementing such a coupling metric.

Stevens et al. [1974] define coupling in the context of software development as a *connection* between a systems *modules*. The important piece of data are these connections, which can be measured by doing a coupling analysis, using different kind of coupling metrics. Some metrics also provide a way to measure the *strength* of such a connection, for a more detailed view. With the help of this analyzed information, one can make assertions and predictions about the *quality* of the analyzed piece of software. As the types of these modules and intermodular connections differ between different metrics and analysis strategies, so do the kind of predictions, that result from the analysis. This difference is demonstrated on the following two examples of coupling metrics in the context of object oriented programming: The static coupling metric and the dynamic coupling metric. Additionally to these two examples the aforementioned change coupling metric will be discussed in more detail.

Static Coupling Metric

The static coupling metric contemplates the source code of a project. Classes, enums and other high level code structures are possible types for the system modules of the metric. Possible measures of detecting a connection between two modules **A** and **B** include: A method call from **A** to **B**, member variables from **A** of type **B** or an exception of type **B** thrown by **A** [Schnoor and Hasselbring 2020].

One way to use this information is to create a dependency map of the modules, to verify packet encapsulation, to review class hierarchies and to spot unwanted architectural characteristics like low cohesion.

Dynamic Coupling Metric

On the one hand static coupling analysis is relatively straight forward to pull of, as only the static code needs to be analyzed. On the other hand, static coupling fails to account for some properties of object oriented programming languages like polymorphism and dynamic binding, while facing the problem of filtering out theoretically unreachable lines of code [Arisholm et al. 2004; Schnoor and Hasselbring 2018].

Dynamic coupling analysis promises to solve these kind of problems, by looking at the call structure between modules at runtime. In the specific context of Java, the looked at modules are now manifested as chunks of allocated heap memory and the corresponding intermodular connections are, along other possible examples, analogous to the contents of the stack memory. So looking into a running application provides results, that represent the applied dependency tree, where all dynamic aspects, that where not inferable from the static code, are resolved. Yet this technique requires more sophisticated analysis tools like ExplorViz [Hasselbring et al. 2020] or Kieker [Hasselbring and van Hoorn 2020], to actually bind to a running application, rather than only looking at text files. The output information of the analysis allows for the same kind of predictions as possible with the static analysis method, while providing a different point of view on the looked at software.

1.2.1 Change Coupling Metric

These very different approaches of bringing about a coupling analysis have a very limiting problem in common. Both techniques need the be specifically tailored to fulfill a single programming language's or system architecture's requirements. In particular the semantic and syntax is different between most programming languages, so the parser of the static analysis can not be utilized for an arbitrary language. Along the same lines the resulting binary program, bytecode or transpiled piece of code vastly differs between the used technologies, which makes them incompatible for the use with a dynamic analysis tool, that needs to hook into running applications or may even needs to be compiled into the looked at software. Contrary to the these two examples, the change coupling metric contemplates a completely different level of a given software project: The contents of the version control system. Viewing a software project from this angle has the benefit of being *almost* independent from any kind of technology used in the project for the following reasons: Any major version control systems works on any platform and with any filetype. And as the contents of the files are not authoritative for this metric, this analysis stays on a file system level, which is a universally compatible abstraction layer on all modern version control and operating systems. The only project inherited factors in this regard are

- That differences in the folder and file structure between the used technologies is suspected to lead to slight differences in quality of the data generated during analysis.
- Also the granularity of file has to be moderate, meaning that projects where developers tend to put many lines of code into few, large files or very few lines of code into many, small files tend to generate worse analysis results.
- This analysis technique can only deliver relevant results for projects with a regular use of the version control system, so enough data is collected over the course of development [Robbes et al. 2008; Robbes 2008].
- Bad *commit practices* of the participating developers can have a potentially huge negative impact on the meaningfulness of the output analysis data.

Especially this last factor is hard to pin down, but plays a very important role in the analysis quality and will be further clarified in later chapters.

As a side note, although the change coupling metric can be defined for a variety of different version control systems, this thesis only focuses on the git[1] version control system, but most statements hold for any other commit based version control system. As part of future work, this tool can also easily expanded to support other version control systems. Further information needed for continued development is provided in Section 2.4.

When analyzing the data of a git repository, the main data records available for analysis are the commits of the version control system. The git history timeline also contains some other data types like tags and releases, but they can only really be used to structure the timeline and also the stages of development itself, as these datasets are just timestamps with an attached label. Yet they do not provide any information about the files and file relations, that the coupling metric is interest in. Commits on the other hand indeed contain all the needed metadata, in particular:

- The **author** that issued the commit.
- The **branch** that the commit is contained in.
- The **date** and **time** the commit was issued.
- The **hash** of the commit, used as an unique identifier.
- The **commit message** that is attached to the commit.
- Most importantly, a list of **modifications** of all the files that changed since the last commit. A git file modification can be of one of the following three kinds:
  1) **Addition**: A new file was added to the repository and will be tracked by the version control system beginning from this commit.
  2) **Modification**: An already tracked file has its content altered and/or was moved to another path inside the repository.
  3) **Deletion**: A tracked file was deleted from the repository and will no longer be tracked. Either by only removing it from the version control system or completely deleting the file from the hard drive.

In fact a commit holds even more records than listed above, among others a list of parent commits or the committer, who can be a different person than the author. But these and other more sophisticated features of the git version control system are left out at this point for the sake of simplicity, as they are not relevant for the contents of this thesis.

From this point on the term *artifact* will be used in addition to the term *file*. As this metric tracks the complete evolution process of a project, an artifact describes the entity of an ever changing file, from the point in time it is added to the version controls system,

[1]Official Git Website: `https://git-scm.com`

up util it is no longer under version control, possibly deleted. The terminology of a file, describes the state of an artifact at a specific commit or other point in time. For example, the addition of a new file creates a new artifact that will exist in the version history from now on. Analogous, a git commit that contains a file modification of an already tracked file, represents a change in the state of the belonging artifact. The artifact will exist in its new state for all future commits and points in time up until further modifications are mode to it.

With this background information in mind, the change coupling metric is defined over the artifacts of a repository's history as the system modules and the intermodular connections are present between artifacts, that somewhere in the version history co-change. Specific, in detail definitions are provided in Section 1.4 and Section 1.5. Further definitions can be found in the paper of Oliva and Gerosa [2015]. Co-change means that those artifacts where changed in the same commit or in two, some kind related commits. A coupling strength can additionally be defined as some kind of numerical value that is dependent on the amount, that two artifacts co-changed, which can be observed in the definition in Section 1.5.

In which cases a co-change or coupling is detected and the formula for calculating its strength, are defined by the specific analysis tool or method, that implements this metric.

1.3 Goals

The main goal of this thesis is to develop such a change coupling analysis tool. Many fundamental algorithmic ideas of this project are based on the work of Oliva and Gerosa [2015]. But contrary to the code snippets provided by that paper, this thesis aims to deliver a ready-to-use, all-in-one solution, doing all the analysis work.

This tools tries to deliver an easy, yet highly customizable usage. Achieving this requires the interface between the analysis tool and the user to be very feature rich, while still keeping most of these settings optional. These optional settings can be utilized by advanced users to optimize the analysis results for specific projects and are fully documented in Section 3.1, while still ensuring that the program can be run by anyone from scratch.

But as this exploratory thesis does not contain any studies about user interaction with the tool, it is expected, that not all users needs are satisfied. So, additionally to a user friendly experience in mind, the code of the tool is built on the premise of high modularity and therefore expandability, so future work on the project can easily add missing features. The tools code architecture and the expansion possibilities are discussed in the chapters Project Structure (Section 2.3) and Modularity & Extendability (Section 2.4).

Though analysis results are be exported in the form of a file, additionally this tool will also contain a small graph visualizer, so that no third party software is needed, to have a quick look at the results. Both the analyzer and the renderer will be contained in a single CLI application, which may also expect any additional parameter settings. This all-in-one package can therefore also be easily utilized in automated processes like statistical evaluations, an example of this can be seen in the Section 4.3.

The analysis part of the tool will assess the the information contained in the provided git repository using the two following algorithms. One of which may be chosen by the user. Also new algorithms can easily be added by a developer, if needed, as part of furture work.

1.4 Undirected Raw Counting

This algorithm represents one of the simplest ways so calculate an artifact coupling and most other algorithms are based on the same foundations as this algorithm. It is therefore perfectly suited to elucidate the general concept of change coupling.

	Artifact A	Artifact B	Artifact C
Artifact A	10	1	4
Artifact B	1	7	5
Artifact C	4	5	11

Figure 1.1. Undirected raw counting example data, randomly generated.

In the above Figure 1.1 artifacts **A**, **B** and **C** are tracked in the version history of a sample project. The values of the main diagonal of the table (values with red circles) represent the number of times, this artifact has changed over the course of development. This demonstrates that artifact **A** changed **11** times, artifact **B** changed **7** times and artifact **C** changed **10** times in total. The other values of the table show how often two artifacts co-changed, which means how often they changed in the same commit. Therefore the two sections of above and below the diagonal are symmetric in respect to their value and the top section can be ignored. For example, artifacts **A** and **B** changed together one time (blue circle) and artifacts **B** and **C** co-changed 5 times (green circle).

For this simple algorithm, these values already represent the final coupling results, as no further computations are done to improve the informative value of the acquired data. This results in coupling connection strengths between two artifacts **X** and **Y** being just the value inside the table cell in the column of artifact **X** and the row of artifact **Y** respectively. This simple method can be used to easily reveal some kind of dependency clusters or globally used objects, that have coupling connections to a huge variety of other artifacts.

But it fails to convey information about how strong each artifact actually depends on the other one. This means the data will not show if artifact **A** is more dependant on artifact **B**, than **B** is on **A**, or vice versa.

1.5 Directed Raw Counting

The directed raw counting algorithm aims to solve this problem, by building upon the information gained by executing the undirected raw counting algorithm and adding another computing layer on top of it. The same example dataset for artifacts **A**, **B** and **C** is used as for the undirected case. The difference to the undirected case is, that between two artifacts there are now up to two connections that also imply a direction by pointing from one artifact to the other and vice versa. The strength of the directed connection is the ratio of the number of times both artifacts co-changed with one another and the total number of changes of the artifact the connection originates from. This can be seen in the following Figure 1.2:

	Artifact A	Artifact B	Artifact C
Artifact A	10	1	4
Artifact B	1	5/7 ≈ **0,71** 7	5
Artifact C	4	5	5/11 ≈ **0,45** 11

Figure 1.2. Directed raw counting example data, randomly generated.

In the context of the example data the directed connection from artifact **B** to **C** (blue box) is therefore 5 (the number of co-changes of **B** and **C**) divided by 7 (the total number of times **B** changed), which is roughly equal to $5/7 \approx$ **0,71**. Accordingly the directed connection from artifact **C** to **B** (green box) is calculated in the same way resulting in a directed coupling strength of about $5/11 \approx$ **0,45**. From this data one can now deduce, that artifact **B** is likely to be more dependent on artifact **C**, than **C** is on **B**. The word *dependant* at this point just conveys the vague idea, that the code written in artifact **B** is in some form related to the code in artifact **C**. For example **B** could contain a unit test of the class coded in artifact **C**.

After having calculated all the connection strengths, there is still the task of identifying all the relevant couplings and filtering them out. Which couplings are relevant depends on the use case. It could be strongest coupling or couplings belonging to a certain set of artifacts. Methods for doing this filtering and other intermediate steps required for a complete analysis, are presented in the appropriate sections of the following Chapter 2.

Chapter 2

Tool Development

This main chapter will cover the complete course of the analysis tool's development. It will be especially helpful for developers aiming to continue development of the tool, as all kind of technical details and used frameworks are discussed. The open source project can be found on GitHub[1].

2.1 Requirements

The entire project, consisting of an analysis and a graph rendering module, are written in python. For all included frameworks to work, at least python version **3.6** is required. The python version used for testing and development was **3.9.1**. An up to date *requirements.txt* file can be found at the root of the project, which is intended to be used by pip[2]. The actual installation steps are described in detail in Section 3.1 and also inside the projects *README.md* file.

Also in order for the git-python interface to work, the standard git CLI itself needs to be installed on the system.

Apart from git, python and the pip modules, which should already be preinstalled on most modern operating systems, the project has no further requirements in order to fulfill an easy-to-use premise.

2.2 External Libraries

Over the course of development the project has experienced a growing number of external python libraries as its dependencies. The major two of these libraries will be presented here, as they form the backbone of the entire project and made it possible for the project to have a low code footprint, while delivering a fair amount of features.

2.2.1 PyDriller

One of the core mechanics needed for a git repository analysis is to actually gather the raw version control system data that is hidden inside database like structures on the filesystem

[1]Analysis tool: `https://github.com/e5kimol4ser/repository-mining`

[2]pip project: `https://pypi.org/project/pip/`

and transform them into a useful data structure by using a git interface. *PyDriller*[3] is an academic open source project providing an easy, performant interface for mining git repositories [Spadini et al. 2018].

In fact it is so easy to use, that the entire git integration of the project can be broken down to two lines of code: Line **3** and **4** of the following code snippet.

```
1 from pydriller import RepositoryMining
2
3 repository = RepositoryMining("https://github.com/example/repository")
4 for commit in repository.traverse_commits():
5   # do stuff
6   pass
```

In line 3 an *RepositoryMining* object is initialized. The passed string argument needs to be a remote url to a publicly available git repository or a relative or absolute path to a local directory, containing a git repository. In this example a remote url was supplied, which means PyDriller will first clone the repository internally and delete this temporary copy after the analysis. The *RepositoryMining* constructor also accepts many more arguments, for example to filter for certain branches or authors. At line 4 this *RepositoryMining* object is now used to retrieve all the commits of that repository in the form of a python *generator*, which can be thought of as a lazily allocated list. This list is then being iterated to compute the analysis for all the commits, each containing at least the needed metadata mentioned in Section 1.2.1. A full documentation of these data types can be accessed at the libraries documentation website[4].

2.2.2 click

To ease the process of developing a full fletched CLI application, with many additional arguments and options, this work has been outsourced to a python library called *click*[5]. Click leverages python annotations, so that developers can simply decorate their functions to add options and arguments to them and to also expose these functions to the CLI application. This is intuitively explained with the help of the following code snippet, which is inspired by one of *click*'s documentation examples:

[3]PyDriller on GitHub: `https://github.com/ishepard/pydriller`

[4]PyDriller Documentation: `https://pydriller.readthedocs.io`

[5]click Website: `https://palletsprojects.com/p/click/`

```
1 import click
2
3 @click.command()
4 @click.argument('name')
5 @click.option('-c', '--count', default=1, help='number of greetings')
6 def hello(name, count):
7   for x in range(count):
8     click.echo(f"Hello {name}!")
```

After importing click in line 1, a new function that will be exposed to the command line is added with the *command* annotation in line 3. This command will inherit its name from the name of the function, in this case *hello*. Then the command is enhanced by an argument with the label *name* in line 4. An argument is a positional parameter, that must be supplied by default, when the corresponding command in called on the command line and is always of type string (*str* in python). Being a positional parameter means that multiple arguments are added to a command. They will be parsed from the command line input in the order of the annotations. Contrary to arguments, options are non positional parameters with more feature and also with a different syntax, when typed into the command line by the user. In line 5 of the code snippet, such an option is added to the *hello* command with the name *count* and a default value of **1**. This default value tells the click library one the one hand, that this parameter is optional, as it will have the value **1** when not supplied. On the other hand the datatype of the option is an Integer (*int* in python) and click will check every value supplied for this option for being a valid integer. Also a shorthand name is supplied, which is optional, to shorten command line invocations. All options and arguments defined by annotations will then be supplied as function parameters to the *hello* function in line 6. This function will print out a greeting message, containing the supplied *name* argument, *count* times.

In general the complete command line input required to call the above snippet looks like this:

```
$ python3 <file_name> hello <name> <--count=X|-c=X>
 |---1---|-----2-----|--3--|---4--|--------5-------|
```

Each part of the command line call is explained in detail below:

1: Running the program with the python interpreter. Version 3 is required for this project.

2: This should be the name of the python file containing the code. For example *greeting* for a file named *greeting.py*

3: This is the name of the command that it called. In this example *hello*.

4: Here the user can type the contents of the required *name* argument. Multiple required arguments would be written one after another and parsed with respect to their order.

5: Optionally the *count* option can be supplied, either with the double dash full name syntax or single dash short name syntax. The *X* variable in the snippet above needs to be replaced with a valid integer. If a command has multiple options, they can be supplied in an arbitrary order, as options are non positional arguments.

Example calls and outputs can look like this:

```
$ python3 greeting hello Alice --count 3
Hello Alice!
Hello Alice!
Hello ALice!

$ python3 greeting hello Bob
Hello Bob!

$ python3 greeting hello -c 2
Usage: . hello [OPTIONS] NAME
Try '. hello --help' for help.

Error: Missing argument 'NAME'.
```

As seen in the last call of the program, where the required *name* argument is not supplied, click also adds an interactive help context, that adjusts accordingly to the missing bits of informations. When defining the option and argument annotations, a help message explaining the parameter can be supplied, which will be shown to the user in this help message.

Further customization options and utilities can the found in the click library documentation[6].

2.3 Project Structure

This chapter will quickly explain the general filesystem structure of the python project as well as other important files of the entire project. A detailed look into the system and code architecture of the tool is given in Section 2.4.

There are no relevant files in the main directory of this project, besides the usual *README.md* file, that quickly explains the usage of the tool for external users, and the already mentioned *requirements.txt* used by pip.

The project repository contains a *cache* folder. Apart from a small test repository, this directory is not tracked by git. It is meant be used as a common place, where repositories, that need to be analyzed, either during development or production use, can be cloned into. This can save time compared to always having to pull a remote repository for each analysis

[6]click Documentation: `https://click.palletsprojects.com`

run and keeps project related files in one place to reduce unnecessary fille cluttering on the host. The testing repository, that is already contained in the *cache* directory, covers most common git operations and can be used as a good starting point for debugging.

2.3.1 Python Project

The actual python project of the tool is located inside the *mining* directory. It contains the *__main__.py* python file that should be called by the python interpreter to launch the analysis tool. The special file name is automatically recognized by python and shortens the command line call to launch the tool. This can be observed in the actual usage examples in Section 3.1.

Alongside the main file, there are three folders. The *analysis* folder contains the analyzing part of the tools, whereas the *rendering* folder contains a small graph rendering module for result visualization. Inside the *util* folder lie several small utility functions, classes or modules, that are shared by the analysis and rendering modules.

Going deeper into the file structure, from now on every folder contains a *__init__.py* file, which needs to be present for the python interpreter to pickup the contained python modules. These files can also contain export definitions, that serve as default exports of this directory, to flatten and better organize the dependency structure.

2.3.2 Rendering Module

As the rendering module is only an additional, non fundamental part of the complete tool, it does not have an as sophisticated featureset as the analysis module and therefore does not need an overly complicated architecture. The four additional files contained in the *rendering* module folder are *graph_metadata.py*, *import_graph.py*, *process_graph.py* and *render_graph.py*. Each representing different steps in the graph data processing and rendering pipeline. Its inner workings are discussed in detail in Section 2.5.

2.3.3 Analysis Module

The analysis module on the other hand designed from the bottom up to be modular and therefore easily extendable, so that the tool can be continued to be developed as part of future work. Architectural details of this approach are discussed in this chapter. The implementation part and information needed for further development are provided in Section 2.4.

The analysis module's structure is divided into different stages. The code for these stages is located inside the *stages* folder inside the *analysis* directory. Similarly to the four steps of the rendering module pipeline, the analysis stages are orchestrated from within the *__init__.py* file. The four stages and their corresponding directories are, in processing order:

1. First of all, in the *initialization* stage, all the data gets extracted from the git repository and all data structures needed for the upcoming analysis are instantiated.
2. After that, the raw data undergoes the *preprocessing* stage. Here different kinds of algorithms are scanning the commits for certain characteristics and try to mutate or filter out commits in an effort to improve the final analysis results.
3. This preprocessed data is than analysed in the *evaluation* stage. It will identify the co-changes of the different artifacts and store this information in an appropriate internal data format.
4. Finally, the *export* stage will transform this internal data format into the specified output file format and write it to the file.

All these stages are far more complex than the rendering pipeline steps and each stage contains multiple sub-stages. The sub-stages are executed by the *__init__.py* file, found in the corresponding folder of each stage. The sub-stage design differs from stage to stage and will be explained in the following Section 2.4.

2.4 Modularity & Extendability

The different software design concepts of the stages and sub-stages are clarified in this chapter. They are the foundation of the modularity and extendability of this analysis tool.

2.4.1 Initialization

The *initialization* stage is structured procedurally, meaning it contains just some functions, which get called sequentially. The three consecutive sub-stages are:

1. The first step is to initialize the PyDriller *RepositoryMining* instance inside *repository.py*. Here many user options regarding the selection of specific commits get passed.
2. The resulting repository then gets passed to the function inside *commits.py*, where all the commits will actually be instantiated and transformed into custom data structures to save hardware resources on large projects. More on that performance related matter is discussed in detail in Section 2.7.
3. Finally an *ArtifactStore* instance is created inside *artifacts.py*. The purpose of the *ArtifactStore* is to keep track of the artifact instances that persist between commits and where defined in Section 1.2.1. A complete description of its technical functionality is given in Section 2.6.1.

Generally speaking, it just transforms the compressed data contained in the file structure of a git repository into uncompressed data structures in memory for the tool to have easy and quick access to this data. Because of this, no further level of abstraction is needed here, because the functionality of this stage is likely not something that will undergo huge changes in future development, as it is neither dependant on specific project requirements, nor flexible in terms of user configurability.

2.4.2 Preprocessing

Contrary to the first *initialization* stage, the *preprocessing* stage has a highly modular structure and the amount of sub-stages that are executed, depends on the users configuration. Therefore an abstraction mechanism for uniformly handling the sub-stages is provided by the *SubStage* class inside the *preprocessing* folder. Each sub-stage has to fulfill the job of applying certain modifications to the raw list of commits, that is generated in the first stage, to improve the quality of the the actual analysis. This is most probably done by either:

1. **Filtering** out a commit, hence ignoring it in the analysis.

2. **Splitting** up the modifications list of a commit. This should be done, when some of the file modifications contained in a single commit, are not considered to be dependent on one another and therefore need to be de-coupled.

3. Or **combining** the modifications of multiple commits, to create new couplings between previously independent file changes.

Examples of all three methods are already included in the project, as four sub-stages are provides by this thesis out-of-the-box. The order of execution of the three methods listed above can also have an impact on performance and analysis quality. Doing the filtering of unwanted commits first, will save computational effort later on. And the order in which the splitting and combination of commits are executed, possibly mixed, should be respected in the sub-stage implementation accordingly. So for example a combining sub-stage could revert the efforts of a previous splitting stage, by re-combining previously split up commits. In general, any sub-stage should not brake the constraints any previous stage wanted to enforce, but this is currently accompanied with much implementation effort and is a possible subject to be improved as part of future work.

It is important to note, that the list of commits itself is not altered, but rather the data inside each commit should be modified. To support the three preprocessing methods describes above, special data structures for commits and file modifications were developed. Details about these data structures and the way commit splitting and combination is implemented can be found at Section 2.6.2.

To implement a new sub-stage, a new *SubStage* object hast to be instantiated and added to *preprocessing/__init__.py*, where the *SubStage*'s only method *process* is called, that iterates over all commits and possible changes them. The *SubStage* constructor expects four arguments:

- **context_sensitive**: A boolean, which indicates that this sub-stage is context sensitive in the terms of needing the complete commit list, even when analyzing just a single commit. This will be especially the case for *combination* algorithms, as one commit is always looked at in the context of other commits that share common properties and therefore need to be coupled together.

- **should_process**: A function that, given the command line input parameters from the user, has to return a boolean, that indicates, if that sub-stage should actually be executed.

- **process_commit**: A function which will enforce the algorithms constrains on a commit, by altering the commits metadata as documented in Section 2.6.2. It is always given the users command line input parameters, the current commit to be analyzed and, if the sub-stage is context sensitive, also the complete list of commits.

- **desc**: A string that is shown to the user alongside a progress bar, which indicates the preprocessing progress of that stage.

Calling the appropriate methods and showing the progress bar is handled uniformly by the *SubStage*.

The following subsections describe the four preprocessing methods and algorithms (in order of actual execution), that are developed as a part of this thesis. Examples on how their usage affects the actual analysis data can be found at Chapter 4.

Filtering out Large Commits

Large commits are identified as those commits, that singlehandedly contain at least a certain number of modified files. This threshold can be configured by the user and has a default value of 50, which is just an arbitrarily chosen number, which is meant to be a starting point, as it needs to be adjusted on a per project basis [Zimmermann and Weißgerber 2004].

Ultimately, when a user chooses to activate this option, all large commits will be ignored for the analysis. The rationale behind this being, that those large commits are regarded as unwanted *noise* in the data set, that needs to be filtered out [Zimmermann and Weißgerber 2004]. This noise originates from certain commit practices, that have negative side effects on the version history. In this case large commits may indicate migrations from another version control system, any other kind of project re-import or huge structural changes in the code. All those cases do not reflect the semantical connection between certain artifacts, that this analysis seeks, but rather contain unrelated co-changes.

This sub-stage can be found in the *filer_large.py* file. It will not be performend, when the third sub-stage, that more gracefully handles large commits, is also executed. Further information about this can be found below in Section 2.4.2, but this is an example of how the constraints of multiple different sub-stages can overlap and need to be handled individually.

Filtering out Merge Commits

The reasoning behind filtering out merge commits is the same as for filtering out large commits. Zimmermann and Weißgerber [2004] also regard them as unwanted noise, as they contain all changes made in all commits, that the merging branch adds to the branch it merges in to, combined. But as debugging during tool development showed, the PyDriller merge commits will always contain an empty modifications list. This is also confirmed by looking into the sourcecode of the library. But as the implementation of this sub-stage was already done at this point, the source code of this substage is still contained in the project, because extending the tool to support more version control systems, as part of future work, may reveal the usefulness of this step. The code for this filtering stage can be found in *filer_merge.py*.

Splitting Large Commits

As mentioned before, this thesis proposes a new, more elegant handling of commits exceeding the large commit threshold. This algorithm will look at the complete file path of each modification and tries to split the modifications based on their distances in the file system tree. This will result in the modifications belonging to similar directories to be still coupled and modifications with very different file paths to be de-coupled. Implementation details are given in Section 2.6.2. The reasoning behind this being, that a large commit that indicates an architectural change in the software system, can still hold valuable information about the new code structure. And trying to keep couplings that lie in the same directory will reflect the new architecture in the coupling data. Additionally in most computer languages or frameworks, files belonging to the same module or sharing a code-dependency are often packed together in the same folder or are at least not far-off in terms of distance in the file system tree.

In the worst case for this algorithm, too many modifications could still occur in the exact same directory, and can therefore not be split further. If this chunk of modification still exceeds the large-commit-threshold and the sub-stage for filtering out large commits would originally have been executed, these modifications will be ignored. This reflects the aforementioned behavior of the first preprocessing sub-stage, so that its constraint, of not coupling a chuck of modifications that is larger than the threshold, is preserved. This algorithm can be found in the *split_large.py* file.

Combining Consecutive Commits

This last preprocessing sub-stage is also a newly proposed algorithm to counteract bad commit practices. But contrary to handling too large commits as described above, this method tries is intended handle very small commits. As some developers might prefer to commit their code very often in small chunks due to personal preference or workflows, a change coupling algorithm would not be able to extract useful information from these incomplete commits. The whole picture of a newly developed feature or patch is split up into small fragments and thus all these small, related commits need to be combined to be properly coupled together. When the user decides to use this preprocessing method, a time intervall needs to be specified. Then this sub-stage will combine all commits, that are committed in the same specified time intervall by the same committer/developer and that are consecutive commits, meaning they need to be directly subsequent in the version history. Further implementation details and the mechanics used to handle multiple commits as one chunk of coupled data, without breaking the commit structure, are described in Section 2.6.2. The implementation is done in the *combine_consecutive.py* file.

2.4.3 Coupling Algorithms

The entire *evaluation* stage consists of two steps:

1. Choose the algorithm the user has selected or throw an error if the user input is invalid.
2. Execute the algorithm and return the results to the stage orchestrator.

The sub-stages of the evaluation stage are represented as subclasses of the abstract class *CouplingAlgorithm*. Only one of the different algorithms, that is available for selection, will actually be executed. Expanding the tools functionality in this stage is done by implementing and offering more algorithms for the user to choose from.

As an abstract class can not be instantiated directly, due to non implemented methods, subclasses need to be created, which implement all missing features. This ensures an uniform interface of any algorithm, which makes all coupling algorithm implementations interchangeable and therefore easy to handle. A new *CouplingAlgorithm* implementation, added by a developer, just needs to be registered in the *evaluation/__init__.py* file and is ready-to-use, which can be seen below, in a code snipped of that file:

```
algorithms = {
  'urc': UndirectedRawCountingAlgorithm,
  'drc': DirectedRawCountingAlgorithm
}
```

This dictionary holds all algorithm implementations selectable by the user. To register a new implementation of the coupling algorithm, a subclass of *CouplingAlgorithm* needs to be added to this dictionary under a free to choose key. Instantiation and execution of the algorithm will be handled automatically, due to the aforementioned interchangeability. The key of the dictionary will be be value of the *–algorithm* option, that the user needs to supply when calling the tool from command line. During the course of this thesis two coupling algorithms are already implemented, the *UndirectedRawCountingAlgorithm* and the *DirectedRawCountingAlgorithm*. The theoretical background of these algorithms are covered in Section 1.4 and Section 1.5 respectively. The concrete implementations are both contained in the *RawCountingAlgorithms.py* file, as they share a large portion of their code.

For example, to call the undirected raw counting algorithm, the user needs to specify the option *–algorithm urc*. *urc* being the key of the corresponding algorithm of the dictionary (line 2 of the above snippet).

A subclass of *CouplingAlgorithm* has to implement the following two methods:

```
1 @abstractmethod
2 def process_commit(self, cli_args, commit, commits, artifacts_store, graph_data):
3   pass
4
5 @abstractmethod
6 def construct_graph(self, cli_args, graph_data):
7   pass
```

The *process_commit* method (line 2) is called for every commit that needs to be analyzed. This commit is passed as the *commit* argument. Additionally some specific algorithms may need to consider other commits for context sensitive analysis methods, as other commits may be related to the commit currently looked at. For this case, the *commits* argument always contains the complete list of commits that will be analyzed. The *artifacts_store* argument should be used to identify the unique artifacts across different commits, the technical functionality behind this is provided in Section 2.6.1. The task to be solved by this method is to fill the supplied *graph_data* dictionary, which persists between each call of the method, with the results of the coupling analysis of each analyzed commit. The structure of the *graph_data* dictionary is not standardized and can be utilized as the developer pleases, as its only use is to get passed to the second method.

This second *construct_graph* method serves the purpose to translate the proprietary *graph_data* argument to a standardized output format, which will be output as a file and contains all the data needed to render the node-edge based graph. The output dictionary of this method should have entries with the keys *nodes* and *edges*, both containing a list of the following data types as their value:

1. A **node** (item of the *nodes* list) represents an artifact, the module of a coupling analysis metric, and is itself a dictionary with the following entries:
   - An **id**, which is a string and has to be unique amongst all other nodes.
   - A **label**, which is meant to be rendered onto a visual representation of the graph, so the user can identify each node with this human readable string.
   - And finally a **value** holding a numerical data type. This value is also presented to the user and represents the total number of changes this artifact undergoes during the analyzed timeline, but the actual semantics of this value can be adjusted to specific algorithms.
2. An **edge** (item of the *edges* list) represents an intermodular connection between two artifacts. It also is a dictionary and has the following entries:

- An unique string **id**.
- A boolean **directed**, that indicates a directed edge.
- **start** and **end** id strings to identify start and end nodes this edge connects. Both should be equal to an *id* of a *node* contained in the same *graph_data*.
- A **weight** that is meant to represent the coupling strength between two artifacts, for example how many times two artifacts co-change. But the final interpretation of this value is flexible and can once again be adjusted to satisfy a special algorithm's demands.

2.4.4 Input/Output Formatter

This chapter covers the abstraction mechanics behind the file *export* stage of the analyzer and also the file import steps of the graph renderer. The concept is the same as in the previous chapter about the algorithm stage. As this abstraction mechanism is shared by multiple parts of the tool, it is located inside the *util/formatters* directory. An abstract *Formatter* class is provided and all available formatter implementations need to be registered inside the *__init__.py* file in the *formatters* dictionary. One the one hand, a formatter subclass has to implement the *import_file* method, which, given a file path contained inside the provided CLI parameters, needs to read this file and return its contents interpreted in the standard *graph_data* dictionary format, specified in the previous Section 2.4.3. On the other hand, the *export_file* method needs to be implemented in such a way, that it will transform the input standard *graph_data* and write the data into a file, located at the user specified file path. This way the same logic of handling a specific file format, either importing or exporting the file, stays in one place and again, through the unified abstract class interface, the formatters are interchangeable, which makes the rest of the tool independant from any file format specification. The already provided *JsonFormatter* provides the implementation of this project's proprietary JSON file format. The detailed specification of that format can be found at Section 3.2.

2.5 Rendering Module

By utilizing the tool's build-in rendering module, the user can visually analyze the output data of the analysis module. The artifacts and their couplings will be rendered on screen in a node-edge based graph. The rendering module is composed of four consecutive steps, each contained in its own python file:

1. First of all the graph's data, which will be rendered, needs to be read from some kind of file. This happens inside *import_graph.py*, by calling the corresponding import and export formatter, based on the provided file type. More information about these formatters can be read in Section 2.4.4.

2. This raw data will the be processed by *graph_metadata.py* to extract useful statistics about the graph. For instance these statistics contain information about minimum and maximum edge weight. This information can then be used to directly display these possibly interesting statistics to the user or utilize them to improve the rendering to make it more comprehensible. This is partly achieved by making low weighted edges less opaque, with the help of the knowledge about maximum and minimum edge weight.

3. As the last step before actually drawing the graph, some filtering options and other user defined parameters are applied to the graph data inside *process_graph.py*. This allows the user to further simply the rendering and makes it more easy to extract the important bits of information.

4. The actual rendering, computed by *render_graph.py*, is the last step in this pipeline. Here nodes, edges, labels and other elements are rendered to the screen. The rendered graph's data and the actual layout (node placement) computation is done by the NetworkX[7] python package. Rendering this graph data to the screen and providing navigational UI elements, for the user to interactively explore the graph, are provided by the matplotlib[8] package.

Orchestrating these four steps and managing the dataflow between them is done in the rendering module's *__init__.py* file.

2.6 Interesting Algorithms

This chapter will provide some implementation details about the most sophisticated and interesting parts of the tool and about newly developed algorithms.

2.6.1 ArtifactStore

The *ArtifactStore* class is located in the *util* directory. On the one hand it contains all the code necessary to extract the actual artifact instances, that persist between commits, from the repository data. On the other hand it acts as a data structure that stores a complete list of the extracted artifacts, that will ever exist in some point in time in the repositories version history, while also providing useful utilities to all the other parts of the analysis tool for easy access and handling of the contained artifacts.

The main problems to be solved are:

1. How to uniquely identify an artifact, even though it exists across different commits with possibly different file names?

[7] `https://networkx.org`
[8] `https://matplotlib.org`

2. How to represent the abstract idea of a persisting artifact from the data perspective?

3. How to create the actual artifact instance from just the data of incoherent file modifications?

4. How to make this data accessible and usable?

The first two problems are addressed in the classes *Artifact* and *AttributedPath*. As the name indicates, the *Artifact* class will represent a single artifact across multiple commits.

To do so, it first needs to be uniquely identified. For that, each artifact instance holds a string ID, that consists of the commit hash, where this artifact appears for the first time in version history (by adding a new file to the repository), and also of the file path of the artifact during this first commit. This is indeed a unique ID, because, even though across the course of a projects version history, there can exist multiple artifacts at the same path at different points in time, there cannot be more than one artifact or file at the same file system path, at the exact same point in time. Thus combining a commit hash with the file path at that commit hash will always resolve to one unique file and therefore the artifact this file belongs to.

Besides its ID, an artifact also holds a record of every commit after it was added to the project and its file path at each of these commits accordingly. This makes it possible, as mentioned above, to get an entire artifact with its complete history, by only providing a single file path at a specific commit. All this history information is stored in a dictionary with the commit hashes as the keys, holding *AttributedPath* instances as their value. An *AttributedPath* holds the file path as well as the information, if the artifact is deleted at this commit. This behavior, that an artifacts persists, even after it was deleted, was not discussed in previous chapters and will come in handy later in this chapter.

The *Artifact* class also contains some methods to ease handling and constructing artifacts, for example checking if the artifact exists at a certain commit or to add a new entry to the history dictionary. Further details can be found in the well documented source code. The *Artifact* and *AttributedPath* classes also lie inside the *ArtifactStore.py* file, together with the actual *ArtifactStore* class.

The *ArtifactStore* itself addresses the last two problems listed above. It will be instantiated in the initialization stage of the analysis tool. It is important that it is created at this early stage, so it can create the artifacts on the basis of the raw, original commit list, that is read from the file system, to be able to track all artifacts and not miss the ones created in commits, that will be filtered or modified in the preprocessing stage. Especially the first (initial) commit in a git repository is likely to contain a huge amount of newly added files, and therefore artifacts, but would be considered a large commit and thus filtered out.

When iterating over all commits, every time a file addition happens inside a commit, a new artifact is created with the *register_new_artifact* method and the first history entry is added accordingly. From this commit onwards the artifact will be tracked and a few different things can happen to an artifact in each of these subsequent commits:

- If a renaming happens, the artifact gets a new history entry, containing the new file path inside the *register_artifact_rename* method.

- If the artifact is deleted in a commit, a new history entry is added, that has the *AttributedPath*'s deleted flag set to **true**. This only happens if the user does not configure the tool to **keep** all artifacts and ignore deletion. When this mode is selected, all artifacts will be kept as not deleted in every case and will persist until the very last commit. The change in analysis quality, when using this feature, is dependant on the analyzed project and needs to be individually evaluated, but is is generally not recommended to use this option as a default configuration, as it can have negative impacts on the analysis output data, because a lot of artifact information is discarded.

- If a new artifact would be registered, but there already is an previously deleted artifact, which happens to have the same path as the new one, the behavior is also dependent on the user's configuration. Additionally to the user configuration above, two other options are available instead:

  1. The default option is to **discard** deleted artifacts. This means that the existing artifact is kept in its deleted state and a completely new artifact is registered as described above.
  2. The other option for the user is to **reuse** artifacts. With this option in place, the existing artifact at that file path is just reincarnated, meaning the deleted flag is set back to **false**. This results in the artifact keeping already counted co-changes of previous commits, contrary to the behavior of the discard option, where this information is lost. The idea behind this is, that sometimes during development certain files get deleted, but are re-added again in a later commit, under the same file path. This can be interpreted as a structural change in the project, that has been reversed, thus the same artifact is just re-added to the project and should not be seen as a completely new one. Of course this interpretation can also be completely false and comparing the contents of the new and the old file or considering the time span between deletion and re-adding the file are both possibilities to improve the automatic detection of artifact reusability as part of future work.

  Because the **discard** option is the most intuitive way of handling artifact deletions, it is the default option of the three possibilities, but the **reuse** option has the possibility to lead to the best analysis data quality, because both other options throw away important data at some point. The possible consequences of using the **reuse** option, compared to the default option, are higher coupling values, as the old values are not discarded, and a lower amount of artifacts, as fewer have to be instantiated due to their reincarnation.

- Finally, if none of the above cases occur, the *AttributedPath* of the previous commit are copied and registered unchanged under the new commits hash.

This way all artifacts will always have a history entry for every commit after they are added and can therefore always be found by the analyzer. The main way artifacts are accessed later on by the actual change coupling algorithm is on a per commit basis. This means that artifacts are always retrieved by providing a specific commit hash and a file path, in this case to the *get_artifact* method the *ArtifactStore* has to offer. Therefore artifacts must be registered at every commit, to be accessible at every commit they exist in.

2.6.2 Preprocessing Algorithms

The preprocessing algorithms for splitting large commits and combining consecutive commits, which are briefly introduced in Section 2.4.2, will be described in more detail in this chapter. Both are newly proposed by this thesis and their inner workings will be explained with examples.

But before that, the classes *CouplingCommit* and *ModificationsBucket* are explained, as they provide the functionality to be able to filter out, split up and combine commits on the data level. Each of these three approaches (filtering, splitting and combination) holds its own problems, which prevents them from being able to work with the commit and modification data structures provided by PyDriller, hence the custom classes for commits and modifications:

- Commits can not just be filtered out of the complete list of commits. When this list is modified, subsequent preprocessing sub-stages would not be able to handle the deleted commits in any way, even if they could solve the problem why they were filtered out. Also it could be possible for some future coupling algorithms inside the *evaluation* stage to also utilize this metadata of deleted commits, and for that they still need to be accessible.

- When splitting up commits, instantiating new git commits that will take some of the file modifications, is not an easy task. Some metadata could be inferred from the commit that is split up, like author, committer or the branch it is contained in, but generating new commit hashes could create conflicts, as they need to be unique. Using the existing git hashes and extending them with some unique suffixes would solve the problem, as these custom hashes would be of different length than the hashes generated by git. But a possible argument against different sized commit hashes is that certain tools that further process the output data of this tool may be designed to work on these specific git hashes and could not handle the different length, but just in the scope of this tool, it would suffice for serving as a unique ID. Also randomly generated hashes would make analysis results non-deterministic and therefore not reproducible.

- Combining some commits, would also lead to the loss of information in the same way as for the filtering case, because the number of commits needs to decrease, but leaving out commits is not an option, as explained above. Also, combining commits by just copying the file modifications to another commit and deleting the original one leads to inconsistencies in the git history. It could lead to files being deleted, before they are created, renaming taking place on non-existing files and all other kind of different problems.

These problems make it clear, that adding, removing, reordering or modifying the original git commits is not a satisfying approach. The solution lies in custom data structures that extend the git commits.

The *CouplingCommit* class, which can be found in the *util/CouplingCommit.py* file, adds an *ignored* flag to each commit, which, if set to **true**, tells the coupling algorithm and other parts of the program, that this commit should be completely ignored. This removes the commit from analysis, while also keeping it accessible, for the case that its data has to be used nevertheless.

To address the problem of splitting commits, the *CouplingCommit* class now contains, in addition to the original list of modifications, a list of *ModificationsBucket*s. On the one hand, a *ModificationsBucket* contains a list of file modifications, which means that a *CouplingCommit* instance now contains a list of modification lists. This enables the preprocessing algorithms to split the original modifications list into sub-lists, which are then each stored inside a *ModificationsBucket*. The coupling algorithm will then consider only those modification as couplings, that lie in the same bucket. This way no new commit has to be generated, but still certain modifications of a commit can be de-coupled from modifications of the same commit. On the other hand the *ModificationsBucket* class also contains features to address the problem of combining commits together. Additionally to the list of modifications, an *ModificationsBucket* instance also contains a list of references to other *ModificationsBucket*s. One such reference is represented as a tuple consisting of a commit hash string and an index integer, which refers the unique *CouplingCommit* belonging to commit hash and the *ModificationsBucket* at the specified index of the bucket list of that commit. The *get_all_-modifications* method of a *ModificationsBucket* then returns a complete list of modifications, including its own modifications, as well as the modifications of all its referenced buckets. With this data construct it is easy to couple two commits, which were previously unrelated, by referencing all *ModificationsBucket*s of the first one to the appropriate buckets of the second one. A coupling algorithm will then consider all modifications coupled, that are returned by one *ModificationsBucket*'s *get_all_modifications* method. To not count the couplings of the referenced commit two times, the referenced commit has to be marked as ignored. Also more complex referencing constructs are possible, if they are needed by any future algorithm. It is important to note that the *get_all_modifications* method will not recursively call the *get_all_modifications* method of its references, but just take their raw modifications list directly. This is intentional, as it prevents any possible reference circles from being called infinitely.

Splitting Large Commits

The functionality of splitting, in other words de-coupling, a commits list of modifications will be demonstrated with the preprocessing sub-stage that splits large a large list of file modifications based on their path, as already outlined in Section 2.4.2.

For instance, an example commit contains the following modified files, that now need to be separated from each other:

```
index.html
src/css/main.css
src/css/button.css
src/img/bg1.jpg
src/img/bg2.jpg
src/img/bg3.jpg
```

In this example the threshold for identifying a large commit shall be set to *3*, which means this commit will be handled by this sub-stage. The algorithm will now try to split the modifications on the bases of which directory they belong to in, beginning at the top directory of the analyzed project and then going deeper into the folder structure.

This means the path of each file will be traversed from left to right and files belonging to different directories, will be split up into different groups. In the example above, *index.html* is the only file that lies in the top-level directory and is therefore separated into its own group. The other five files are all located inside the *src* directory and thus grouped together. After each of these grouping iterations, groups of modifications that contain less files than the large commit threshold are not split up any further and put into a *ModificationsBucket*, which is added to the commits list of *ModificationsBucket*s. In this example, the first group, containing only the *index.html* file, satisfies this condition and will therefore no longer be treated by this algorithm. The group belonging to the *src* directory on the other hand still contains *5* files, which is not less than the threshold of *3* and will therefore undergo the next grouping round.

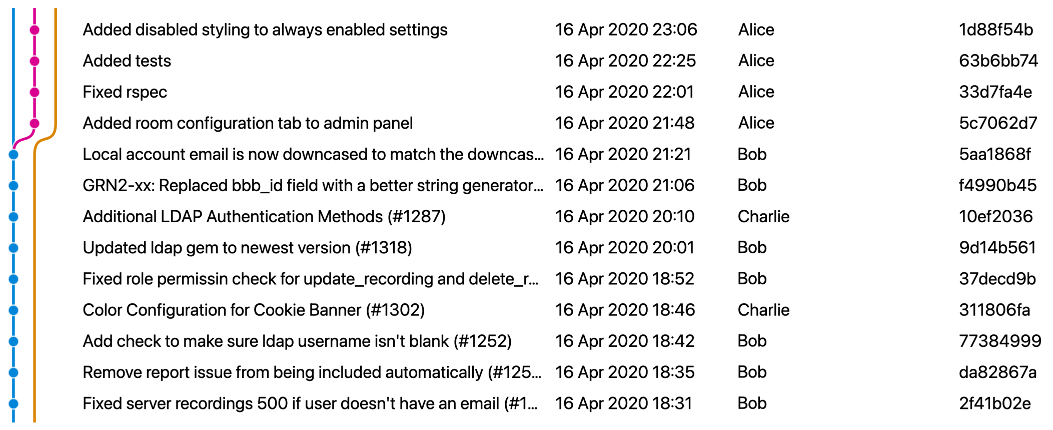
This time the two css files are separated into the *src/css* group and the three jpg files now belong to the *src/img* group respectively. This results in the two css files being put into a *ModificationsBucket*, because the *src/css* group contains less than **3** files, which is the threshold. The three jpg images however do not undercut the threshold, but also can not be separated any further, as there are no more subdirectories inside the *img* directory. The algorithms behavior on how to proceed in these kind of cases, is dependant on the user's configuration. If the user set the flag to filter out large commits to true, than groups that can not be divided any further, are discarded. Otherwise, as this is the smallest possible grouping, even those larger groups of modifications are put into a *ModificationsBucket*.

It is recommended to also use this option when filtering out large commits, as it can try to retain some couplings, that would be lost otherwise. Contrary to this recommendation, it can also be argued, that files belonging to the same folder should inherently stand in some kind of relation from a system architecture point of view and therefore the couplings added by this method to these already related files do not add any informational value. But this also implies, that even from this skeptical perspective, the analysis output data is not worsened and can therefore only possibly be influenced in a good way.

Combining Consecutive Commits

The algorithm of for combining consecutive commits, in other word coupling previously unrelated commits, provides an example use case of the reference feature of the *ModificationsBucket* class. As already explained in Section 2.4.2, this algorithm finds successive commits of the same developer, that all lie in a user configurable time window.

The explanation of the implementation is performed on the example data of an extract of a real world version history, which can be seen below in Figure 2.1:



Added disabled styling to always enabled settings	16 Apr 2020 23:06	Alice	1d88f54b
Added tests	16 Apr 2020 22:25	Alice	63b6bb74
Fixed rspec	16 Apr 2020 22:01	Alice	33d7fa4e
Added room configuration tab to admin panel	16 Apr 2020 21:48	Alice	5c7062d7
Local account email is now downcased to match the downcas...	16 Apr 2020 21:21	Bob	5aa1868f
GRN2-xx: Replaced bbb_id field with a better string generator...	16 Apr 2020 21:06	Bob	f4990b45
Additional LDAP Authentication Methods (#1287)	16 Apr 2020 20:10	Charlie	10ef2036
Updated ldap gem to newest version (#1318)	16 Apr 2020 20:01	Bob	9d14b561
Fixed role permissin check for update_recording and delete_r...	16 Apr 2020 18:52	Bob	37decd9b
Color Configuration for Cookie Banner (#1302)	16 Apr 2020 18:46	Charlie	311806fa
Add check to make sure ldap username isn't blank (#1252)	16 Apr 2020 18:42	Bob	77384999
Remove report issue from being included automatically (#125...	16 Apr 2020 18:35	Bob	da82867a
Fixed server recordings 500 if user doesn't have an email (#1...	16 Apr 2020 18:31	Bob	2f41b02e

Figure 2.1. An extract from a real version history. Developer names are obfuscated.

Commit messages and branches are not of interest in this case, only developer names and the commit dates are important. It is important to note, that the analyzer traverses the commit list beginning at the oldest commit, but the commits in the example are sorted the other way around, as most applications showing a visual representation of a version history prefer it this way, to show the newest commits on top of the list. But for this particular algorithm, the output stays the same for both ways of sorting, as the algorithm is based on their commit date. Thus, for the sake of simplicity, the example commit list will be traversed from top to bottom, even though this does not match the behavior of the analysis tool.

The user input required for this preprocessing stage is a time intervall. Many different possibilities are offered to the user, as the time intervall can be defined using seconds, minutes, hours and days. For this example it is assumed, that the specified time intervall is **40 minutes**.

Now the commit list is iterated one by one, beginning at the first commit from Alice at 23:06h. First of all every looked at commit must not be split up by the previous preprocessing sub-stage, so that it only contains one *ModificationsBucket*, which will come in handy later on, as it keeps the reference structure simple. The algorithm will then, for every commit, look at previously processed commits and check if the following conditions apply:

1. The current commit has to be made by the same developer than the preceding commit.
2. The (absolute) time intervall between the two commits has to be less or equal than the specified time intervall.
3. The preceding commit has to be either already been combined with an even earlier looked at commit or the amount of modifications of both commits combined does not exceed the large commit threshold, if large commits would have to be filtered out.

This last part of the third condition is in place, because combining/referencing lists of modifications may break the constraint of the preprocessing sub-stage, that filters out large commits, because a new large commit would be created after the execution of that filtering sub-stage. Again, this is an example of how the preprocessing stages can interfere with each others goals and have to be individually adjusted to comply with all previously executed sub-stages.

Back in the example, the first commit does not have any predecessors and is therefore not processed by this algorithm. The second commit at 22:25h on the other hand has a predecessor, the first commit, and therefore the list of conditions is checked. The first condition applies, as both commits are made by Alice, but the second conditions fails, as the time intervall between the first and the second commit is 41 minutes, which is more than the specified 40 minutes. This results in the first and second commit to not be combined. However, when the algorithm looks at the third commit from Alice at 22:01h and compares it with the previous commit, the time intervall between them is 24 minutes and therefore the second condition is also fulfilled. In the example it is assumed, that large commits are not filtered out, because Figure 2.1 does not provide any information about the modifications made in each commit. So the third condition will always be satisfied in this example.

This leads to the third commit being combined into the second commit, which has the following procedure:

1. As mentioned above, both commits only have one *ModificationsBucket* and the one of the third commit is now being referenced in the *ModificationsBucket* of the second commit.
2. The third commit is then being flagged as ignored, so that its modifications will not be counted twice later in the algorithm (one time in each of the two commits).

With that the third commit is successfully merged into the second commit and their modifications are now coupled together.

When the fourth commit is now analyzed, comparing it with the previous third commit, suffices the first two conditions, because both commits are made by Alice and lie 13 minutes apart. Also the third condition applies, because the third commit has already been combined with second commit. In this case however, the third and fourth commit will not just been coupled together, due to the third commit being ignored. Referencing the fourth commit in the third commit and ignoring it, would remove it completely from analysis, because the previously mentioned *get_all_modifications* method of a *ModificationsBucket*, which in this case is called on the not ignored second commit, will not recursively collect chained references and therefore would not pick the fourth commit, when it is referenced in the third commit. Also the third commit is ignored, so the modifications of the fourth commits will also not be picked up there. Bypassing this problem by setting the ignore flag on the third commit to false again will lead to the modifications of the third commit to be counted twice and is therefore not an option either. The solution is, that the algorithm will in those case go back one commit in the version history and will now compare the fourth and the second commit. The comparison between those two also satisfies all three conditions, as they are 37 minutes apart. This means the algorithm will go back even further and now compare the fourth with the first commit, but as the time intervall between those two is 78 minutes, the second condition fails. This means the last commit, where all three conditions are met, was the second commit and because the second commit is not ignored, the fourth commit will be combined with it. The second commits only *ModificationsBucket*, now hold two references, to the third and fourth commit respectively.

In fact the algorithm tried the same back then, when successfully comparing the third and second commit, and tried to compare the third and first commit. But this also failed due to the two commits lying 65 minutes apart, breaking the seconds condition.

To summarize the algorithms behavior, continuous chunks of commits, made by the same developer and all contained inside the specified time intervall, will be combined into the first commit of this chunk.

In particular this means, that in this example the first two commits of Bob (21:06h and 21:21h) will be combined. Also the last three commits of Bob will be combined (18:31h - 18:42h), but even though the commit from Bob made at 18:52h would also fit into the time intervall of that last chunk of commits, the intermediate commit from Charlie at 18:46 prevents this commit from being combined with the other three.

It has to be noted, that many of the decisions made in this algorithms design are good candidates to be improved in future work. For example finding ways to reliably reference commits, that have been split by previous sub-stages, to enable them to be considered by this algorithm. Also the branches of two combined commits can be different at this point, which invalidates the whole purpose of combining semantically connected, incomplete commits.

Due to these and possibly more problems, the usage of this preprocessing stage can not be generally recommended. The results also differ between different projects and development workflows and are highly dependent on the user input time intervall. But when evaluated in detail on a per project basis, this algorithm can be an important first step to counteract the aforementioned bad commit practices and improve analysis results.

2.7 Performance

During the planning and early development phases of this software, performance was not a major consideration and had a very low priority, because at that point in time, the data mined from each git commit, did not contain the file modifications and the need for a huge data structure like the artifact store was not apparent.

But it was exposed in later development, when implementing the actual coupling algorithms, that actually getting the modifications of a commit is handled by PyDriller by calling the *git diff* command between the commit and its predecessor. This is a very expensive operation in terms of computational time. And even though this tool only works with the file names contained in the commits list of modifications, the call of *git diff* would always also calculate the lines of code that changed, which also involves cryptographic computations. Additionally to this CPU intensive task, the complete modification data, containing file modifications, language specific syntax trees of the code and other additional computations made by PyDriller, turned out to be very memory intensive too. Even though all this behavior can not be completely changed and the differences of the files contents is always been calculated and copied to memory, two performance improvements have been implemented to counteract this behavior. Both improvements take place in the second sub-stage of the initialization stage, where the *CouplingCommit*s are created from the git commits:

1. The method called on each git commit when calculating the list of modifications has been altered by the *CouplingCommit* class. It is adjusted to no longer perform most of the cryptographic calculations made with the blobs, which represent the files contents. But as the changed lines of each file modification is still calculated, the performance gains are limited. To completely stop to the file modifications from being calculated, the source code of PyDriller has to be modified.

2. When instantiating a *CouplingCommit*, all data, that is not needed by this tool is discarded to free up memory. Especially all data related to the actual file contents of each modification did make a huge difference here and lowered RAM usage by roughly 70%.

As the performance improvements on the CPU side were not significant, efforts were made to parallelize parts of the program, especially the initialization phase. Theoretically, this would have been possible, as all sub-stages are conceptually just iterating over the list of commits, which could be done in parallel. But the two slowest parts of the program, unpacking the git commits modifications and initializing the *ArtifactStore*, are unable to be distributed to different cores of the cpu, at least in the simple context of iterating through different chunks of the commit list in different threads. On the one hand, the mining of the repository data on the low level, where the git command line is used, concurrent access is simply not possible, as native libraries are crashing, causing errors on the python side, that can not be resolved by the tool itself, because there is no alternative on accessing the git files. This ultimately leads to the program being aborted by python. On the other hand, when parsing the git history to create the histories of each artifact inside the *ArtifactStore*, the order of the commits must be strictly adhered, as discussed in Section 2.6.2. This invalidates the idea of parallelizing the commit iterations, by diving the list into chunks, but as part of future work, the viability and performance implications of a more sophisticated, parallelized queue can be explored.

With the above performance improvements in place, a computer equipped with good to high end hardware, is able to analyze most repositories in a reasonable amount of time. The computer this tool is developed and tested on has the following specifications:

- Intel i9-9980HK
- Dual channel 32GB DDR4-RAM with 2666 Mhz
- High performance NVMe SSD
- macOS 11.2.3

All tests are made using repositories that are cloned to the local hard drive. The following orders of magnitude in terms to repository size are provided to convey the general idea of how long it takes to analyze repositories of certain sizes and how much memory is used. The real time to it takes to compute the coupling analysis is subject to many parameters, like commit size, commit practices and many more. The many options provided by this tool to counteract some of these problems all have different performance implications, and need to be individually adjusted for best performance on a per project basis.

- With the hardware mentioned above, small repositories with less than 1000 commits can be analysed in a matter of seconds. The memory usage of a few hundred megabytes is negligible.
- Also, repositories below 5000 commits will generally take less than two minutes and will take up 1-2 GB of RAM.
- Repositories around the 10000 commit mark will take around 8 to 10 minutes and can take up to 10 GB of memory.
- When analyzing repositories with around 20000 commits, it can take 40 minutes or more and uses above 40 GB of memory. When exceeding the RAM capacity, that is available on the system, memory compression and the use of swap files can increase the analysis time significantly.

Everything above this scale should be performed on high performance compute clusters, especially the amount of memory needed will exceed most personally owned computers. As mentioned above, further time and development should be invested into parallelizing the code, when regularly analyzing very large repositories. For testing purposes, the analysis of the repository of the linux kernel with around one million commits was attempted on a dedicated root server with 128 GB of RAM, but the program crashed during the initialization of the *ArtifactStore*, due to memory problems. Up to this point of crashing, unpacking the git commits took around 12 hours.

Even the vague orders of magnitude provided above, that show the amount of memory used by the tool, indicate a more than linear size of memory needed, which makes the analysis of very huge repositories impractical, but an idea to solve this issue as part of future work and is provided in Chapter 6.

Chapter 3

Documentation

Additionally to the completely commented source code of this project, this chapter will explain all information needed to install and operate the tool. Also the specification of the proprietary file format, that is used by the analysis and rendering modules, is provided.

3.1 Usage Explanation

3.1.1 Quick Start

Section 2.1 already explained the technologies used in this project and for the following usage instructions to work, **git** and **python3** need to be installed on the system and working. Also this project has to be already cloned to the local machine. All command line interface calls described below are performend from the root directory of this project. It is important to note, that using a python virtual environment[1] is advised, but not described in the following examples, as it is a user preference and not a technical requirement. The python dependencies are installed by calling

```
$ pip3 install -r requirements.txt
```

After that, the application is ready to be launched by calling

```
$ python3 mining analysis
```

for analyzing a repository or

```
$ python3 mining render
```

to render the data of a previously executed analysis.

3.1.2 Advanced Usage

In Section 2.2.2 a brief introduction to the use of command line options is provided, but this tool also provides an interactive shell for entering all required parameters, when just using the bare command line calls from above. But for most use cases providing all the different options directly at launch is the preferred way to go. Therefore a complete

[1] `https://docs.python.org/3/library/venv.html`

list of all options is provided below. An option is added to the command line call by either providing its full name prefixed by two dashes or by using the short name version prefixed by a single dashed. After providing the options name, in either of the two styles, the value of that option is typed into the command line, separated by a blank space.

Option Name	**Short Name**	**Type**	**Description**
Required options:			
–repository	-r	Path or Url	The repository to be analyzed. Either provide a local path to a git repository folder or a remote url to a publicly available repository.
–algorithm	-al	String	The change coupling algorithm used in the analysis. Possible values and details can be found in Section 2.4.3.
–output	-o	Path	The path, where the output file is written to. An existing file will be overwritten.
Commit filtering on the git level:			
–branch	-b	String	Limit the commits that are analyzed to a specific branch.
–filetype	-f	String	Only consider file modifications, whose file ending matches one of the specified filetypes. Can be supplied **multiple** times.
–author	-a	String	Only consider commits made by one of the specified authors. Can be supplied **multiple** times.
–commit	-c	String	With this option, specific commit hashes can be supplied and only matching commits are analyzed. Can be supplied **multiple** times.
–fromdate	-fd	String	A date string needs to be supplied with format yyyy-mm-dd. Only commits made after this point in time are analyzed.
–fromcommit	-fc	String	A commit hash needs to be supplied. Only commits made after this point in time are analyzed.
–fromtag	-ft	String	A string identifying a tag in the git history has to be supplied. Only commits made after this point in time are analyzed.

Option Name	Short Name	Type	Description
–todate	-td	String	A date string needs to be supplied with format yyyy-mm-dd. Only commits made before this point in time are analyzed.
–tocommit	-tc	String	A commit hash needs to be supplied. Only commits made before this point in time are analyzed.
–totag	-tt	String	A string identifying a tag in the git history has to be supplied. Only commits made before this point in time are analyzed.
–nowhitespace	-nw	Boolean	When provided, modifications containing only whitespace changes, will be filtered out and not analyzed. This is a **flag**, meaning it simply has to be provided without a value to be considered true, and is false otherwise
Commit preprocessing:			
–largethreshold	-lt	Integer	The threshold used to identify commits that are considered to be large commits. The **default** value is 50.
–nolarge	-nl	Boolean	When provided, large commits will be filtered out and not analyzed. Large commits are identified using the *largethreshold* option. This is a **flag**, meaning it simply has to be provided without a value to be considered true, and is false otherwise
–splitlarge	-sl	Boolean	When provided, large commits will be divided into smaller chunks, if possible. Large commits are identified using the *largethreshold* option. This is a **flag**, meaning it simply has to be provided without a value to be considered true, and is false otherwise
–nomerge	-nm	Boolean	When provided, merge commits will be filtered out and not analyzed. This is a **flag**, meaning it simply has to be provided without a value to be considered true, and is false otherwise

Option Name	**Short Name**	**Type**	**Description**
–combineconsecutive	-cc	String	A string indicating a time unit has to be supplied, in the form of an arbitrary number followed by either **d** for day, **h** for hour, **m** for minute or **s** for second. Commits that are committed in the same specified time intervall are combined and considered one single commit.
		Other options:	
–handledeletedfiles	-df	String	Either **keep**, **discard** or **reuse** has to be supplied. This option changes the internal behavior of the artifact store, which is explained in Section 2.6.1.
–fileformat	-ff	String	Specifies the file/data format of the output file. The **default** value is inferred, which means the format is automatically detected based on the file ending of the supplied output file path.

The command line options for the rendering module are listed below:

Option Name	Short Name	Type	Description
Required options:			
–input	-i	Path	The path to the input file, that contains the output data of the analyzer.
Node and edge filtering:			
–filetype	-f	String	Only show nodes (artifacts), whose file ending (label) matches one of the specified filetypes. Can be supplied **multiple** times.
–inspectfile	-if	String	A regex expression, which any nodes id is matched against and filtered out when not matched.
–minweight	-w	Float	The minimum weight of an edge. All edges with smaller weights are not rendered.
–minvalue	-v	Float	The minimum value of a node. All nodes with a smaller value are not rendered.
–weightpercent	-wp	Float	Only show the top X percent of all edges with the highest weight.
Other options:			
–fileformat	-ff	String	Specifies the file/data format of the output file. The **default** value is inferred, which means the format is automatically detected based on the file ending of the supplied output file path.

For example a completely configured command line call for the analysis could look like this:

```
$ python3 mining analysis -r cache/kieker/ -o kieker.json -al urc -cc 5m -nl -nm -nw
-sl -df reuse
```

And accordingly for the renderer:

```
$ python3 mining render -i ./kieker.json -v 50 -w 10
```

Additionally it is possible to use this tool in an automated environment, an example of which can be found in Section 4.3.

3.2 File Format Specification

The standard file format used by this tool is a normal JSON file, that contains the dataset specified in this chapter. It is used for the output file of the analysis module and the input file of the rendering module. It is virtually identical to the format of the *graph_data* presented in Section 2.4.3 Thus the following specification is reduced to field names and their types, because the semantics of the contained fields is already explained in detail in the mentioned chapter. The JSON object encoded in the file has the following three fields:

1. An array of **nodes**. Each node represents an artifact and is a JSON object with the following fields:
   - **id**: unique string
   - **label**: string
   - **value**: number
2. An **edges** array, containing edges in the form of objects. Each edge has the following fields:
   - **id**: unique string
   - **directed**: boolean
   - **start**: string
   - **end**: string
   - **weight**: number
3. An **cli_args** object. This is the only metadata, that is not contained in the internal *graph_data*. Its purpose is to provide the complete set of command line option, that were used for the corresponding analysis run, to make it reproducible. Its keys can be deduced from the options table in Section 3.1.2 but are listet here for the sake of completeness:
   - **algorithm**: string
   - **author**: string array or null
   - **branch**: string or null
   - **commit**: string array or null
   - **combineconsecutive**: string or null
   - **fileformat**: string
   - **filetype**: string array or null
   - **fromcommit**: string or null
   - **fromdate**: date string or null

- **fromtag**: string or null
- **handledeletedfiles**: string
- **largethreshold**: number
- **nolarge**: boolean
- **nomerge**: boolean
- **nowhitespace**: boolean
- **output**: string
- **tocommit**: string or null
- **todate**: date string or null
- **totag**: string or null
- **repository**: string
- **splitlarge**: boolean

This format was created to fulfill the following two goals:

1. To provide a very simple, minimalist format, that has no overhead in terms of features and file size.
2. To make the data easily accessible by all kinds of different, external applications and tools.

The first goal is achieved by just converting the *graph_data* to JSON and adding a few lines of extra metadata. And the second goal is achieved by the formats easy structure and the use of the popular, standardized JSON format.

Using an already existing graph file and data format, would add unnecessary complexity to such a simple tool, which this JSON format does not. But as already mentioned, third party formats can easily be implemented and added to the tool, if needed.

One additional benefit of this proprietary JSON format is its extendibility, as additional fields and metadata can be added to a file, even for single nodes or edges, without breaking the format specification. For example this enables third party tools to further analyze and enrich the files produced by this analyzer, while maintaining the files compatibility with the rendering module.

Chapter 4

Examples

In this chapter some example runs of the tool will be performed on publicly available repositories. The results will then be described shortly, but interpretations and statistical evaluations are not a part of this thesis. All examples are run on the same hardware as described in Section 2.7

4.1 Kieker

The first example analysis is performend on the Kieker Monitoring Framework [van Hoorn et al. 2012; Hasselbring and van Hoorn 2020], which was briefly mentioned in Section 1.2. It is a tool performing a dynamic coupling analysis by monitoring a system at runtime, implemented in Java[1]. The repository can be found at `https://github.com/kieker-monitoring/kieker`. At the time it was analyzed it contained 7.507 commits. The analysis took 37 minutes and 9 seconds and used 8,89 GB of system memory in total. The command line options used are:

–algorithm urc

–nolarge

–nomerge

–nowhitespace

–splitlarge

–combineconsecutive 5m

–handledeletedfiles reuse

Output file and repository location are dependant on the file system of the user and are therefore not mentioned. With this list of parameters, this exact run can be repeated and verified, by supplying the **–tocommit 68fba3886f44f26d6a94b974065edf73d58c8822** option, which represents the latest commit present during this analysis run. The possibility of repeating a run applies to all the following examples as well and the latest commit hash

[1]Official Java website: `https://www.java.com`

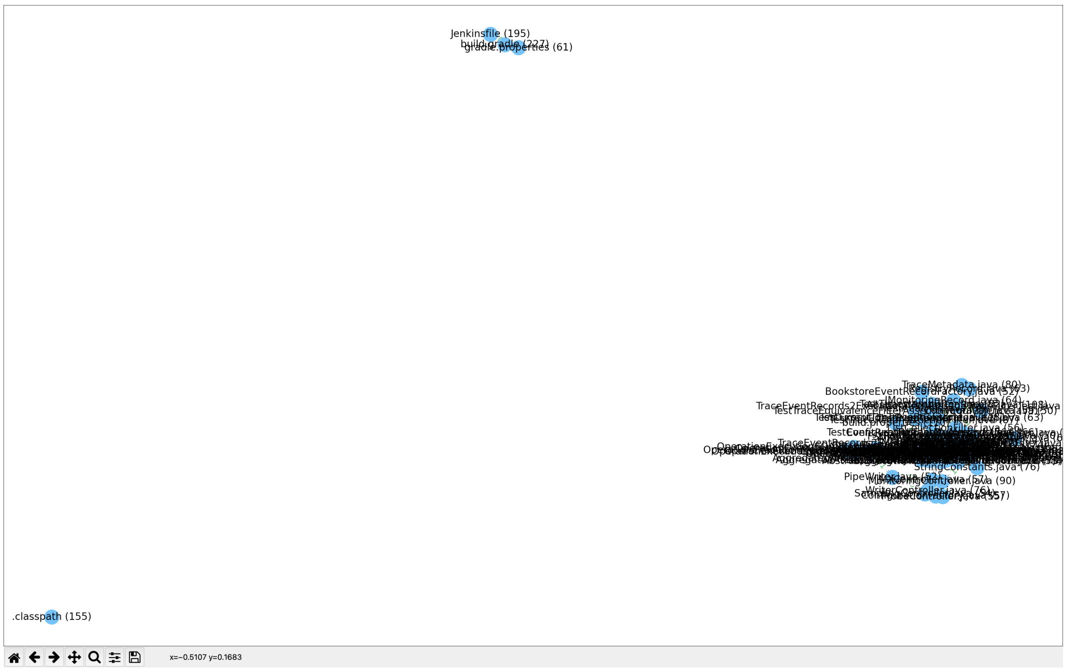
will always be provided. It is to note though, that the rendering of the graph uses a non deterministic node layout and can therefore differ from run to run, but the underlying data is identical This enables to reader to interactively experience all examples for himself/herself.

This specific analysis run will result in 5.223 artifacts and 123.648 couplings. Displaying such a huge graph with the included rendering module is generally not possible, as the pixel rendering it is not GPU accelerated. Even if it would somehow be displayed, the entire monitor of the user would be filled with overlapping nodes, edges and labels and no valuable information could be extracted. To solve this problem, the graph, that will be rendered, should be filtered beforehand. The rendering module provides these filtering options on a command line input level, as described in Chapter 3. The command line output of the renderer also displays the number of nodes and edges, that will be rendered, prior to actually rendering them. This makes it possible to experiment with the filtering values, without having to wait for each rendering to complete. To get a general idea of the artifacts change values of the nodes and the coupling weights of the edges, a look into the output file of the analysis can be helpful. With these methods, a reasonable filtering was found for this analysis with the following options for the renderer:

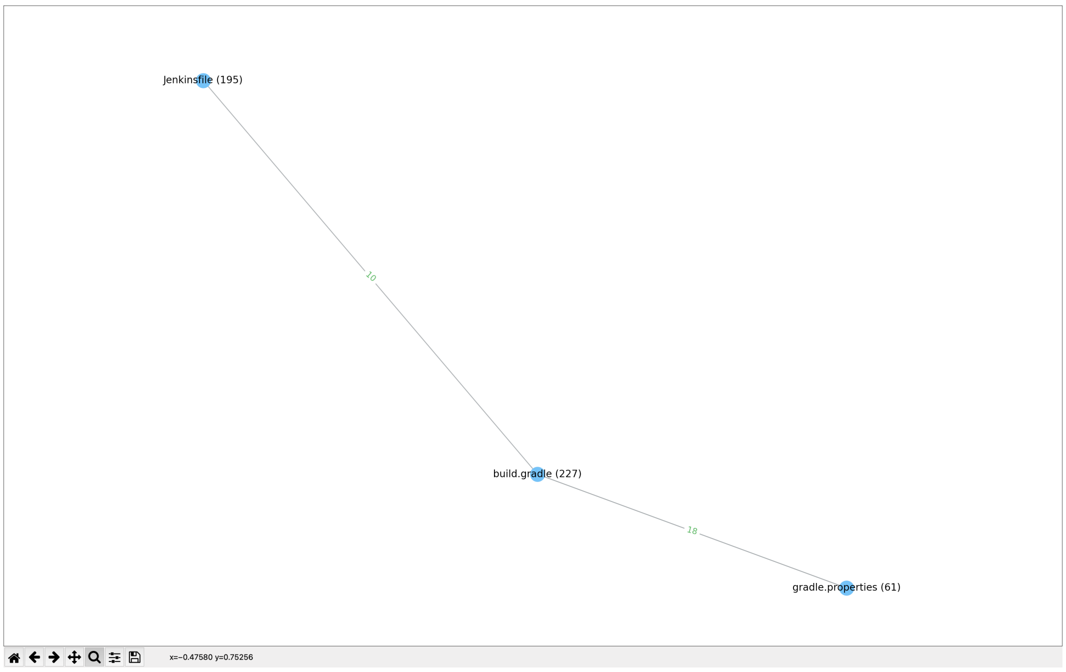
–minvalue 50 to only show artifacts with at least 50 changes.

–minweight 10 to only show couplings with a weight of at least 10.

This results in a graph with 83 nodes and 728 edges, which is comprehensible, when displayed to the user. The complete rendering can be seen below:

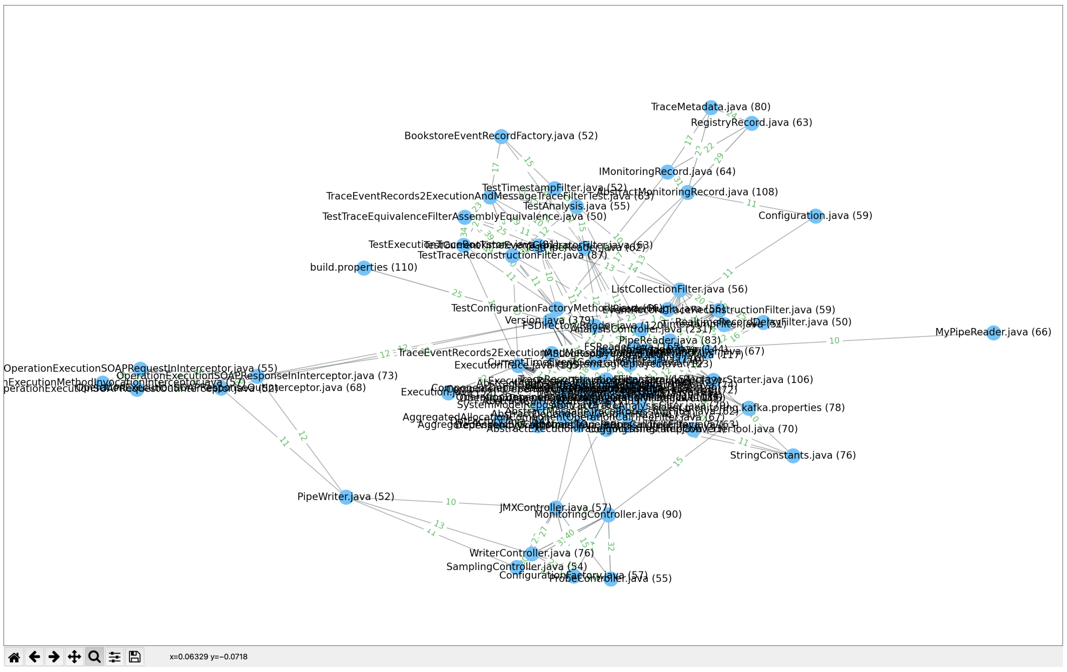

Jenkinsfile (195)
build.gradle (227)
gradle.properties (61)
TraceMetadata.java (80)
BookstoreEventRecordFactory.java (52)
StringConstants.java (76)
MonitoringController.java (90)
WriterController.java (76)
.classpath (155)
x=−0.5107 y=0.1683

The layout algorithm used by the rendering module, will position nodes connected by edges with higher weights closer together. This results in the graph being divided into three clusters. The first cluster in the bottom left corner is just a single *.classpath* artifact with 155 total changes. This file is not connected to other files, because all its coupling weights are below the *minweight* threshold of 10. The second cluster on the top of the rendering consists of the three files *Jenkinsfile*, *build.gradle* and *gradle.properties* and can be seen in a zoomed in view below:

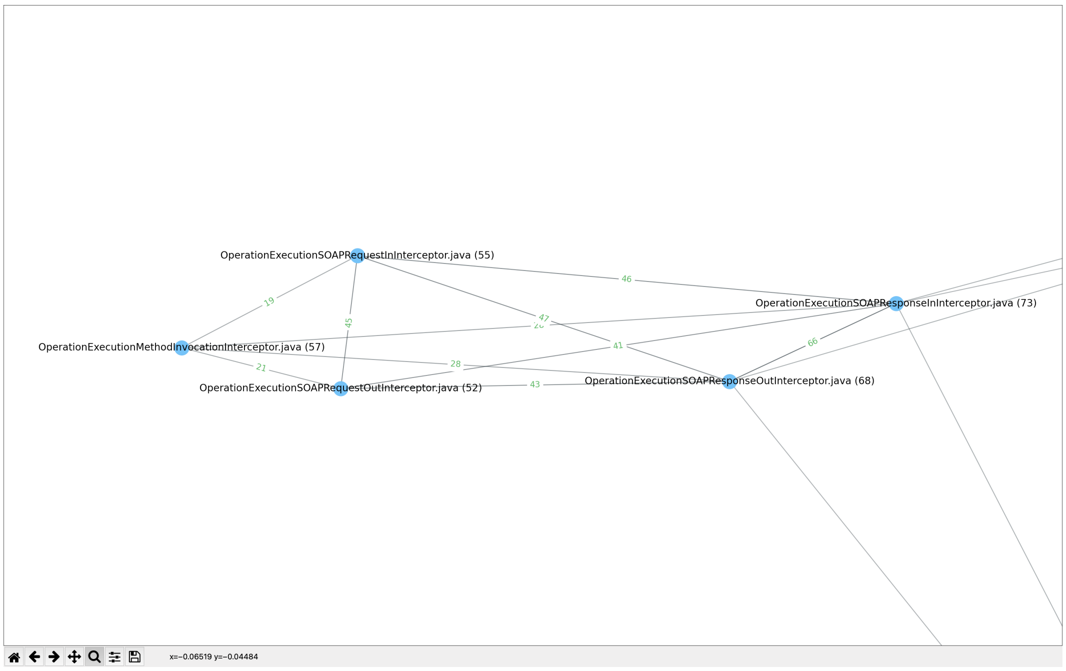


This screenshot also shows the highlighted button in the bottom toolbar, that is used to zoom into a graph. But all three files of the second cluster not not contain any code, but are configuration files for build and continuous integration tools.

However zooming in to the large, third cluster in the bottom right corner in the following figure, reveals interesting sub-clusters of actual *.java* source code files:


TraceMetadata.java (80)
RegistryRecord.java (63)
BookstoreEventRecordFactory.java (52)
IMonitoringRecord.java (64)
TestTimestampFilter.java (52)
AbstractMonitoringRecord.java (108)
TraceEventRecords2ExecutionAndMessageTraceFilterTest.java (65)
TestAnalysis.java (55)
TestTraceEquivalenceFilterAssemblyEquivalence.java (50)
Configuration.java (59)
TestTraceReconstructionFilter.java (87)
build.properties (110)
ListCollectionFilter.java (56)
TestConfigurationFactoryMethods.java
Version.java (379)
AnalysisController.java (231)
PipeReader.java (83)
MyPipeReader.java (66)
OperationExecutionSOAPRequestInInterceptor.java (55)
OperationExecutionSOAPResponseInInterceptor.java (73)
TraceEventRecords2Execution
ExecutionTrace.java
Starter.java (106)
kafka.properties (78)
Tool.java (70)
StringConstants.java (76)
PipeWriter.java (52)
JMXController.java (57)
MonitoringController.java (90)
WriterController.java (76)
SamplingController.java (54)
ConfigurationFactory.java (57)
ProbeController.java (55)
x=0.06329 y=−0.0718

The left sub-cluster for example, which is shown in detail below, contains only artifacts with similar file names, indicating that the calculated couplings do in fact capture real relationships between artifacts. The two files on the right of that sub-cluster for instance are very tightly coupled, as they co-changed 66 times, while only changing 68 and 73 times in total respectively:


OperationExecutionSOAPRequestInInterceptor.java (55)
OperationExecutionSOAPResponseInInterceptor.java (73)
OperationExecutionMethodInvocationInterceptor.java (57)
OperationExecutionSOAPRequestOutInterceptor.java (52)
OperationExecutionSOAPResponseOutInterceptor.java (68)
19
45
46
47
41
66
21
28
43
x=-0.06519 y=-0.04484

Such a graph can hold much more information than outlined in this example, but it conveys the general concept of visually and interactively exploring the analysis results. The third cluster in particular still contains many more sub-clusters and a huge bulk of artifacts in the center, which should be split up even further by using different filtering options.

4.2 ExplorViz

In this second example, one repository of the ExplorViz project stack will be analyzed. It will be used to present more renderer options to filter the incomprehensible amount of nodes and edges. ExplorViz is also a tool to perform dynamic coupling analysis, by visualizing the communication between software modules at runtime in a 3D environment [Hasselbring et al. 2020; Fittkau et al. 2017]. The repository analyzed here contains the backend code written in Java and can be found at `https://github.com/ExplorViz/explorviz-backend`.

By the time this analysis in performed, the repository contained 1264 commits, with the latest commit being **7cd4b189797a5f0117bab0d33834aea8237cd321**. The command line options used are the same as for the previous analysis, with the only difference of using the directed raw counting algorithm:

–algorithm drc

–nolarge

–nomerge

–nowhitespace

–splitlarge

–combineconsecutive 5m

–handledeletedfiles reuse

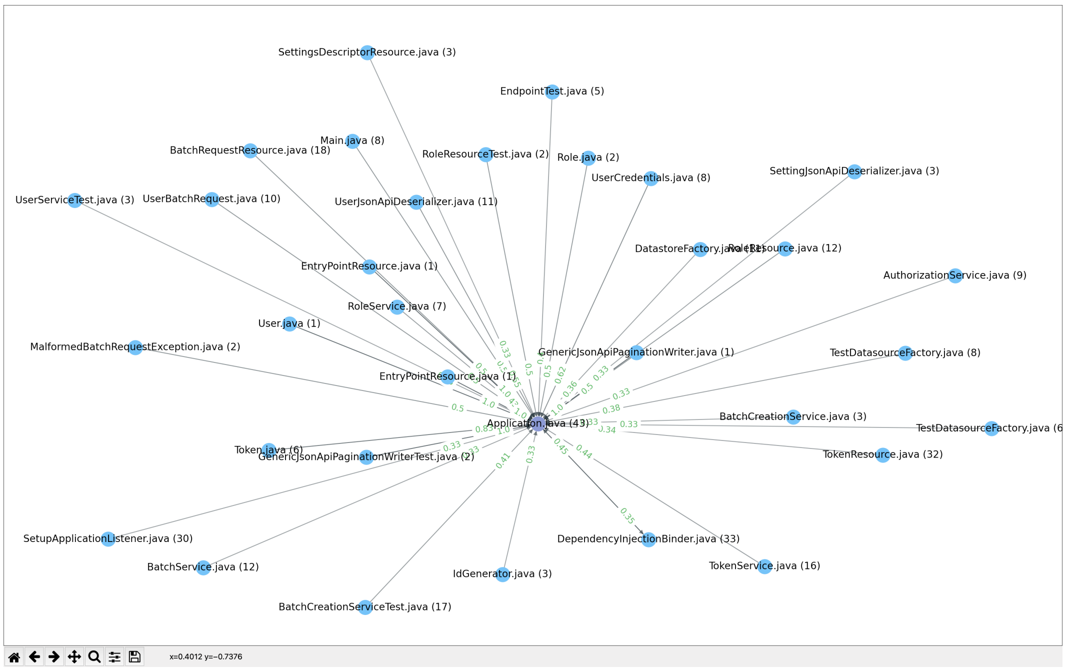
The analysis took 28 seconds and needed 300 MB of memory. The complete graph contains 498 nodes and 8988 edges. Two options will be used to reduce a amount of data, that is rendered on the users screen. The first one being the **–filetype .java** option, which will only show artifacts, that have the *.java* file ending, thus only analyzing actual source code files. For example the Jenkins and Gradle configuration files, which are contained in the rendering of the Kieker analysis example, would have been filtered out with this option in place. Note that this command line option can be supplied multiple times, so that projects consisting of multiple languages can still make use of this feature. This brings the rendering down to 199 nodes and 3854 edges, which is still still too complex. When taking a quick look ito the source code of the repository, it becomes clear, that the entire backend consists of many small services. The user can now pick out certain files, by providing a regex to the **–inspectfile** option. All artifacts file paths will be checked against this option and only those nodes who pass the test, or those who are coupled to a passing node, will be rendered. In this example the artifact at file path *authentication/src/main/java/net/explorviz/security/server/main/Application.java* has been picked, as it has a relatively high count of changes. An additional filtering is done to remove low weighted couplings, as suggested by Oliva and Gerosa [2015]. So the complete list of rendering options is the following:

–filetype .java

–inspectfile authentication/src/ [...] /security/server/main/Application.java

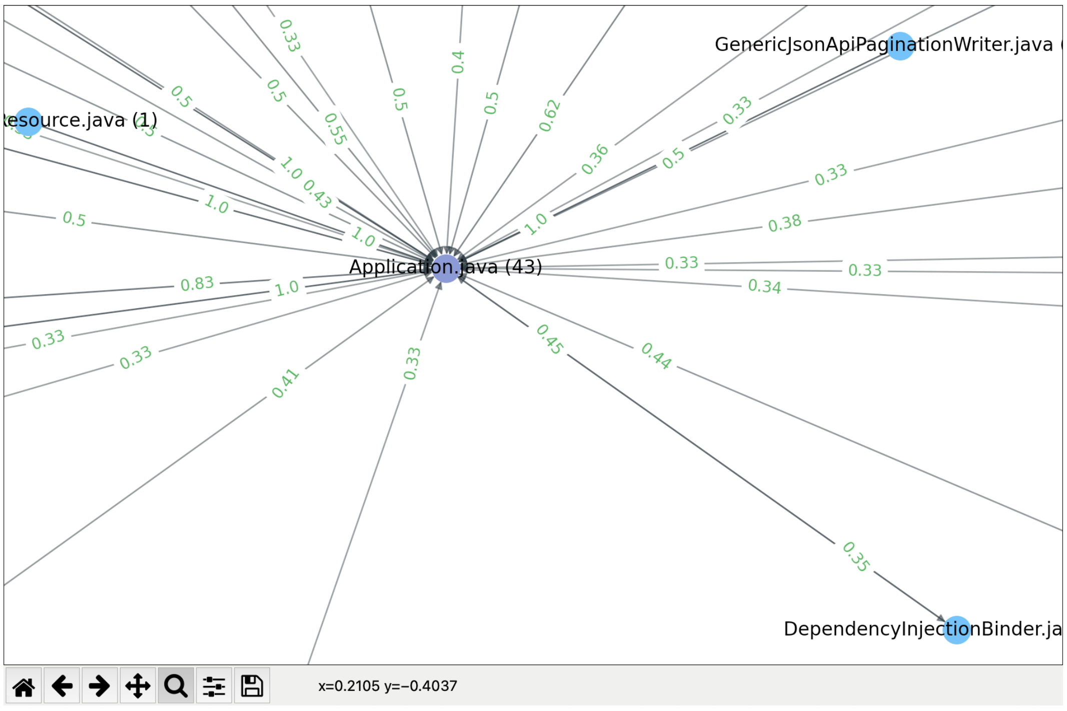
–minweight 0.33

This results in 33 nodes and 33 edges being rendered, which can be seen below:


SettingsDescriptorResource.java (3)
EndpointTest.java (5)
Main.java (8)
BatchRequestResource.java (18)
RoleResourceTest.java (2)
Role.java (2)
UserCredentials.java (8)
SettingJsonApiDeserializer.java (3)
UserServiceTest.java (3)
UserBatchRequest.java (10)
UserJsonApiDeserializer.java (11)
DatastoreFactory.java (1)
RoleResource.java (12)
AuthorizationService.java (9)
EntryPointResource.java (1)
RoleService.java (7)
User.java (1)
MalformedBatchRequestException.java (2)
GenericJsonApiPaginationWriter.java (1)
TestDatasourceFactory.java (8)
EntryPointResource.java (1)
Application.java (43)
BatchCreationService.java (3)
TestDatasourceFactory.java (6
TokenResource.java (32)
Token.java (6)
GenericJsonApiPaginationWriterTest.java (2)
SetupApplicationListener.java (30)
DependencyInjectionBinder.java (33)
BatchService.java (12)
TokenService.java (16)
IdGenerator.java (3)
BatchCreationServiceTest.java (17)
x=0.4012 y=−0.7376

It shows all the artifacts that are coupled with the inspected *Application.java* artifact. By using the directed raw counting algorithm, a digraph is created and when rendering a digraph, as it is the case here, each edge has two possible labels, each at one end.

The many edge labels in the center become more clear in the zoomed in view below:


Resource.java (1)
GenericJsonApiPaginationWriter.java
Application.java (43)
DependencyInjectionBinder.ja
x=0.2105 y=−0.4037

These edge labels are on the end of the edge that points toward the *Application.java* artifact and therefore show the weight of the coupling *from* any other artifact *to* the *Application.java* artifact. So it can be observed, that all the rendered artifacts are very dependent on the *Application.java*, but not vice versa. The only exception being the *DependencyInjectionBinder.java* file at the bottom right, on which the inspected *Application.java* artifact also depends on.

This small example made clear, how this tool can be used to specifically pick small parts of a complex system and analyze them in more detail.

4.3 Titan

This example will analyze a small repository to explain the idea of automating the usage of this tool. The repository to be analyzed is part of the Titan Control Center [Henning and Hasselbring 2021; Hasselbring et al. 2019], a big data analytics tool. It can be found at `https://git.industrial-devops.org/titan/DataFlowEngine/flowengine-go`. Manually executing the analysis with the same parameters as in the previous chapter only takes 5 seconds and an insignificant amount of memory. This makes it a good candidate for performing an ensemble run. An ensemble run will perform many analysis runs and slightly changing one or more parameters for each run. Then the different output data of the analysis is read and the differences in the parameter values is mapped against the changes in the output.

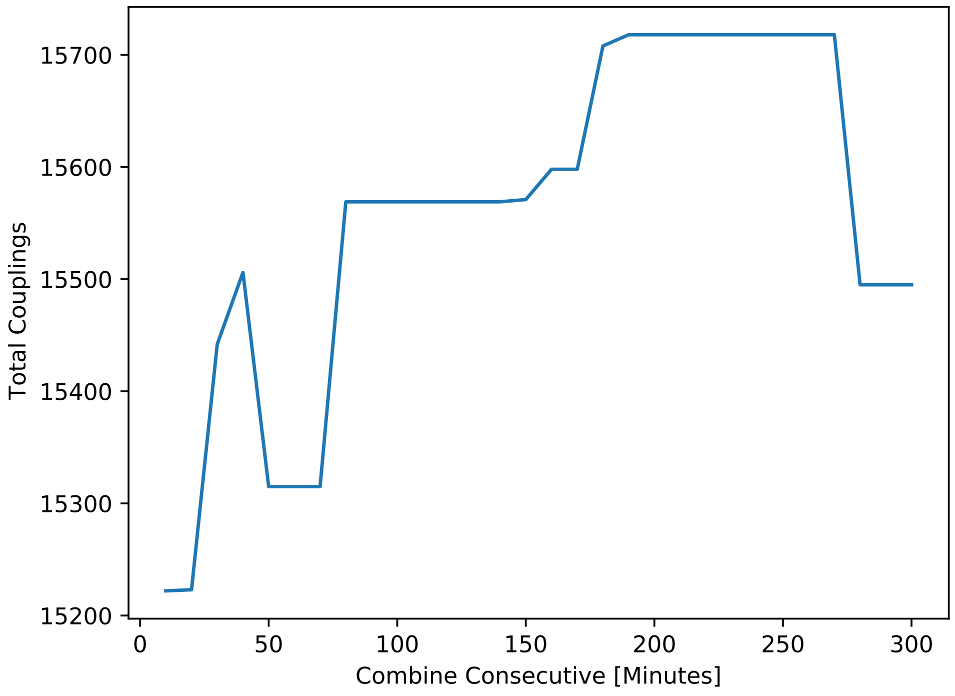
A simple python script doing such an ensemble run is provided below, but can also be found in the projects source code in the *ensemble_run.py* file:

```
import subprocess
import matplotlib.pyplot as plt
from tqdm import tqdm
from mining.util.formatters.JsonFormatter import JsonFormatter
import os.path

N = 29
start = 10
end = 300

step = (end - start) / N
parameters = list(map(lambda x: round(start + x * step), range(N + 1)))
formatter = JsonFormatter()

total_couplings = []

for parameter in tqdm(parameters, unit=' run', desc="Performaing analysis"):
  filename = f"ensemble-output/{parameter}.json"

  if not os.path.isfile(filename):
    subprocess.run([
        "python3",
        "mining",
        "analysis",
        "-r", "cache/flowengine-go",
        "-al", "urc",
        "-nm",
        "-nw",
        "-nl",
        "-df", "reuse",
        "-cc", f"{parameter}m",
        "-o", f"{filename}"
    ], capture_output=True)

  data = formatter.import_file({'input': filename})
  total_couplings.append(sum(edge['weight'] for edge in data['edges']))
```

```
37
38 plt.figure()
39 plt.xlabel("Combine Consecutive [Minutes]")
40 plt.ylabel("Total Couplings")
41 plt.plot(parameters, total_couplings)
42 plt.savefig("results.pdf")
43 plt.show()
```

The run can be configured with the variables in lines 7 to 9. **N** is the number of runs performend, starting at **start** and then testing each run in evenly distributed steps util it makes the last analysis using with the **stop** value. After some initialization of other variables, the parameters are iterated in line 18 and the name of the output file of this single run is set in line 19. If this file does not exist, meaning this specific run has not been performend in previous ensemble runs, the actual analysis tool is started from the command line. This can be seen beginning in line 22 with a set of static parameters, but also the dynamic parameters: *combineconsecutive* in line 32, based on the current parameter to be tested, and the already determined dynamic output file name in line 33. After that run is completed the *JsonFormatter* of the analysis tool is used to extract the graph data from the output file of the analysis in line 36 and this data is then evaluated in line 37. In this case, the simple output analysis consists of just adding up the total amount of couplings, that are detected across the entire project. In the last six lines 40 to 45, the evaluated data is then plotted and shown to the user and also exported as a vector based graphic inside a pdf file.

In this particular example, the performed ensemble run will run the analyzer for different values of the *–combineconsecutive* option, from **10m** to **300m** in 10m steps. This means, that a total of 29 analysis runs will be performend and then the data is plotted on a graph, which can be seen in in the following graph:



This data can then be further analysed and interpreted, but it is interesting to observe, that the number of couplings does not simply go up, when combining more and more commit. In fact the graph also shows, that at certain thresholds the number of couplings goes actually down. This special behavior and more use cases of automatic evaluation techniques are described in Chapter 6

4.4 Spring Framework

The last example is performend on the open source repository of the Java Spring Framework, which can be found at `https://github.com/spring-projects/spring-framework`. The goal of this example analysis is not to actually visualize any data or to explain the usage of the tool, as this is done in the previous examples. Rather this huge example repository is used to describe the performance limitations of the tool in its current state. At the time of this analysis the spring framework repository contained 22.353 commits, the most recent one being commit **610de3ae786812f332b71a7453a67afd39834a03**. It took 2 hours and 53 minutes to complete and peaked at 44 GB of memory usage. The first step of initializing the *CouplingCommit*s was completed in 43 minutes and only used 9 GB of memory. Up until this point the system was still fast an responsive and the commits where unpacked at a steady rate. But at the beginning of the second half of initializing the *ArtifactStore*, the tool reached 24 GB of memory usage, which, in addition to the around 6 GB of memory used by other parts of the system, forced the system to begin compressing the memory contents. This is a CPU intensive task and also slows down memory access times, thus the rate of commits processed gradually decreased with increasing memory compression. At around 80% progress of initializing the *ArtifactStore* memory usage of the tool was at around 36 GB and the memory could not be compressed any more, so the operating system began to page out the memory contents to the internal hard drive. This hugely increased memory access times and commits would only process at a rate below 10 commits per second. Also, the system regularly needed to swap out large chunks of the memory which haltet the analysis completely for several seconds. The initialization of the *ArtifactStore* was finally completed after 1 hour and 35 minutes. From this point on, memory usage will no longer increase, but the unresponsiveness of the system leads to the change coupling algorithm taking another 35 minutes to complete.

All in all, an analysis run at this scale is doable, but should be performed on hardware that provides enough memory for the analysis tool, without the need of swapping out parts of the memory. It also outlines the problem, that most of the time is needed in the initialization stage. When performing multiple successive analysis runs, the output of this stage is likely not to change between runs and therefore most of the time is wasted with redundant computations. This problem is further discussed in Chapter 6.

Chapter 5

Conclusion

In conclusion, the analysis tool developed during the course of this thesis does fulfill all perviously defined goal, with a few, low priority exceptions. The tool manages to deliver a feature rich experience with a high degree of user configurability. The two coupling algorithms introduced in Section 2.4.3 are also both fully implemented. It is able to analyze all reasonably sized repositories, but still has much room to be improved in terms of performance and features. But due to the exploratory nature of this thesis, this need for further improvement was predictable, thus the tool was designed from the beginning to be modular and extendable, which was achieved in almost all parts of its architecture.

The examples in Chapter 4 have shown, that the analyzer can already produce valuable results and that useful information can be extracted from this output data by either visualizing it with the build-in renderer or by automatically processing the data with external scripts or tools.

The only planned goals not achieved were, on the one hand the implementation of a more sophisticated coupling algorithm presented by Oliva and Gerosa [2015], that would utilize statistical evaluation methods to further improve the analysis results. On the other hand the analysis of the contents of the commit messages, to detect links to external issue tracking tools, therefore being able to detect related commits and couple them together.

Overall this explorative thesis provides a nice entry point for the field of change coupling analysis and opens the door for many different paths, that further scientific work could take.

Chapter 6

Future Work

Many ideas of improving and using the analysis tool have popped during the development phase and they are outlined in this chapter. Easily achievable ideas to further develop the tool, due to their low implementation effort, may be for example the addition of new coupling algorithms, preprocessing techniques or output file formatters. For instance the NetworkX package used in the rendering module already supports the conversion of a graph's data into different popular data formats, which then just need to be exported to the right file format. Or for example a new preprocessing algorithm, that does not simply filter out large commits, but count couplings detected in such large commits with less weight than in normal commits. Also this tool is suited nicely to not just do change coupling analysis, but to simply analyze and output useful statistical and analytical data about the repository itself, like for example developer participation, work flows or individual commit behaviors.

Some more advance tasks include the improvement the already provided algorithms, by consulting repository data, that is untouched at this point or implement performance improvements. For instance the way artifacts are reused, after they got deleted and added again, could be individually decided, by looking into the file contents and comparing the deleted artifact with the newly added one. Another direction for this would be to utilise the knowledge gained in other software system analysis tools, like static or dynamic coupling, and use the combined knowledge of multiple techniques to deliver better analysis results. An idea in terms of improving performance would be to cache the output of the initialization phase in some kind of file, so that subsequent analysis runs can save time. Also the issues that currently prevents parallelizing the application, so that multiple CPU cores can be utilized, can be studied in detail and solutions can be explored. To decrease the amount of memory needed, one idea would be to change the architecture of the entire tool in such a way, that not all commits are iterated over and over in each stage, which leads to all commits being in system memory at once, but to iterate over all commits exactly one time. Then all stages are executed on that current single commit, then proceeding to the next commit. This way only the last X iterated commits need to be kept in memory for algorithms which also look at neighboring commits, like a processing stage that combines commits. A more simple approch of reducing the tools footprint on memory usage is to outsource the *CouplingCommit*s and *ArtifactStore* to external databases, which would be way more efficient in terms of memory usage.

Ideas of actually using the tool to gain more insights contain, among many other possibilities, the analysis of problematic repository characteristics, like the amount of large commits or too small commits. This is coupled with the aforementioned mining of additional repository metadata, that is still untouched. Also, when adding the capability of supporting different version control systems, the quality of the output data can be compared between them. Also many legacy, long life repositories may switch their version control system at some point in time and the consequences of that transition can be analyzed, for instance in terms of before and after comparison of repository characteristics or an in depth look at the commits in close proximity to transition.

One last idea for future work is the use of ensemble runs, as outlined in Section 4.3, to automatically find the best configuration options for each repository. For example the goal for such an optimization could be the maximization of the number of detected couplings. The data collected in the ensemble run in the Titan example indicates, that there are some sweet spots in the input configuration that maximize the number of couplings, which is a good starting point, but other parameters and side effects of the use of particular combinations of parameters need to be tested too.

Bibliography

[Arisholm et al. 2004] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering* 30.8 (2004), pages 491–506. (Cited on page 2)

[Ball et al. 1997] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk ... (Oct. 1997). (Cited on page 1)

[Fittkau et al. 2017] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with explorviz. *Information and Software Technology* 87 (July 2017), pages 259–277. URL: http://eprints.uni-kiel.de/33464/. (Cited on page 45)

[Hasselbring et al. 2019] W. Hasselbring, S. Henning, and A. Möbius. A scalable architecture for power consumption monitoring in industrial production environments. In: *2019 IEEE International Conference on Fog Computing (ICFC)*. June 2019, pages 124–133. (Cited on page 48)

[Hasselbring et al. 2020] W. Hasselbring, A. Krause, and C. Zirkelbach. Explorviz: research on software visualization, comprehension and collaboration. *Software Impacts* 6 (2020), page 100034. URL: https://www.sciencedirect.com/science/article/pii/S2665963820300257. (Cited on pages 3, 45)

[Hasselbring and van Hoorn 2020] W. Hasselbring and A. van Hoorn. Kieker: a monitoring framework for software engineering research. *Software Impacts* 5 (2020), page 100019. URL: https://www.sciencedirect.com/science/article/pii/S2665963820300063. (Cited on page 3)

[Hasselbring and van Hoorn 2020] W. Hasselbring and A. van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impacts* 5 (June 2020). (Cited on page 41)

[Henning and Hasselbring 2021] S. Henning and W. Hasselbring. The titan control center for industrial devops analytics research. *Software Impacts* 7 (2021), page 100050. URL: https://www.sciencedirect.com/science/article/pii/S2665963820300415. (Cited on page 48)

[Oliva and Gerosa 2015] G. A. Oliva and M. A. Gerosa. "Chapter 11 - change coupling between software artifacts: learning from past changes". In: *The Art and Science of Analyzing Software Data*. Edited by C. Bird, T. Menzies, and T. Zimmermann. Boston: Morgan Kaufmann, 2015, pages 285–323. URL: https://www.sciencedirect.com/science/article/pii/B9780124115194000112. (Cited on pages 5, 46, 53)

[Robbes et al. 2008] R. Robbes, D. Pollet, and M. Lanza. Logical coupling based on fine-grained change information. In: *2008 15th Working Conference on Reverse Engineering*. 2008, pages 42–46. (Cited on page 3)

[Robbes 2008] R. Robbes. Of change and software. PhD thesis. Università della Svizzera italiana, 2008. (Cited on page 3)

[Schnoor and Hasselbring 2018] H. Schnoor and W. Hasselbring. Toward measuring software coupling via weighted dynamic metrics. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pages 342–343. URL: https://doi.org/10.1145/3183440.3195000. (Cited on page 2)

[Schnoor and Hasselbring 2020] H. Schnoor and W. Hasselbring. Comparing static and dynamic weighted software coupling metrics. *Computers* 9.2 (2020). URL: https://www.mdpi.com/2073-431X/9/2/24. (Cited on page 2)

[Spadini et al. 2018] D. Spadini, M. Aniche, and A. Bacchelli. Pydriller: python framework for mining software repositories. In: *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018. (Cited on page 10)

[Stevens et al. 1974] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal* 13.2 (1974), pages 115–139. (Cited on page 2)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22-25, 2012: ACM, Apr. 2012, pages 247–248. (Cited on page 41)

[Zimmermann and Weißgerber 2004] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In: *In MSR '04: Proceedings of the 1st International Workshop on Mining Software Repositories*. IEEE Computer Society, 2004, pages 2–6. (Cited on pages 16, 17)