



university of
 groningen

university college
 groningen

Connecting Discussions within Issue Tracking Systems

Bachelor Thesis

Alexander Fyodorov

Faculty of Science and Engineering
University of Groningen

20 July 2020

Supervisors:

dr. M.A.M. (Mohamed) Soliman

prof. dr. ir. P. (Paris) Avgeriou

Abstract

To prevent architectural decays, several tools that predict future issue reports by analyzing existing ones are developed. The results obtained by these tools are followed by opened issue cases, relevant issue discussions, and commits aimed at solving reported issues. All these issue resolution components, in turn, represent a nice knowledge source that can be reused by other projects to prevent similar issues. However, knowledge extraction from solutions can be a challenging task since it requires to traverse all sources relevant discussions refer to, such as forum discussions, issue-resolving commits and even other issues directly or indirectly related to each other.

The purpose of this project is to determine how discussions and knowledge sources related to a specific issue can be linked to each other to provide a coherent overview of approaches and decisions made to solve the issue. To test the effectiveness of the results, we will develop a tool that will link discussions and generate a human-readable report.

Contents

1	Introduction	1
1.1	Goal and research questions	2
2	Background	4
2.1	Issue tracking systems	4
2.2	Version control systems	5
3	State of the Art	7
3.1	Time series analysis of issues	8
3.2	Active Hotspots	8
3.3	Prevention of future issues	9
4	Analysis of references in issue tracking systems	10
4.1	Jira issue representation	10
4.2	Analysis process	13
4.2.1	Goals and challenges of statistical analysis	14
4.2.2	Approach	15
4.2.3	Python Jira	17
4.2.4	References parsing	18
4.3	Analysis results	20
4.3.1	URLs classification	20
4.3.2	References frequency	21
5	Linking discussions from issues	27
5.1	System architecture	27
5.1.1	Introduction	27
5.1.2	Process flow	28
5.1.3	Components description	30
5.1.4	Technical challenges	33
5.2	System evaluation	35
5.2.1	Valid input	35

<i>CONTENTS</i>	iii
5.2.2 Invalid input	37
6 Conclusion	40
6.1 Future work	41
7 Appendix A: ReadMe	44
8 Appendix B: Plots	47

1

Introduction

The development of software products is a complex task. There is a regularly occurring phenomenon of architectural decays and code smells in big and long-lived projects. As a project keeps growing, its architecture may be changed multiple times to meet new requirements, and in the worst cases, this leads to increased maintenance costs and the overall deteriorating quality of the project [1][2]. And the only way to prevent architecture degradation is to detect architectural flaws as soon as possible before the damage is done. For this purpose, several tools and techniques were developed, such as *SonarQube*, *Structure101*, *DV8*, and *Active Hotspots* [3].

If the scenario of architectural modifications can be predicted, then it is highly desirable to avoid them by considering them during building the initial architectural plan. Due to a big number of open-source projects available online, there is always a chance that knowledge about suitable architectural decisions can be extracted from resolved software issues from other projects. Moreover, solutions may be available on forums and Q&A (question and answer) websites, such as *StackOverflow*¹ and *Quora*², and developers may refer to them during issue discussions.

However, knowledge extraction is not an easy process. An issue may contain references to other issues and forums, and they, in turn, may also contain links to other knowledge sources. Moreover, some references are presented as unique identifiers and

¹<https://stackoverflow.com/>

²<https://www.quora.com/>

thus require some manual exploration. Therefore, it may be a complex, labor-intensive, and exhaustive task to traverse through all these sources until suitable knowledge is extracted, leading to the short attention span and probable unintentional information gaps.

This project is aimed at determining a generic approach to link discussions on software issues. We will examine which elements of issue reports can be used as connections to external sources. Those can be direct URLs or some unique identifiers that may refer to other issues within the same project. We will also examine how exactly these connections can combine multiple sources of knowledge into one. The results of the research will give developer communities a basis for project-specific, discussions linking approaches and a guideline for developing tools that link information about issues within some specific project and generate all-in-one human-friendly reports containing knowledge about consistent decisions made towards issues resolutions. To demonstrate the effectiveness of the approach selected, another goal of the project is to provide a working prototype of a tool that connects discussions of a project across multiple platforms, mainly bug tracking systems and code repositories, such as *Jira* and *GitHub*.

1.1 Goal and research questions

Existing issue analyzing methodologies provide efficient ways of predicting future bug reports and eliminating architectural decays. However, decisions made for a software issue resolution may be left buried inside issue trackers and are not intended to be easily found by search engines due to a lack of references to the issue from external resources. The ability to capture such information may provide other projects opportunities to avoid similar problems in the future or even to develop better solutions for already resolved ones and share them with the community.

There are two main goals of the project.

The first one is to describe how to build connections between issue trackers, source code repositories, and forums related to issues and how these connections can help with extracting useful information from discussions. Each issue is accompanied by discussions, primarily in a form of comments, and each discussion may include references. Such references may forward to forums and other issues with similar problem reports or relevant sources of information and even ready-made solutions. Moreover, most, if not all issues, contain some identifiers or keywords which can act as search terms to be used in target platforms. For example, all commits targeted at fixing some specific bug may contain references to an appropriate bug report in their commit messages.

This leads to the following research questions:

1. Which elements of issue reports can be defined as connections (connecting attributes) to other sources?
2. How can different types of connecting attributes be used to link multiple discussions into one source of knowledge?

These questions are expected to lead to the following contributions when answered:

1. Determine a generic approach to link discussions on software issues.
2. Design and develop a tool to support linking information about software issues within a certain project.

To test the effectiveness of the approach, a working prototype of the tool described is developed.

In case of success, the project can help IT companies and standalone developers make better decisions in architecture development by simplifying knowledge extraction.

2

Background

2.1 Issue tracking systems

When a software product is released, it is not guaranteed that it is bug-free. Unfortunately, developers and even software testing teams are unable to track every possible bug or issue. To simplify the process of tracking known issues and bugs, issue tracking systems are attached to the development process of a complex project.

Before defining what an issue tracking system, or issue tracker, is, we first have to explain what an issue tracking is in general. In 2006, Henderson provided a definition of issue tracking: "issue tracking, often called bug tracking (and sometimes request tracking), is the process of keeping track of your open development issues." [4] If a new issue is reported, its detailed description and any additional information are added to a specific computer software system that provides developers the way to access, update and discuss the information of every recorded issue. Such systems are called *Issue tracking systems*, also known as *Issue trackers* and *Bug tracking systems*. As defined by Black in 2002, "a bug tracking system is some program or application that allows the project team to report, manage, and analyze bug reports and bug trends." [4] An issue contains a description, comments and links to other issues and external sources, so it may represent an extensive source of knowledge about important design decisions.

Jira is a proprietary issue tracking system developed by Atlassian. According to the official website, around 170.000 customers from over 190 countries use *Jira* for issue

tracking and project management ¹. Being initially developed as a bug and issue tracker, Jira was eventually turned into a powerful work management tool, providing a friendly environment for teams practicing agile methodologies.

2.2 Version control systems

Any development process may be simultaneously accompanied by bad design decisions, e.g. a feature is implemented that eventually harms the quality of the project architecture or introduces a hard-to-fix bug. In this case, a developer may want to revert all the changes to some point before making a bad decision. For this purpose, he/she has to store the old state of a program for being able to revert to it at any time. However, manually saving the entire state can be time- and space-consuming, moreover, the developer has to keep a record of which files are changed and what is the reason for saving this state (in other words, the description of changes made).

Version control systems (VCSs) are aimed at solving these issues. According to Chacon and Straub (2014), Version control is a "system that records changes to a file or set of files over time so that you can recall specific versions later" [5]. It provides the user with a way to create snapshots of a project at any time and provide a detailed description of the latest changes made. The key feature of a VCS is that it can show the differences between any two snapshots, i.e. which files are changed and in which way, and the user can revert the project to the desired state in a fast and simple way.

The two key terms introduced by version control systems are *revision* and *commit*. Although these words are often used as if they are interchangeable, it is useful to understand the difference between them.

1. **Revision** describes the state of the product. Consider a project for which a VCS is used. The initial state is described as *Revision 1*. After some changes being made, the state of the project is changed and now corresponds to *Revision 2*.
2. **Commit** describes the *difference* between states/revisions. While a revision represents a state of a project, a commit specifies which files are changed and how. For example, when a project is created, *Commit 1* describes the difference between an empty project folder and all the files initially created. If the user adds 3 lines to a file *X*, then *Commit 2* points at the added lines in the file *X*.

Apache Subversion, abbreviated as *SVN*, is a software versioning and revision control system developed by the *Apache Software Foundation* in 2000 [6]. It is widely used for Apache's projects and is often used in conjunction with other version control systems,

¹<https://www.atlassian.com/company>

such as Git.

Git is a distributed VCS developed by Linus Torvalds in 2005 to accompany the development of the Linux kernel. According to Eclipse Community Survey 2014, Git is the most widely used code management tool; around one-third of all developers around the world use Git as their primary version control system. [7]

GitHub is the largest web-service for hosting software projects using Git. As of January 2020, more than 100 million repositories are hosted on GitHub [8] and more than 40 million users are registered². GitHub is heavily used alongside Apache Subversion as the main source code hosting service for Apache projects. GitHub may contain pull requests and commits for issues we are interested in extracting knowledge about. A *pull request* is a request to check and review changes made to a project to merge them into the main code base.

Sometimes, an important information has to be sent to multiple recipients who can be developers or testers. For this purpose, *mailing lists* are used. A mailing list represents a list of email addresses to which the same information is sent.³

²<https://github.com/search?q=type:user&type=Users>

³<http://www.list.org/mailman-member/node5.html>

3

State of the Art

In this section, we will explain basic concepts by referring to the relevant literature and support the goal of the project. It is important to understand the process of issue analysis in order to build connections between software issues discussions. All the literature can be found at *IEEE Xplore*.¹

The process of designing a software architecture involves making a lot of design decisions, each of which affects properties and the functionality of the software product in varying degrees. That is why architecture decay is one of the most acute problems in software development. As a software project grows, new architectural decisions are added and existing ones are modified or removed [1]. In other words, the architecture keeps changing constantly during the lifetime of the project. This leads to an increasing number of architectural smells making the system hard to maintain and increasing costs and efforts of software maintenance [9] [2].

Correct predictions of bug reports and requests for system enhancements may dramatically reduce the amount of upcoming architectural decays and help large teams to allocate staff to different tasks more efficiently [10]. There are several well-described methodologies for detecting and predicting architectural smells and debts, and none of them is universal. However, they all have a common characteristic: the foundation of knowledge for each of them is taken from the analysis of issue reports, both resolved and opened, and enhancement requests, as well as connections between them and their

¹<https://ieeexplore.ieee.org/Xplore/home.jsp>

atomic components, e.g. source files [1] [9] [3].

3.1 Time series analysis of issues

By analyzing issue reports in 832 open-source and proprietary projects, Krishna et al. were able to build time series models on issues, which could be used to predict future bugs and enhancements [10]. Temporal trends in the data mined were modeled using *Autoregressive Integrated Moving Average* (ARIMA) by applying a rolling-window analysis, and moderately strong correlations between bugs, issues, and enhancements were found. Furthermore, ARIMA showed itself being accurate for predicting new bugs and issues, with a very low level of errors and the variance in errors.

Forecasts made by ARIMA originate from the analysis of past temporal data or issue reports. Past temporal data illustrates different temporal trends in issues, bugs and architectural improvements. The results demonstrated by Krishna et al. show that predictions based on past temporal trends are statistically comparable to the ones based on issue reports only.

All things considered, ARIMA shows clear connections between issues, bugs and enhancements and proves that these connections can be used to predict future trends and upcoming issue reports.

3.2 Active Hotspots

One of the most recent methodologies to reveal architecture problems and predict them is *Active Hotspot* - an issue-oriented model that tracks changes in source files and their relations within the scope of each issue [3]. To be more precise, it determines files whose modifications address multiple issues, finds architectural and semantic relations between them through four propagation patterns, and forms groups of files called *active hotspots*.

The propagation patterns were deduced after it was found that 96% of bug fixes followed four recurring patterns [3]:

1. **Dissemination:** if a method/field is modified in one file, then all files that use this method/field are adjusted accordingly.
2. **Concentration:** changes in multiple classes are reflected in a single class which depends on them.
3. **Domino:** a change performed in one file leads to a cascade of consequent changes.

4. **ScatterShot**: an injection of similar patch logic into multiple files.

The priority for the analysis goes to *dominant* (affecting 5 or more files) and *persistent* (long-lasting) hotspots. In comparison with other architecture smell analysis tools, the number of files with smells captured by hotspots does not correlate with the size of the project since it is related only to the intensity of architectural relations and issue interactions between affected files. In other words, hotspots can show high precision and a low recall of finding bug-prone and change-prone files due to reporting a smaller number of files in a constantly evolving project, and this can be achieved by focusing on dominant and long-lasting hotspots the number of which does not intend to be changed.

3.3 Prevention of future issues

Existing methodologies for issues analysis represent a nice source of knowledge about design decisions and may help to avoid architectural inefficiencies for other projects. However, discussions that lead to appropriate decisions may be hidden deep inside issue reports and bug trackers and are not that easy to extract due to a high fragmentation level, for example, the same issue can be discussed on multiple platforms, such as source code hosting services like *GitHub*, bug trackers like *Jira* or even forums dedicated for software projects. This implies that in order to build an overall picture of some specific decision, it may be required to manually traverse all these platforms, which may be time-consuming and easy to miss important information.

Shahbazian et al. developed a tool *RecovAr* to recover architectural decisions from the project's historical artifacts, such as resolved issues and commits that address them [11]. The tool shows a high level of recall and precision, however, some discussions can still be lost, moreover, the recovered ones may not be easy to read or understand. The reason for that is that the tool builds connections only between issues and relevant code changes, while a lot of information related to decisions may be found on forums and bug trackers.

It is expected to extract more useful information if new types of connections are found. The best candidates are URLs to forums, bug trackers or even old issues and commits which led to some design decisions.

4

Analysis of references in issue tracking systems

4.1 Jira issue representation

The first step of the research is to select a suitable Jira-powered system with some projects with various development processes. And the one that fulfills our needs is ASF JIRA - a publicly available issue-tracking system related to open-source Apache projects.

Before the beginning of the analysis, it is important to understand how issues of projects hosted on ASF JIRA are structured. For illustrative purposes, the issue *PDFBOX-3017* related to the *Apache PDFBox* project is taken.

1. When the issue is opened (Figure 4.1), the first thing that catches an eye is the *Description* section. Alongside the description of the problem, it contains a number of links to external resources and references to other issues.



PDFBox / PDFBOX-3017
Improve document signing

Details

Type:	Improvement	Status:	OPEN
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	2.0.0, 3.0.0 PDFBox	Fix Version/s:	3.0.0 PDFBox
Component/s:	AcroForm, Signing		
Labels:	None		

Description

Improve signing code:

- incremental save only works for signatures and doesn't respect certificates such as Adobe Extended Usage Rights
- [prepareNonVisualSignature](#) clears the AcroForm DR `acroForm.setDefaultResources(null)` which is not good if there are other form fields
- [visual/nonVisualSignature](#) should move into the `interactive.forms` package and be handled within the signature field
- [verify signature \(to have tests that go full circle\)](#) done June 2016
- document or refactor / rewrite visible labyrinthine signature code
- why is it not possible to pass only the signatureField to `addSignature`, instead having to create a COSDocument with a page and annotations that has the signature field, and that must be searched for in `prepareVisibleSignature()`?
- [support rotated pages \(see <https://stackoverflow.com/questions/34012293/pdfbox-sign-landscape-file-error/34359956#34359956>\)](#) done in [PDFBOX-3671](#)
- make sure that signed PDF/A files are still PDF/A (see <http://www.pdfa.org/wp-content/uploads/2011/08/tn0006-digital-signatures-in-pdf-a-1-2008-03-14.pdf>); /ID possibly not OK; /Annots is possibly required (Tilman Hausherr removed this for invisible signatures); test signed files with PDF-Tools and with preflight tested, they are OK with PDF-Tools and preflight
- test whether "bad" signatures are detected by preflight (search in old issues)
- [PDFBOX-3363](#) - why is the stream cached in a file? Should it be done in memory? done on July 15, 2016
- remove `setVisualSignature(PDVisibleSigProperties visSignatureProperties)` from `SignatureOptions.java`, all it does is to call `visSignatureProperties.getVisibleSignature()` which returns an `InputStream`, and this is already available
- `checkSignatureField` violates the "do one thing" rule
- decide whether the whole certificate chain should be passed in the sample code, instead of only the first one yes the whole chain is stored
- check certificate chain, revocation lists, etc, only if needed by users, [code here](#)
- deprecate / remove all `PDVisibleSignDesigner` constructors except those with a `PDDocument` object, to avoid a file being opened twice
- ... your ideas...

Figure 4.1: PDFBOX-3017, highlighted references

Here we can distinguish three different types of references, where:

- is a reference to the StackOverflow forum.
- stands for another issue.
- refers to a PDF document.

2. The next two sections that deserve special attention are *Attachments* and *Issue Links* (4.2).

The screenshot displays the 'PDFBOX-3017' issue page with three main sections:

- Attachments:** A table listing five files:

pdfa_signed_invisible.pdf	35 kB	02/Apr/16 15:29
PDFBOX-3017_certificate_chain_Screenshot.png	104 kB	17/Jul/17 09:42
PDFBOX-3017_certificate_chain.diff	2 kB	17/Jul/17 02:40
QV_RCA1_RCA3_CPCPS_V4_11.pdf	994 kB	16/Oct/18 15:54
S052757037-Signed3-OCSP-with-KeyHash.pdf	33 kB	02/Dec/18 09:50
- Issue Links:**
 - is depended upon by:** PDFBOX-3498 Visible Signature N2 layer / Support signature with text (CLOSED)
 - links to:**
 - GitHub Pull Request #39
 - Stackoverflow: PDFBox 1.8.10: Fill and Sign Document, Filling again fails
 - Unhelpful error message when using X509CRL.verify(java.security.PublicKey, java.security.Provider) in project int...
- Sub-Tasks:** A list of five tasks:
 - Remove classic signing and keep external signing only (OPEN, Unassigned)
 - signatureField.setValue() not implemented (CLOSED, Maruan Sahyoun)
 - reference existing signature field when signing (CLOSED, Tilman Hausherr)
 - Adjust signature field for rotated pages (CLOSED, Tilman Hausherr)
 - Explicit support for certification signatures (CLOSED, Tilman Hausherr)

Figure 4.2: PDFBOX-3017, issue links

Attachments may contain essential information about the issue that is not included as references.

Issue links contain two types of references:

- Other issue links.** These are the links to other issues, like a dependency of one issue from another or duplicating issues.
- Remote link.** These links point to external resources. One link leads to a GitHub pull request, another one - to a StackOverflow discussion, and the last one - to a GitHub issue of another project, *Animal Sniffer*.

3. The last important section which may contain a number of references is the *Comments* section (4.3):

Activity

All Comments Work Log History Activity Transitions ↑

▼ ● Tilman Hausherr added a comment - 09/Oct/15 19:10

I don't like these "chained" calls in the visible signature stuff. I didn't like these calls in the 80ies, when I first saw them. Isn't this against the "law of demeter"?

▼ ● Michael Klink added a comment - 02/Nov/15 13:09 - edited

`prepareNonVisualSignature` clears the AcroForm **DR** (`acroForm.setDefaultResources(null)`) which is not good if there are other form fields

The pendant for this issue in the context of visual signatures has been observed in real life, cf. the stackoverflow posting [PDFBox 1.8.10: Fill and Sign Document, Filling again fails](#).

▼ ● ASF subversion and git services added a comment - 02/Nov/15 15:04

Commit 1712037 from Maruan Sahyoun in branch 'pdfbox/trunk'
[<https://svn.apache.org/r1712037>]

~~PDFBOX-2846~~, PDFBOX-3017: don't drop AcroForm /DR for invisible signature

▼ ● ASF subversion and git services added a comment - 02/Nov/15 15:15

Commit 1712040 from Maruan Sahyoun in branch 'pdfbox/branches/1.8'
[<https://svn.apache.org/r1712040>]

~~PDFBOX-2846~~, PDFBOX-3017: don't drop AcroForm /DR for invisible signature

▼ ● Maruan Sahyoun added a comment - 02/Nov/15 15:18

Michael Klink thank you for the 'reminder' - I've changed that so the /DR is kept

Figure 4.3: PDFBOX-3017, comments

It may contain any type of references described above, and moreover, some projects, including *PDFBox*, have an activity-reporting bot which provides automatically generated summaries of what has been done. For example, the bot shown in the selected issue reports revision ID, name of the author, and affected issues.

4.2 Analysis process

To determine how can issues and discussions be linked to each other, it is important to manually analyze resolved and unresolved issues for a bunch of projects. The issue tracking system selected for the analysis is *Jira* and source code repositories for the projects are taken from *GitHub*. Since some discussions may describe solutions by providing references to external forums, *StackOverflow* is selected as one of the most popular question and answer websites used by over 12 million programmers around the world [12].

To define the most common characteristics for connecting attributes, a quantitative analysis of software issues must be applied. The list of connecting attributes includes, but is expected not to be limited to, entries of the following types:

1. **Commit ID.** A resolved issue always includes one or more commits to the code base. By capturing commit IDs, it is possible to link them to relevant issues as some issues also include lists of commit IDs addressed at them.
2. **Issue key.** Each issue has a unique identifier that distinguishes it from the others. A commit message may include some meta-information such as an issue key so that a reviewer knows exactly which issue is addressed by the commit. An issue key may be changed in the future, for example, if a user creates an issue with a key *ABC-123*, where *ABC* is the name of a project, and later the project is renamed to *DEF*, then the issue key is updated to *DEF-123*
3. **URL.** This can be a link to an external website with a ready-made solution, another issue, a human-readable overview of some commit, etc.

Two main connecting attributes are **Commit IDs** and **Issue keys**. Thus, the basis for building connections inside a project can be formed from two approaches (separate or combined):

1. Extract commits IDs from the code base and search for these IDs or URLs that lead to these commits inside a list of issues.
2. Extract issues keys from the issue tracker and search for commits that address relevant issues.

4.2.1 Goals and challenges of statistical analysis

To understand how a project evolves, we need to collect the statistics of different types of issue references that appear in issue discussions. Alongside other issues and commits/revisions, an issue may contain several URLs that are expected to be categorized. For example, there can be a URL to a pull request on GitHub or a discussion forum.

One of the goals of statistics collection is to describe different categories of URLs mentioned in discussions. Once a new category is determined and a key characteristic that helps to distinguish a URL from the others is found, then all URLs belonging to that category are filtered out, reducing the size of the pool of uncategorized URLs.

Another goal is to determine the "breakpoints" - checkpoints indicating periods when the project development process is affected by some events, e.g. the development team switches to new technology or an issue resolution directly or indirectly results in some

other issues reported. If the frequency of such events occurred is high at some point, then it may indicate that the corresponding period of the development process contains important design decisions.

The first obstacle to overcome is the heterogeneity of references in issues, e.g. one issue may contain no references at all and another one includes a bunch of URLs and referred commits. Because the target projects are massive (e.g. PDFBox has 4915 issues and Cassandra - 15945 issues by July 15th, 2020), it may be relatively hard to determine the breakpoints. Thus, it is decided to combine sequential issues into blocks of the same size, i.e. 100 issues per block, and to collect statistics per block and not per issue.

The second issue that may arise is the lack of additional information about references that may help us to classify them. For example, a project may use mailing lists to share information among developers, and it is nice if the project description contains somewhere the list of mailing list services that are used. However, if the necessary information is unavailable, an additional manual analysis of references has to be performed.

4.2.2 Approach

Before the statistics can be analyzed, we first need to perform several steps to retrieve the data in the form suitable for the analysis. Statistics collection requires a small number of details about each issue.

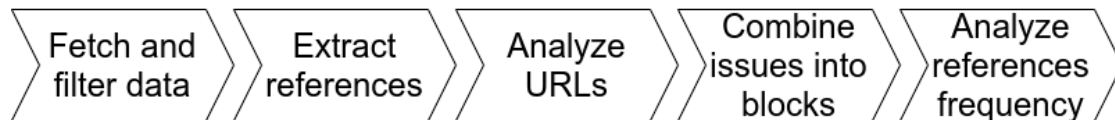


Figure 4.4: Analysis of references flow

1. Before we start the analysis of issues, we firstly have to fetch them using Jira API. Initially, each Jira issue contains a great amount of data that is not required for the analysis, such as technical details of the issue for internal use. This data has to be filtered out, leaving out the one containing references, such as issue description or comments. An example of the data that may contain references (highlighted in green) is taken from the issue *PDFBOX-4815*:

```

241     "watches": {
242         "self": "https://issues.apache.org/jira/rest/api/2/issue/PDFBOX-4815/watchers",
243         "watchCount": 2,
244         "isWatching": false
245     },
246     "created": "2020-04-17T19:00:58.000+0000",
247     "updated": "2020-04-19T18:21:59.000+0000",
248     "timeoriginalestimate": null,
249     "description": "Hello, I have an issue with a PDF with gray background in some parts.
250     "customfield_10010": null,
251     "customfield_12314127": null,
252     "customfield_12314126": null,
253     "customfield_12314125": null,
254     "customfield_12314124": null,
255     "customfield_12312340": null,
256     "customfield_12314123": null,
257     "customfield_12312341": null,
258     "customfield_12312220": null,
259     "customfield_12314122": null,
260     "customfield_12314121": null,
261     "customfield_12314120": null,
262     "customfield_12314129": null,
263     "customfield_12314128": null,
264     "summary": "Gray background preview becomes black squares",

```

Figure 4.5: PDFBOX-4815, sections that may contain references

2. For each issue, all issue keys, revision IDs, and URLs are extracted and stored in separate sets using pattern matching via regular expressions. The same reference may occur multiple times across the issue, while the connectivity for each issue is measured by *unique* references, so sets allow us to ensure that the same reference will not occur more than once (more about them in Subsection 4.2.4). All the parsed data is persisted in a corresponding file for each issue to simplify the manual analysis of references and to prevent repeated data retrieval.
3. As references are extracted, we have to perform a manual analysis of URLs, i.e. to check how they can be categorized based on the content of their addresses.
4. Due to the heterogeneity of references in issues as explained above, all issues are combined into blocks of 100 issues each. For each block, the total amounts of issue keys, revision IDs, and URLs are calculated.
5. For each type of reference, a separate plot is made. Each plot describes how the usage of each type of reference changes while the project evolves. Besides, a

separate plot describing the frequency of references combined is made.

4.2.3 Python Jira

With Jira API, it is possible to retrieve any required information about issues and even projects. This information is stored in JSON format, so a suitable tool has to be chosen to parse the information.

Python is decided to be an appropriate choice due to several reasons:

1. It allows rapid prototyping, which helps the user to focus on the main task and start working on the retrieved data as soon as possible.
2. Working with string data is heavily simplified, taking a relatively small amount of code required to perform some tasks in comparison to languages like *Java* or *C++*.
3. A JSON string can be converted to Python dictionaries and lists and vice versa with a single statement from the standard library.
4. Python's *matplotlib*¹ provides us with handy functions to build plots.

To communicate with Jira API, it was decided to use a Python API wrapper to avoid plain REST API requests. One of the options is to use the official solution *Atlassian Python API wrapper*², however, its main purpose is to simplify development processes using Jira, and thus, it requires extra configuration. So the choice fell on *Jira Python* library provided by the *PyContribs* project³.

When making a GET request via Jira API, we have to make sure that all sections from an issue description containing any types of references, as illustrated on the screenshots in Section 4.1:

1. **Description:** contains the explanation of the reported issue.
2. **Attachments:** contains attached files that complement the description. They may contain, for example, PDF documents that are decided for some reason to be not included as remote links. One such reason can be the inaccessibility of the document by the original URL.
3. **Issue Links:** contains links to other issues (e.g. a duplicate or a derivative of another issue) and links to external resources, including forums and URLs to GitHub issues and pull requests.

¹<https://matplotlib.org/>

²<https://github.com/atlassian-api/atlassian-python-api>

³<https://jira.readthedocs.io/en/master/>

4. **Comments:** contains discussions related to solving the issue.

By making a GET request with no fields specified, it is found that the response does not include all the necessary data: only *description* and *other issue links* are included by default. Thus, to retrieve the rest of the fields, they have to be specified explicitly. It is also observed that if any field is included in the request, all other fields are missing from the response.

To retrieve all necessary data, the request should contain the following set of fields:

"description,attachment,issuelinks,comment"

It is worth mentioning that remote links (i.e. links to external resources) cannot be obtained with other fields, so an extra request is required. This is important for the following explanation of the issues retrieval mechanism.

Let's take the *PDFBox* project as an example. By July 6th, 2020, there are 4903 opened and closed issues. Jira API allows the user to retrieve multiple issues with a single request, so it was decided to fetch the issues in blocks of 100 issues each. This results in $\lceil 4903/100 \rceil = 50$ standalone requests. However, as explained above, remote links are not fetched with other fields, thus, for each issue, there should be an extra request. The total amount of requests is $50 + 4903 = 4953$, which is a relatively big number. This may cause additional problems, e.g. the server may block all requests from an IP address if an intense activity is detected.

4.2.4 References parsing

After all the necessary data is fetched, the text has to be analyzed to extract all references and categorize them. Since they always follow some pattern, it is reasonable to use regular expressions to filter them out.

1. Other issues

As it can be seen on Figures [4.1](#), [4.2](#), and [4.3](#), all issues are referenced by issue keys.

An issue key is a unique identifier of an issue, and it is written in the following form:

<project_name>-<numerical_ID>

By using this pattern, we can extract all issue keys from a text. For example, the issue described in the figures has the key *PDFBOX-3017*.

2. URLs

URL detection is a challenging task. Usually, it is enough to check a URL for being in compliance with the standard *RFC 3986*, which stands for *Uniform Resource Identifier* (URI). However, this standard is limited to a subset of the ASCII character set, and there is no guarantee that issues descriptions and comments section will not contain characters of an extended character set.

For this purpose, it is decided to find a regular expression that matches the standard *RFC 3987*, which stands for *Internationalized Resource Identifier* (IRI). This standard bypasses the restrictions introduced by *RFC 3986* by allowing characters from the *Universal Character Set* (Unicode).

Mathias Bynens, a developer advocate on the V8 JavaScript engine team, provided an overview of some regular expressions for URL validation⁴. He introduced two sets of URLs to test: the ones that are expected to be valid and the ones that should fail.

The solution developed by Diego Perini⁵ showed outstanding results: it detected 36 out of 37 "valid" URLs and rejected all 39 "invalid" ones. It was decided to use his regular expression to detect URLs in plain text.

3. Commits/Revisions

It is common for Apache projects to use a combination of SVN and GitHub in their development process. Each commit made on GitHub is reflected on the corresponding revision in SVN. To extract them, it is important to understand how are they represented in issues discussions. A pre-analysis of a number of projects including *PDFBox*, *Cassandra*, and *HDFS* allows us to determine the patterns of commits and revisions references that can be extracted from discussions.

- (a) **Commits** are represented in the form of their 40-digit hexadecimal hash values or their shortened 7-digit variants.
- (b) **Revision IDs** references can be met in various forms. Consider, for example, a revision with ID 1234567. It is observed that the revision can be mentioned by developers as:
 - 1) **r1234567** - the most common form and the official representation of a revision.

⁴<https://mathiasbynens.be/demo/url-regex>

⁵<https://gist.github.com/dperini/729294>

- 2) **rev. 1234567**
- 3) **Rev. 1234567**
- 4) **Revision 1234567**
- 5) **revision 1234567**

One of the key reasons why revisions and commits should be separated from each other is because it is observed that while commits describe the difference between two states, revisions are linked to changes in a single file. If, for example, a commit modifies 5 files, then there will be 5 separate revisions generated. This fact allows us to determine which commits introduce massive changes in a project: the more files are affected by a commit, the more changes are introduced.

4.3 Analysis results

To recap the goals of statistics collection (Subsection [4.2.1](#)), the analysis of results is aimed at two issues:

1. Divide URLs into categories to define the nature of external sources laying behind them.
2. Determine the periods of the project development when there is the highest chance of important design decisions made.

4.3.1 URLs classification

The manual analysis of extracted URLs helped us to discover two categories of URLs that can be easily distinguished:

1. **PDF documentation:** Often, developers may refer to the documentation of external projects to gather useful ideas for their projects or to retrieve detailed guidance on using a tool or a piece of hardware. URLs pointing at PDF documents can be distinguished by their postfix: when a URL is linked to some file, it ends with the extension of the file. Thus, URLs leading to PDF documents end with the ".pdf" extension.

It has been observed that the usage of PDF documentation by developers during discussions is relatively high to be defined as a separate category.

2. **Mailing lists:** mailing lists are often used to coordinate the development of the software. Initially, it was thought that it was enough to check whether a URL contains the substring "mail" to tell whether it is a mailing list URL. However, the

presence of this substring does not guarantee that a URL is a mailing list. For example, the URL `https://pdfbox.apache.org/maillinglists.html` has been found in the issue *PDFBOX-1822*, which is expected to provide a list of mailing lists used, but not to describe the content of mailing lists.

Thus, it is decided to manually specify mailing list keys, i.e. substrings that have the highest chance to occur in mailing list URLs. For the project *PDF-Box*, it is observed that initially, most mailing list URLs belonged to `http://mail-archives.apache.org`, however, at some point, the project switched to `https://markmail.org` as the main mailing list service. Thus, the list of mailing list keys contains the entries "mail-archive" and "markmail". We can also extend this list with mailing-list archivers, such as "pipermail" or "hyperkitty"⁶.

3. **Archive files:** the manual analysis of the earliest issues shows some URLs pointing to archive files, in other words, the URLs are ending with ".tar", ".zip" or another popular archive formats.

In order to separate URLs to archive files from the others, it is decided to check whether a URL ends with one of the following formats: ".zip", ".tar", ".rar", ".iso", ".gz", ".rz", ".lz", ".7z". If there is evidence of some other format being actively used, the appropriate extension can be added to the analysis tool.

4. **Other URLs:** any URL that does not belong to the categories specified above goes to "Other URLs". These URLs are expected to be analyzed more deeply in the future to categorize them or are not expected to act as "connections" at all (for example, URLs to ".txt" files).

4.3.2 References frequency

By analyzing the plots of frequencies of references per block of issues, we can select the most "interesting" periods of a project evolution to extract connections from. For the demonstration purposes, the projects "PDFBox", "Derby", and "Cassandra" are selected.

At the moment, plots are generated for the frequencies of:

1. Revisions
2. Other Issues
3. Mailing Lists

⁶<https://hyperkitty.readthedocs.io/en/latest/>

4. PDF documents
5. Archives
6. Other URLs
7. Total references (union of all other types of references)

We will have a look at plots generated for *Revisions* and *Other issues*. It is decided to not analyze all plots since their goal is to provide an overall idea of how the frequency of references can show which periods of the project development contain the biggest number of external references. The rest of the plots can be found in [Appendix B](#).

1. Revisions

By analyzing the frequency of revisions made for each project, we can distinguish the periods of the project development where the biggest changes to the source code are made.

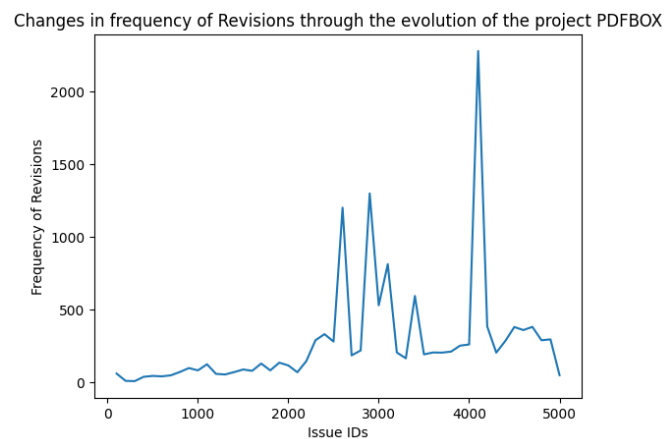


Figure 4.6: Frequency of revision references throughout the PDFBox project evolution

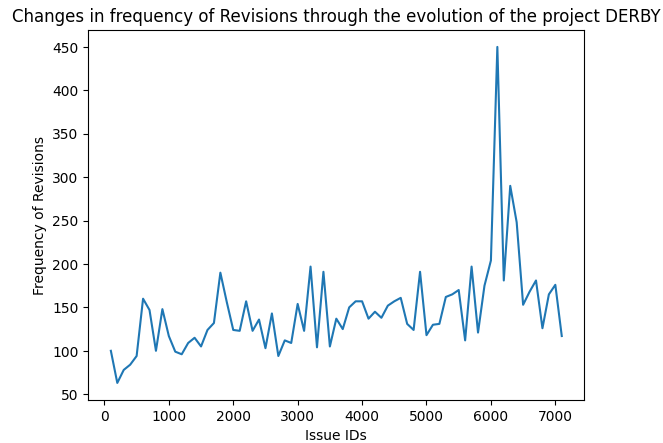


Figure 4.7: Frequency of revision references throughout the Derby project evolution

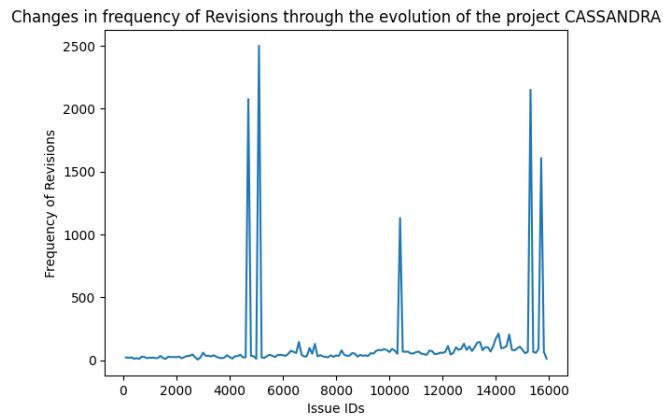
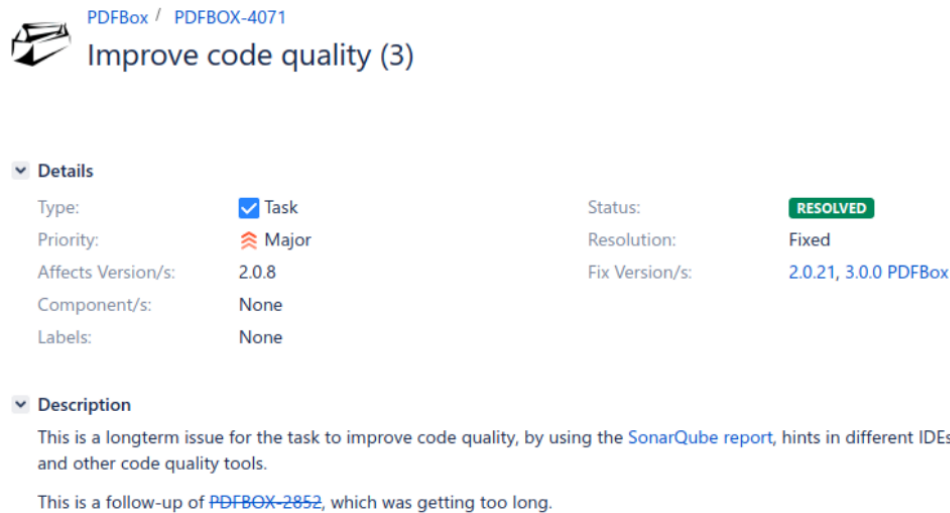


Figure 4.8: Frequency of revision references throughout the Cassandra project evolution

As can be seen on the plot, *PDFBox* received a massive contribution between issues 2500 and 3500, and the biggest changes made to the project were done approximately between issues 4000 and 4200. For *Derby*, there are on average 1-2 revisions per issue most of the time, with a surge of contribution after issue 6000. *Cassandra* project shows a relatively low activity throughout its entire history, with the only contribution burst between issues 14000 and 14500.

By selecting the specified ranges of issues, the user can extract the information about the biggest changes in these projects. We can show how this information

can be useful by taking the peak between issues 4000 and 4200 on the Plot 4.6 as an example. The analysis of the number of revisions per issue shows that the issue *PDFBOX-4071* contains 2002 revisions, and it is aimed at improving the code quality of the project:



PDFBox / PDFBOX-4071

Improve code quality (3)

Details

Type:	<input checked="" type="checkbox"/> Task	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	2.0.8	Fix Version/s:	2.0.21, 3.0.0 PDFBox
Component/s:	None		
Labels:	None		

Description

This is a longterm issue for the task to improve code quality, by using the [SonarQube report](#), hints in different IDEs, the FindBugs tool and other code quality tools.

This is a follow-up of [PDFBOX-2852](#), which was getting too long.

Figure 4.9: PDFBOX-4071, issue description

This issue illustrates important design decisions and the discussions connected to it may provide a nice overview of architectural flaws of the project.

2. Other issues

The relationship between different issues demonstrates that a solution development for an issue may be greatly influenced by other issues, and thus, the knowledge extracted from an issue can be expanded with the knowledge from the connected issues.

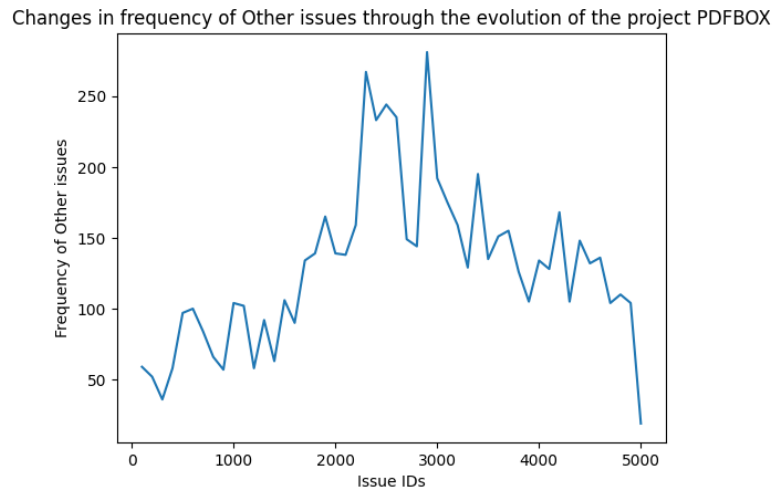


Figure 4.10: Frequency of other issues references throughout the PDFBox project evolution

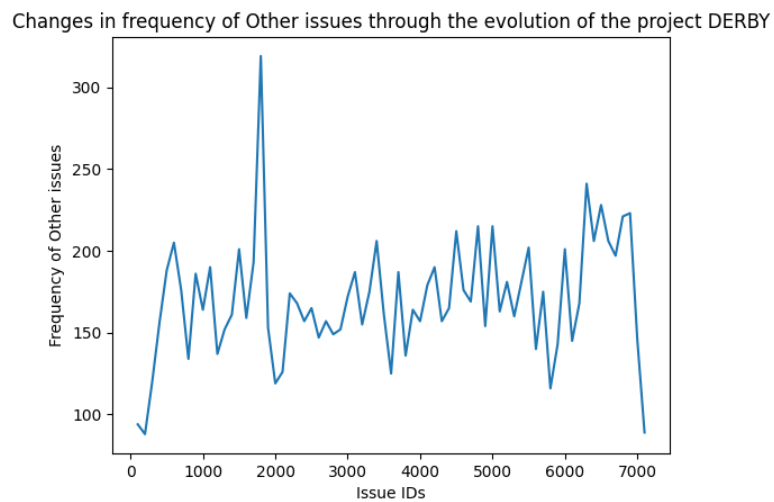


Figure 4.11: Frequency of other issues references throughout the Derby project evolution

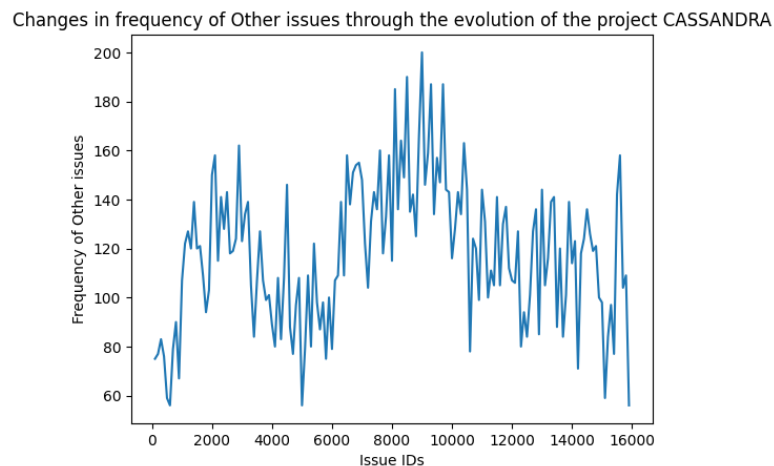


Figure 4.12: Frequency of other issues references throughout the Cassandra project evolution

It can be seen that for *PDFBox*, the region between issues 1500 and 3500 contains the biggest amount of cross-references between issues, so this region arouses heightened interest. For *Derby*, there is a peak of other issues references approximately between issues 1500 and 2000. Finally, *Cassandra* shows a slight increase in the middle of the project development process between issues 8000 and 10000.

It is observed that some *Cassandra* issues within the specified period are referencing a relatively big number of other issues, for example, *Cassandra-9302* connects to 18 other issues and *Cassandra-9318* references 20 other issues. *Cassandra-9318* is found to be a unification of other issues, i.e. it combines the problems other issues addressing, so it may be used as a good source of knowledge since it allows the user to bypass the analysis of other issues referenced by it. *Cassandra-9302* seems to be in conflict with a number of other issues (there are several duplicates and blocked issues) which has been successfully resolved, so this issue can teach developers how to deal with overlapping issues.

These plots provide a nice overview of which stages of the project development process may contain the most knowledge of the project evolution. By extending the list of known types of references, it is possible to operate on the data more flexibly and omit unnecessary data, leaving only the most interesting sources of knowledge.

5

Linking discussions from issues

5.1 System architecture

5.1.1 Introduction

The goal of the project is to provide a prototype of an easy-to-use tool for generating human-friendly reports on Jira issues analysis. For each specified issue, the user can retrieve an overview of issue discussions, additional details that the one may consider being useful, an overview of other issues that may be referenced in discussions, and the content of GitHub pull requests and commits that refer to the issue. This project makes use of the following connecting attributes:

1. **Other issues**, references to which may occur during discussions
2. **Commits** and **Pull requests** that address the target issue

By default, for each issue, the following sections are included:

1. **Summary**: a brief explanation of the issue.
2. **Description**: a detailed overview of the issue, e.g. steps to reproduce, etc.
3. **Attachments**: a list of URLs pointing to attachments of the issue.

4. **Commits:** if the source code repository is specified, all commits that address the issue are described. Commits are extracted from the source code repository and are filtered by the presence of the issue key in the commit message.
5. **Pull requests:** if the source code repository is specified, all pull requests that address the issue directly or refer to it are described. Pull requests are extracted from the source code repository and are filtered by the presence of the issue key in their title or description.
6. **Comments:** an overview of comments left during the issue discussion.
7. **Other issues:** if the issue refers to other issues, an overview of the same structure is produced for them as well and is included in the report.

The user is free to exclude any of these sections.

The tool is written in Python language. It combines all the data into the *LaTeX* format and automatically produces a PDF document using *PyLaTeX* library ¹.

5.1.2 Process flow

The report generation process can be split into the following stages:

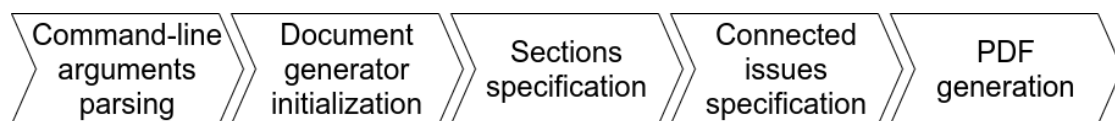


Figure 5.1: Report generation flow

1. **Command-line arguments parsing.** Initially, the user has to feed the tool with the following parameters:
 - (a) **Project name:** the name of the project as it appears in ASF JIRA (i.e. written in capital letters).
 - (b) **List of issues:** a number of issues to generate reports for. The issues are separated by command and can be defined as ranges. For example, "124,136-152,174" means "issues 124, 174, and from 136 to 152 inclusive". Once the list of issues is parsed, they are filtered in ascending order.
 - (c) **List of bots:** a comma-separated list of bot names. If the user wants to filter out messages generated by bots, e.g. a commit report, he/she can specify the names of bots, and thus, their messages will not appear in reports.

¹<https://jeltef.github.io/PyLaTeX/current/>

- (d) **GitHub repository:** if the GitHub repository is specified, then the user can retrieve commits and pull requests referenced by the issue.
 - (e) **List of sections to exclude:** if the user does not want specific sections to be included in the report, he/she needs to specify them, separating them with a comma.
2. **Document generator initialization.** For each issue, a report-generating object is created. During creation, it fetches and parses the issue details, as well as the details of the connected issue. When an issue is fetched and parsed, all the intermediate results are cached, i.e. persisted in JSON format. This allows us to avoid repeating data retrieval.
- If a GitHub repository is specified, then the tool additionally fetches and parses commits and pull requests. It is commonly accepted in Apache projects that commit messages targeting specific issues start with an issue key, for example, "PDFBOX-3017: replace method with library call". Pull requests are filtered by the presence of the issue key inside their title or the body.
3. **Sections specification.** Each section that is not in the list of the excluded ones is described in the following ways:
- (a) **Summary and Description.** These two sections are parsed in the same manner: they may contain code listings or *no-format* sections, so proper string manipulations are done to convert them to a LaTeX-friendly format.
 - (b) **Attachments** are represented as clickable names of files that are attached to an issue. Clicking a name redirects the user to a file location.
 - (c) **Commits.** Each commit is described as its short hash-value, the author, the date, and the commit message.
 - (d) **Comments.** First of all, the comments left by bots are filtered out. Then, for each comment, the same listing escaping is applied as for **Summary and Description**.
 - (e) **Pull request.** For each pull request, the title, the author, the date, and the status are defined. Then, each comment on the pull request is described by its author, the date, and the comment body.
4. **Connected issues definition.** Every connected issue is described in the same way as the "root" issue. The only difference is that we do not recursively describe issues connected to each of them.
5. **PDF generation.** All the data is converted to LaTeX format and compiled to a PDF document with the name being equal to the issue key.

5.1.3 Components description

Due to the nature of the Python language, the components of the project are divided into two categories:

1. Top-level .py file representing the entry point of the program.
2. Python modules that are folders containing .py files but acting as a unified structure.

The tool is defined by the following components:

1. **report-generator.py** script file: a Python top-level program file that is responsible for command-line arguments parsing and initializing the procedure of reports generation.
2. **genreport** module: a Python module responsible for composing parsed Jira issues and GitHub commits and pull requests into a PDF format with LaTeX acting as a format specifier. It contains the class **ReportGenerator** which sets up the LaTeX layout and makes use of the **jira_parser** module for retrieving issues details and the **github_fetcher** module for extracting commits and pull requests from a source code repository.
3. **jira_parser** module: a module responsible for fetching and parsing issues for the specified Apache Jira project. It contains the class **JiraParser** which fetches Jira issues and parses them. When **ReportGenerator** requests an issue details, **jira_parser** firstly checks whether the requested data is cached. If so, then it just returns the data. Otherwise, it performs several actions before the issue is prepared to be sent to **ReportGenerator**:
 - (a) The raw JSON-formatted description of the issue is requested from ASF JIRA. The following fields are included in the request:

comment, attachment, issuelinks, status, issuetype, summary, description, created, updated, project, creator

Once the issue is fetched, remote links (links to external sources of information) are requested separately. All the raw data is written to a corresponding file (cached).
 - (b) The raw issue data is parsed to filter out unnecessary data. The parsed data is also cached in JSON format to be reused for a repeated request.
4. **github_fetcher** module: a module responsible for fetching and parsing commits and pull requests from the specified GitHub repository. It contains the class **GitHubFetcher** which fetches GitHub commits and pull requests and persists them. Follows the same procedure as **jira_parser** to provide **ReportGenerator** with the necessary data.

5. **utils** module: a module providing reusable code, mainly for string operations. This includes reference extraction, conditional filtering of data, and substring substitution functionality, e.g. Jira code listing to LaTeX listing conversion.

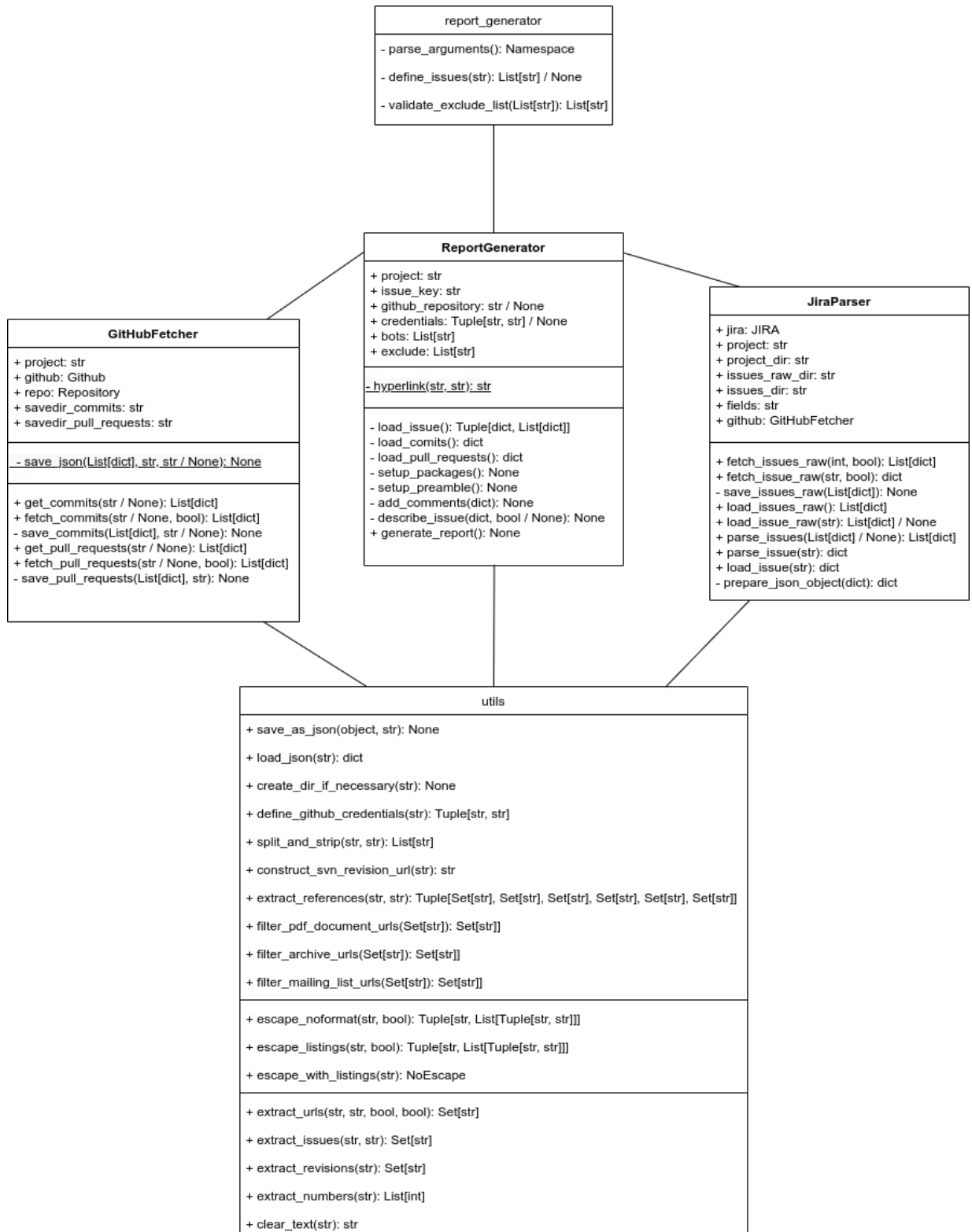


Figure 5.2: Architectural model

5.1.4 Technical challenges

1. **Caching.** The first obstacle to overcome is the number of requests required to retrieve data from Jira and GitHub. Referring back to Subsection 4.2.3, the user can retrieve blocks of Jira issues with a single request. However, several additional details can be received with a separate request only, such as remote links. Thus, the user should be aware of the number of requests he/she makes and avoid repetitive requests as it may lead to the IP address blocking. A similar problem is faced when making GitHub requests. The only difference is that if a project contains a huge amount of commits or pull requests, it may take a while to retrieve the data for a single issue.

It may be relatively hard to figure out exactly what data may be useful in the future. Moreover, there can be outstanding cases where the data is in an unusual format, for example, there is no author of an issue assigned (issue *PDFBOX-32*, the corresponding JSON value is "null") which may result into a hard-to-reproduce bug.

Such cases are easier to prevent if the user has access to intermediate results of each stage. Python has a built-in JSON encoder and decoder in its standard library ², with the ability to save/load the data to/from a file and convert a JSON string to a Python dictionary and vice versa. All results for each stage are written to the "Projects/<project_name>" directory to one of the following subdirectories:

- (a) **Issues_raw:** contains unparsed Jira issues in the same form as they are retrieved from ASF JIRA, with an additional top-level field "remotelinks". All issues are stored in JSON files with the name "<issue_key>.json".
- (b) **Issues:** contains parsed Jira issues ready to be used by the tool. All issues are also stored in JSON files with the name "<issue_key>.json".
- (c) **Commits:** contains fetched and parsed commits from the specified GitHub repository. the **github_fetcher** modules have the functionality to fetch commits, filter them by an issue key, and store in a file "<issue_key>.json". However, for our purposes, it is decided that there is a global pool of commits stored in a file "all.json", and when the tool tries to access commits addressing the specified issue, the module loads the file "all.json" and filters entries by itself.
- (d) **PullRequests:** contains fetched and parsed pull requests from the specified GitHub repository. Pull requests are saved and operated on in the same way as commits, i.e. using a shared "all.json" file.

²<https://docs.python.org/3/library/json.html>

2. **Text formatting.** The second problem arising is the necessity to convert sections of text into a LaTeX-friendly format. The key point here is that the text may contain code listings or *no-format* sections. Thus, we face an issue when we try to add the unmodified text to LaTeX as it most likely fails to compile due to special characters like `'_'` or `'{'`. PyLaTeX provides a nice way to escape all characters in a string, however, we do not want to lose the listing specifications, and instead, we want code blocks to be captured by LaTeX listings.

Consider the following example:

```
{code:java}
public static void main(String[] args) {
    System.out.println("Hello, World!");
}
{code}
```

We need to convert it to the following format without escaping:

```
\begin{lstlisting}[language=java]
public static void main(String[] args) {
    System.out.println("Hello, World!");
}
\end{lstlisting}
```

This is achieved by applying the following procedures:

- (a) Cut out a block of text of the form `"{code:<language>}<any_text>{code}"` using pattern matching and replace it with some unique identifier using pattern matching. In our case, the "flag" that is placed instead of the text snippet has the form `"<<!PDFGENCODEid!>>"`, where *id* represents the serial number of the snippet.
- (b) The *language* parameter is extracted from the block of text. `"{code:<language>}"` is replaced with `"\begin{lstlisting}[language=<language>]"` and `"{code}"` with `"\end{lstlisting}"`.

If the language is not specified, e.g. ”`{code}<any_text>{code}`”, then the block is replaced with ”`\begin{lstlisting}<any_text>\end{lstlisting}`”.

- (c) All special characters are escaped in the text. The flags do not contain any character that has to be escaped, so they remain unchanged.
- (d) Each flag is replaced with the formatted block of code.
- (e) The procedure is repeated until no code blocks are left.

Blocks of the text of the form ”`{noformat}<any_text>{noformat}`” are parsed similarly. Instead of listings, the *no-format* snippets are replaced with LaTeX verbatims.

5.2 System evaluation

The only way to check that the tool works properly is to stress it with several inputs, both valid and invalid.

We will take the *PDFbox* project as an example. Some tests are combined as they do not intersect with each other.

5.2.1 Valid input

We will start with tests that should successfully produce reports.

1. Use cases:

The user wants to

- (a) generate a complete report on the specified issue, so that he/she can extract knowledge from it
- (b) specify which issues he/she wants to generate reports for, so that he/she does not have to run the program for each issue separately

Description: suppose we want to retrieve issues 130-132 and 137. We want all sections to be included. Thus, we call the program with the following parameters:

- (a) **project:** *PDFBOX*
- (b) **issues:** *130-132,137*
- (c) **github:** *<https://github.com/apache/pdfbox>*

Result: 4 PDF documents are generated: *PDFBOX-130*, *PDFBOX-131*, *PDFBOX-132*, *PDFBOX-137*. Each document contains sections: *Summary*, *Description*, *Attachments*, *Commits*, *Comments*, *Pull requests*. Since the issues are relatively old, they have the author missing. Issues 130-132 have empty sections *Attachments*, *Commits*, and *Pull requests*.

Issue 137 has a single commit addressing it. Moreover, it has two connected issues: 222 and 1138. According to discussions, issue 222 is a duplicate of issue 137, and issue 137 was implicitly resolved after resolving issue 1138.

2. Use cases:

The user wants to

- (a) generate a reports without specified sections
- (b) have well-formatted code listings

Now we test the sections exclusion and the listing conversion. Issue 4861 is a suitable candidate. We want to include only the *Description* and the *Comments* sections. The parameters are:

- (a) **project:** *PDFBOX*
- (b) **issues:** *4861*
- (c) **exclude:** *summary,attachments,other_issues*

Description: we do not exclude *Commits* and *Pull requests* since the GitHub repository is not specified, so the tool has to exclude these sections by itself.

Result: the document contains only sections *Description* and *Comments*. *Description* contains a Java code snippet generated from the LaTeX Java listing.

3. Use cases:

The user wants to

- (a) filter out comments generated by bots, so that he/she can analyze discussions between developers without unnecessary information

Description: a number of comments can be generated by bots, and we may want to filter them out. Most comments in issue 4690 are left by the bot *jira-bot*. Since we are interested in filtering bots only, the program is called with the following parameters:

- (a) **project:** *PDFBOX*
- (b) **issues:** *4690*
- (c) **exclude:** *summary,description,attachments,other_issues*
- (d) **bots:** *jira-bot*

Result: we can observe that no comments left by *jira-bot* are present in the report.

4. Use cases:

The user wants to

- (a) retrieve the list of issue attachments so that he/she can analyze them separately
- (b) extend the knowledge of how an issue is resolved by analyzing discussions within GitHub pull requests

Description: the last case to test is the presence of attachments and pull requests in an issue. Our candidate is issue 3812. Parameters:

- (a) **project:** *PDFBOX*
- (b) **issues:** *3812*
- (c) **github:** *https://github.com/apache/pdfbox*
- (d) **exclude:** *summary,description,other_issues,commits,comments*

Result: each entry in the *Attachments* section is clickable and represents a URL to a file. An overview of each pull request and comments under it is present in the *Pull requests* section.

5.2.2 Invalid input

Now we will test the tool by providing it with invalid parameters.

1. Scenarios:

The user accidentally forgets to specify:

- (a) the project
- (b) the issues
- (c) the project and the issues

Description: **project** and/or **issues** parameters are missing.

Result: the program tells that either one of them (and which) or both are missing; no report generated.

2. Scenarios:

The user accidentally specifies:

- (a) no issues (empty string)

Description: **issues** parameter is an empty string: the program prints that at least one issue has to be specified.

Result: no report generated.

3. Scenarios:

The user accidentally specifies:

- (a) an invalid issue(s)

Description: **issues** parameter contains non-numeric data.

Result: the program prints that the format of the issues list is invalid; no report generated.

4. Scenarios:

The user accidentally specifies:

- (a) an issue(s) that do not exist

Description: **issues** contain a non-existing issue. As of July 20th, 2020, *PDF-Box* has 4914 opened and closed issues. Consider calling the program with the following parameters:

- (a) **project:** *PDFBOX*
- (b) **issues:** *4000-4003,5000-5500,4500*

Since the issues are sorted in ascending order, then the program should terminate once it reaches a non-existing issue. It is derived from the assumption that if issue 5000 does not exist, then issues 5001, 5002, etc. do not exist as well.

Result: successfully generated reports for issues 4000, 4001, 4002, 4003, and 4500; a message that issue 5000 does not exist.

5. Scenarios:

The user accidentally specifies:

- (a) an invalid GitHub repository

Description: `github` parameter represents a non-existing URL to a GitHub repository or invalid URL.

Result: the program prints that the GitHub repository is invalid; no report generated.

6. Scenarios:

The user accidentally specifies:

- (a) an invalid section to exclude from reports

Description: the `exclude` parameter contains an invalid section. The program is run with the following parameters:

- (a) **project:** `PDFBOX`
- (b) **issues:** `4000`
- (c) **exclude:** `abc,summary,description,def`

Result: the program prints that "abc" and "def" are invalid sections. No report generated.

7. Scenarios:

The user accidentally specifies:

- (a) all sections to be excluded from reports

Description: the `exclude` parameter contains all sections:

- (a) **project:** `PDFBOX`
- (b) **issues:** `4000`
- (c) **exclude:** `summary,description,attachments,commits,pull_requests,comments,other_issues`

Result: the program prints that all sections are excluded. No report generated.

6

Conclusion

Issue discussions are a great source of knowledge of project evolution processes. Apart from problem-solving dialogues, they may contain references to external sources, such as other issues, pull requests, forums, and Q&A websites. However, the person who decides to extract knowledge from issues faces a major obstacle: discussions may contain an enormous amount of data to parse, and all the referred sources have to be manually analyzed. This fact dramatically increases the complexity of issue analysis, and if an automated data-gathering solution is developed for one project, it is not guaranteed that it will suit other projects of choice. Thus, a generic solution for linking discussions has to be developed.

Pattern matching is found being an effective approach to capture references of different types. Issue IDs follow the format `<project_name>-<numerical_ID>`, so a connection with another issue of the project or even an issue of another project can be established. Apache Subversion system uses 7-digit decimal numbers with an "r" letter in front of them to display revision IDs. GitHub commit IDs are represented as 40-digit hexadecimal numbers or their shortened 7-digit versions.

However, one type of references requires a more complex analysis to be classified: URLs. Sometimes, the content of a URL can be guessed by analysing its postfix. For example, URLs ending with ".pdf" most likely point to PDF documentations, while prefixes ".zip" and ".tar" describe archives that may contain snapshots. However, URLs that lead to mailing list discussions are hard to be distinguished: if it contains the substring "mail" or even "mailinglist" do not guarantee that the URL leads to a discussion. One

possible solution for that is to search for the occurrence of a numerical ID inside the URL.

6.1 Future work

Statistics collection helps us to determine the periods of a project development that contain a high density of references. For URLs, there is a chance to discover new categories of links, which may dramatically help in the development of the URL classification approach. Thus, a deeper analysis of non-classified URLs may help to discover new types of references.

The prototype of the issue discussions collection tool uses issue keys and commit IDs as connecting attributes at the moment. Further development implies making use of URLs to extract the data from the sources they are pointing to. In this way, the missing parts of discussions can be gathered from forums that may contain crucial information about the development process of a project.

There is a big number of issue tracking systems that provide a comprehensive source of knowledge, however, this research aims at Jira issues only, what is more, only Apache projects are analysed. Without considering other issue trackers, it may be hard to develop a generic approach in extracting references of different types. Therefore, it will be useful in the future to extend the research area to other issue tracking systems such as *Planio*¹ or *Backlog*².

The same goes for version control systems. Although Git and Apache Subversion are in high demand, there are other products that are heavily used, for example, *Mercurial*³ - a distributed revision-control system. It may happen that during discussions, developers may refer to other projects, and there is a chance that one of such projects uses Mercurial. As it should be always assumed that each reference may contain the required knowledge, such cases have to be captured as well.

¹<https://try.plan.io/issue-tracking-gdm-lp/>

²<https://backlog.com/>

³<https://www.mercurial-scm.org/>

Bibliography

- [1] D. L. D. M. Le, A. Shahbazian, and N. Medvidovic, “An empirical study of architectural decay in open-source software,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, 2018, pp. 176–185.
- [2] D. Sas, P. Avgeriou, and F. A. Fontana, “Investigating instability architectural smells evolution: An exploratory case study,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA, 2019, pp. 557–567.
- [3] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu, and H. Fang, “Active hotspot: An issue-oriented model to monitor software evolution and degradation,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA, 2019, pp. 986–997.
- [4] D. Bertram, A. Voids, S. Greenberg, and R. Walker, “Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams,” 01 2010, pp. 291–300.
- [5] C. Scott and S. Ben, *Pro git*. Apress, 2014.
- [6] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, “What is subversion?” [Online]. Available: <http://svnbook.red-bean.com/en/1.7/svn.intro.whatis.html#svn.intro.history>
- [7] I. Skerrett, “Eclipse community survey 2014 results,” 2014. [Online]. Available: <https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/>
- [8] K. Johnson, “Github passes 100 million repositories,” 2018. [Online]. Available: <https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/>
- [9] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, “An empirical study of architectural change in open-source software systems,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, Florence, 2015, pp. 235–245.

- [10] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies, “What is the connection between issues, bugs, and enhancements?” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, Gothenburg, 2018, pp. 306–315.
- [11] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic, “Recovering architectural design decisions,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, 2018, pp. 95–104.
- [12] “All sites - stack exchange,” <https://stackoverflow.com/sites?view=list#users>, [Accessed: 25-02-2020].

7

Appendix A: ReadMe

This section describes the usage of the report generating tool.

Requirements

The following software has to be installed on the target machine:

1. **Python 3.7**
2. **LaTeX compiler** (pdfLaTeX or Latexmk)
3. **pip** package manager

Python libraries

The following Python libraries are required:

1. **jira**: a Python wrapper around Jira API
2. **PyGithub**: a Python wrapper around GitHub API
3. **pypallex**: a library for creating and compiling LaTeX files

They can be installed via *pip* with the help of the *requirements.txt* file:

```
pip install -r requirements.txt
```


Usage (analyzer)

```
analyzer.py [-h] -p PROJECT [-g GITHUB] [-c CREDENTIALS]
```

Command-line arguments

1. `-h, --help`: show help message and exit
2. `-p PROJECT, --project PROJECT`: Target Jira project in capital letters (compulsory)
3. `-g GITHUB, --github GITHUB`: Target Jira project's GitHub repository
4. `-c CREDENTIALS --credentials CREDENTIALS`: GitHub username and password separated by comma. Compulsory if GitHub repository is specified

The arguments that are separated by comma also allow whitespaces around commas, for example, "124, 136-152 , 174". All the whitespaces are trimmed once the string is split by comma.

Example

```
analyzer.py -p "PDFBOX"  
            -g "https://github.com/apache/pdfbox"  
            -c "github_username,github_password"
```

Usage (report generator)

```
report_generator.py [-h] -p PROJECT -i ISSUES  
                   [-g GITHUB] [-c CREDENTIALS]  
                   [-b BOTS] [-e EXCLUDE]
```

Command-line arguments

1. `-h, --help`: show help message and exit
2. `-p PROJECT, --project PROJECT`: Jira project in capital letters as it appears in ASF JIRA (compulsory)

3. `-i ISSUES, --issues ISSUES`: Issues to generate reports for, separated by comma and/or defined as ranges. For example, "124,136-152,174" (compulsory)
4. `-g GITHUB, --github GITHUB`: Target Jira project's GitHub repository
5. `-c CREDENTIALS --credentials CREDENTIALS`: GitHub username and password separated by comma. Compulsory if GitHub repository is specified
6. `-b BOTS, --bots BOTS`: List of bots to exclude from report, separated by comma
7. `-e EXCLUDE, --exclude EXCLUDE`: Sections to skip when generating report, separated by comma. Sections are: [summary, description, attachments, commits, pull_requests, comments, other_issues]

The arguments that are separated by comma also allow whitespaces around commas, for example, "124, 136-152 , 174". All the whitespaces are trimmed once the string is split by comma.

Example

```
python report_generator.py -p "PDFBOX"  
                          -i "3017,4000-4005,4015"  
                          -g "https://github.com/apache/pdfbox"  
                          -c "github_username,github_password"  
                          -b "jira-bot,githubbot"  
                          -e "attachments"
```

8

Appendix B: Plots

In this section, you can find the plots for the rest of categories of references.

Archives

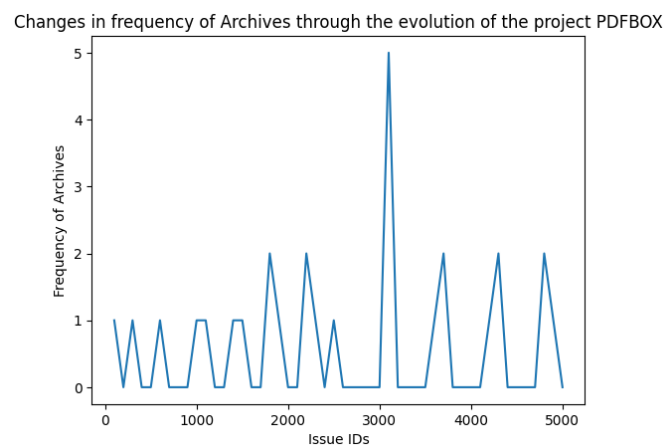


Figure 8.1: Frequency of archive references throughout the PDFBox project evolution

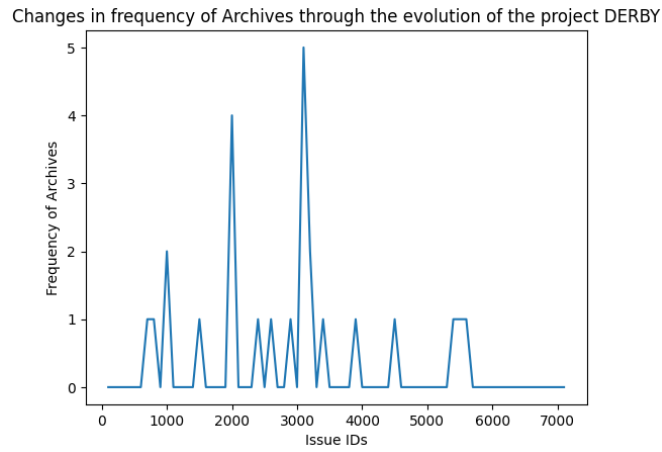


Figure 8.2: Frequency of archive references throughout the Derby project evolution

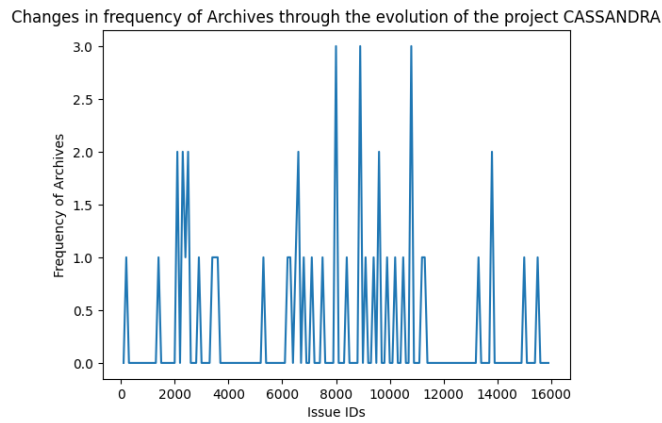


Figure 8.3: Frequency of archive references throughout the Cassandra project evolution

Mailing Lists

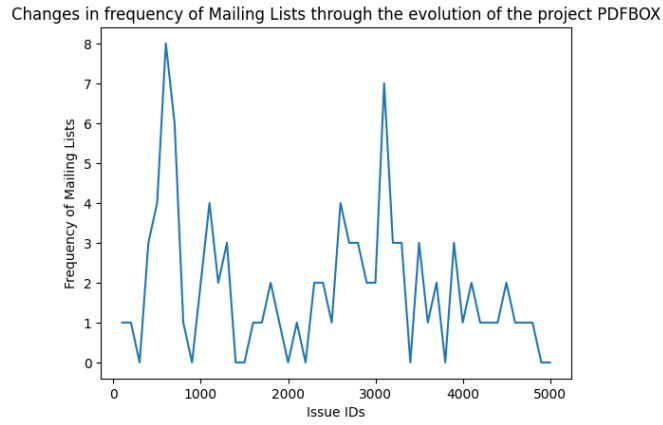


Figure 8.4: Frequency of mailing list references throughout the PDFBox project evolution

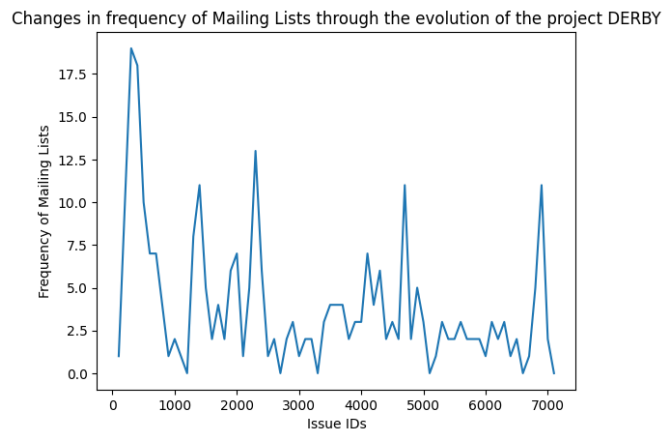


Figure 8.5: Frequency of mailing list references throughout the Derby project evolution

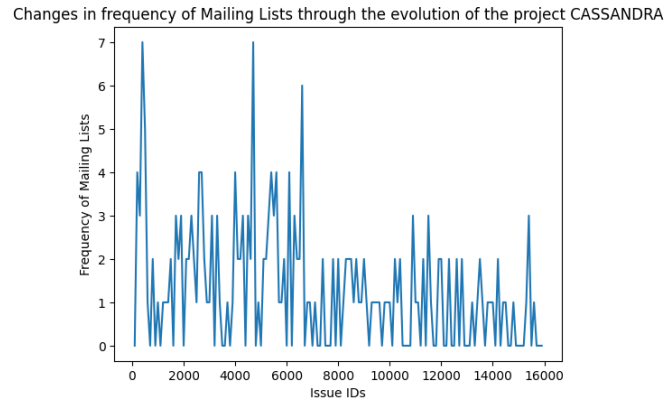


Figure 8.6: Frequency of mailing list references throughout the Cassandra project evolution

Other URLs

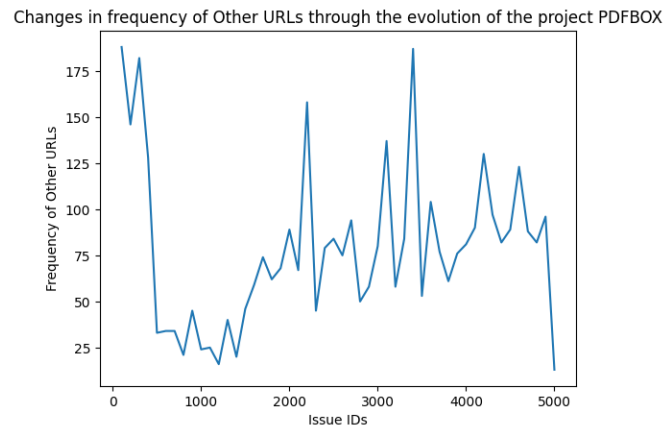


Figure 8.7: Frequency of other URLs references throughout the PDFBox project evolution

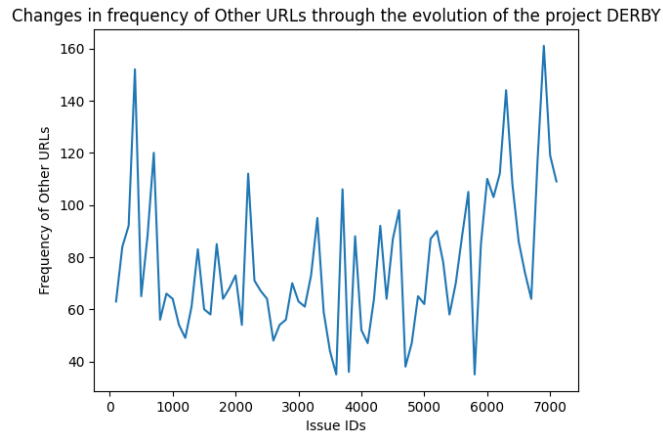


Figure 8.8: Frequency of other URLs references throughout the Derby project evolution

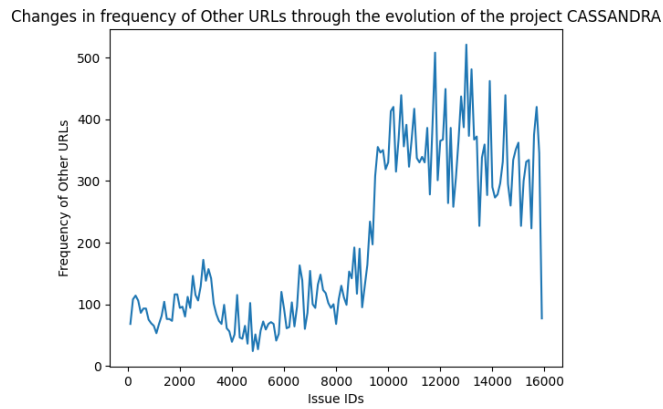


Figure 8.9: Frequency of other URLs references throughout the Cassandra project evolution

PDF documents

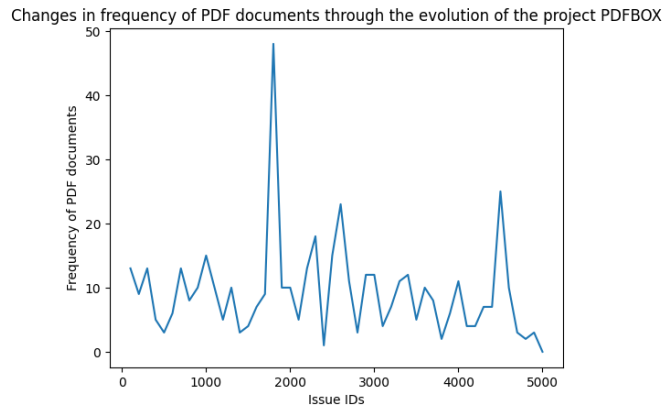


Figure 8.10: Frequency of PDF document references throughout the PDFBox project evolution

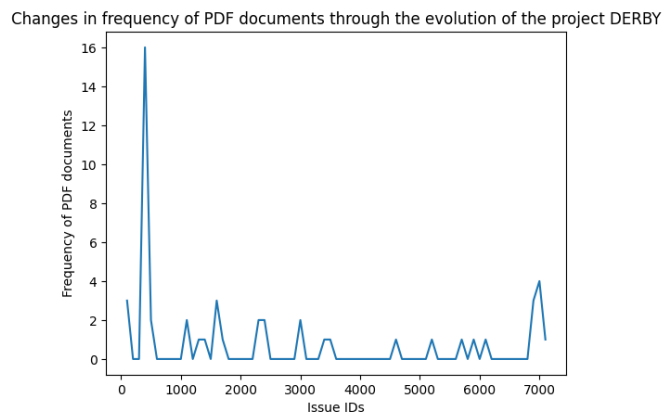


Figure 8.11: Frequency of PDF document references throughout the Derby project evolution

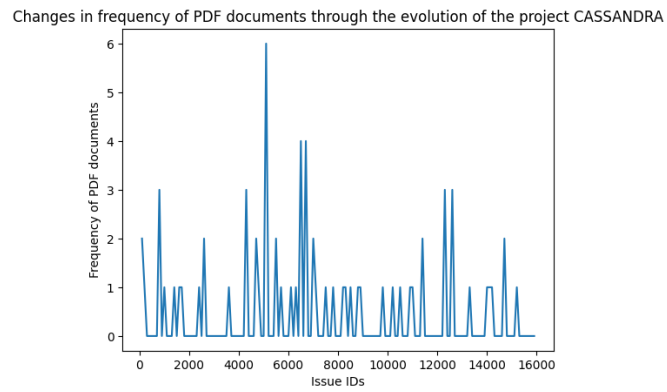


Figure 8.12: Frequency of PDF document references throughout the Cassandra project evolution

Commits

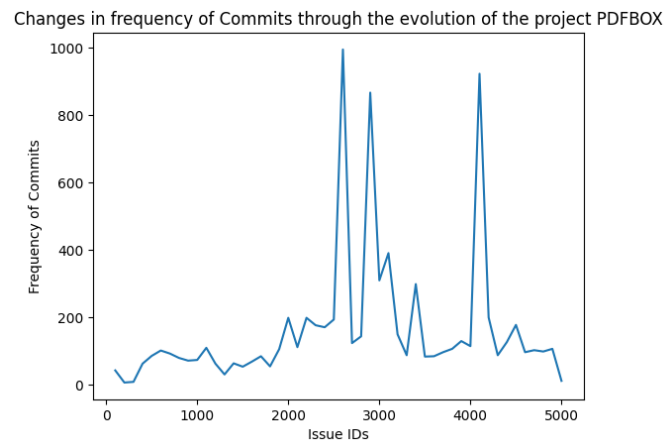


Figure 8.13: Frequency of commit references throughout the PDFBox project evolution

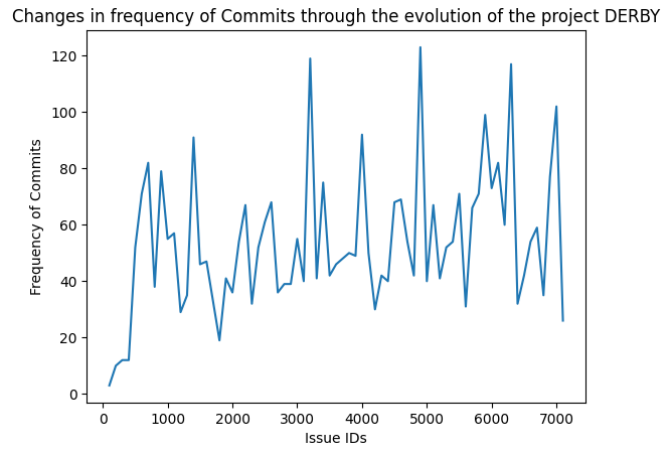


Figure 8.14: Frequency of commit references throughout the Derby project evolution

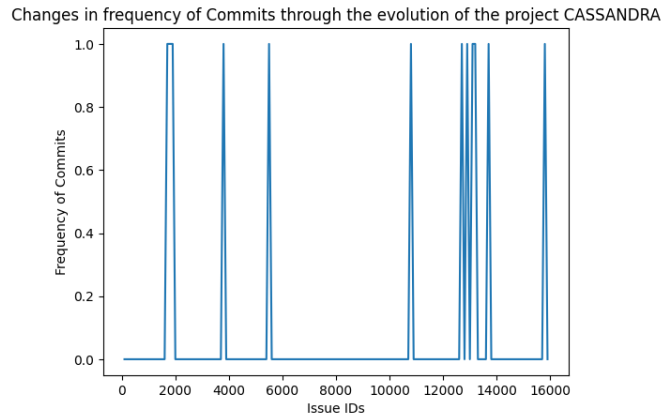


Figure 8.15: Frequency of commit references throughout the Cassandra project evolution

Pull requests

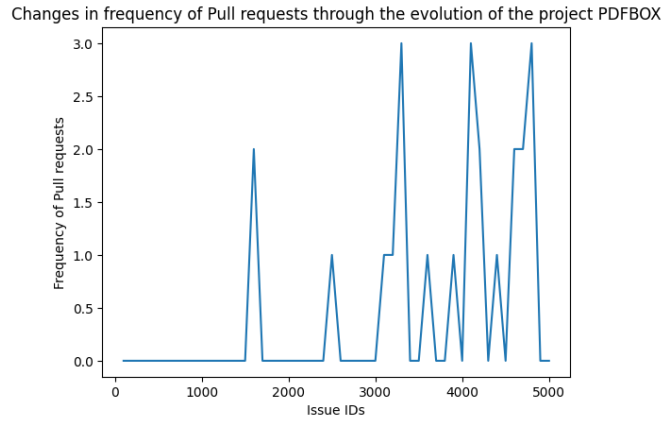


Figure 8.16: Frequency of pull request references throughout the PDFBox project evolution

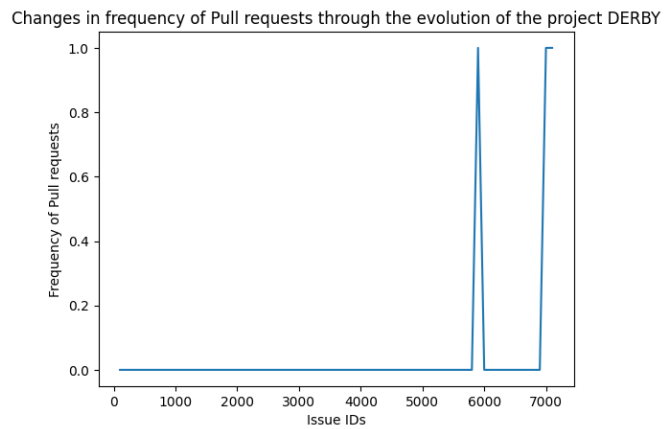


Figure 8.17: Frequency of pull request references throughout the Derby project evolution

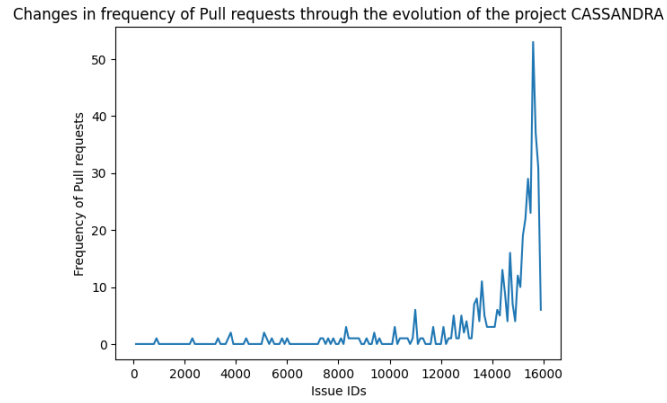


Figure 8.18: Frequency of pull request references throughout the Cassandra project evolution

Total references

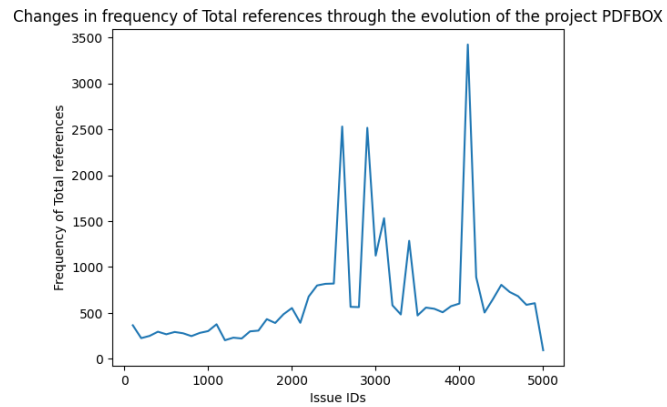


Figure 8.19: Total frequency of references throughout the PDFBox project evolution

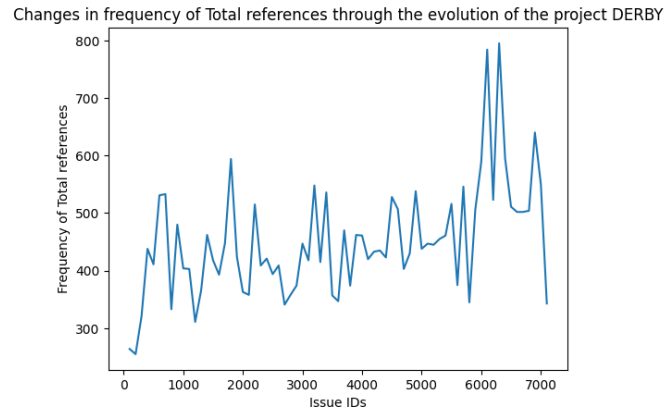


Figure 8.20: Total frequency of references throughout the Derby project evolution

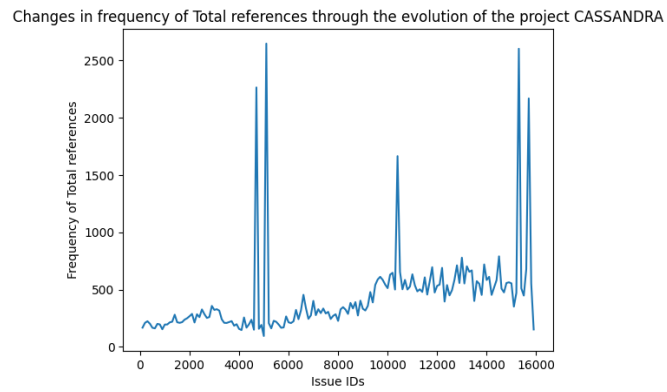


Figure 8.21: Total frequency of references throughout the Cassandra project evolution