Combined method for mining and extracting processes, related events and compliance rules from unstructured data

Deliverable D3.3 version 2

FFG – IKT der Zukunft
SHAPE Project
2014 – 845638

Table 1 Document Information

Project acronym:	SHAPE
Project full title:	Safety-critical Human- & dAta-centric Process management in Engineering projects
Work package:	3
Document number:	3.3 v2
Document title:	Combined method for mining and extracting processes, related events and compliance rules from unstructured data
Version:	2
Delivery date:	01 October 2016 (M3)
Actual publication date:	————————
Dissemination level:	Public
Nature:	Report
Editor(s) / lead beneficiary:	WU Vienna
Author(s):	Saimir Bala, Cristina Cabanillas, Jan Mendling, Axel Polleres
Reviewer(s):	Alois Haselböck, Cristina Cabanillas

Contents

1 Introduction

Deliverable 3.3 version 2 of the SHAPE project[1] reports work performed under Task 3.4 *Mine processes, resource consumption, witnesses for task completion from gathered events*. This document meets milestone **(M3)** *Combined method for mining and extracting processes, related events and compliance rules from unstructured data*.

In the previous version of D3.3 Cabanillas et al. [2016b], we studied the applicability of combined methods for mining projects out of unstructured and structured data. Specifically

- We presented a data model to access and analyze projects. We have identified the main entities and their relationships that are typical in Version Control System (VCS) and have constructed a schema. This allows for storing data into a database and enables interesting queries. This approach is more scalable since it allows for more flexibility in getting insights into projects without the burden of encoding new approaches from scratch.
- We studied possible text mining techniques in order to enrich the information provided by the mining algorithm in Cabanillas et al. [2015c]. Text mining can help to assign labels to the activities of a Gantt chart. In this deliverable we have tested a) topic modeling from emails; and b) semantic analysis of VCS comments.
- We developed and tested an algorithm that combines both statistical information and commit-comments from VCS repositories to classify users according to roles.
- We presented ongoing work in mining Camunda logs that is aware of resource scheduling.
- We presented ongoing work on developing a query language to access Camunda history logs through an SQL-like console.

This deliverable refines the previous version by further research on text and process mining. Specifically, we extend D3.3 version 1 with the following contents.

- Mining project phases
- Analyzing the language used by software engineers
- Story mining from software repositories
- Enabling SQL querying to Camunda logs
- Implementation of a process miner component within the a comprehensive framework for Complex Engineering Processes

[1] https://ai.wu.ac.at/shape-project/

This new version of Deliverable 3.3 is organized as follows. Section 2 presents the model we use to capture VCS data. Section 3 introduces shows how we intend to use text mining to support for better insights on projects. Section 4 presents work on mining software engineering processes or projects. In particular, it shows how to we use text comments from software repositories to discover the Rational Unified Process (RUP) model. Section 5 presents advances for using existing process mining techniques by converting the de facto XES standard for process mining into a set of relational tables. This allows for process mining through SQL queries. Section 6 presents a combined techniques for mining roles from VCS comments. Section 7 shows the implementation of a mining and monitoring component, in the setting of the general SHAPE framework. Section 8 concludes the deliverable.

2 Mining VCS data for project-oriented processes and resources

In this section we analyze how we can mine useful insights from processes that use VCS. Engineering projects like the ones in SHAPE make use of version control systems to keep track of their documents. While the project evolves, the amount of information in these systems increases. New needs may emerge during the project lifetime or post completion. For instance, new rules and regulations may come into play, which have to be obeyed immediately; or a posteriori checks may be required from an auditor to manually check for unsafe patterns in the work that was done. Storing project data into a database allows to systematically access those data. Furthermore, it gives the flexibility to respond to new requirements without necessarily developing new algorithmic approaches to mine the data. For instance, new requirements can be checked by issuing a proper query to the database. In this section we will discuss our data model and show how we can use queries to analyze projects.

2.1 Data model

Here we give an overview of our data model. VCS are captured through the data model in Figure 1 using the Entity-Relationship notation from Chen [1976]. The entities and their relationships are described as follows.

Commit stores information about a checkout in the repository. Each checkout has an id or revision number, a timestamp and may have a comment. Comments are used from project members to describe their contribution. A commit

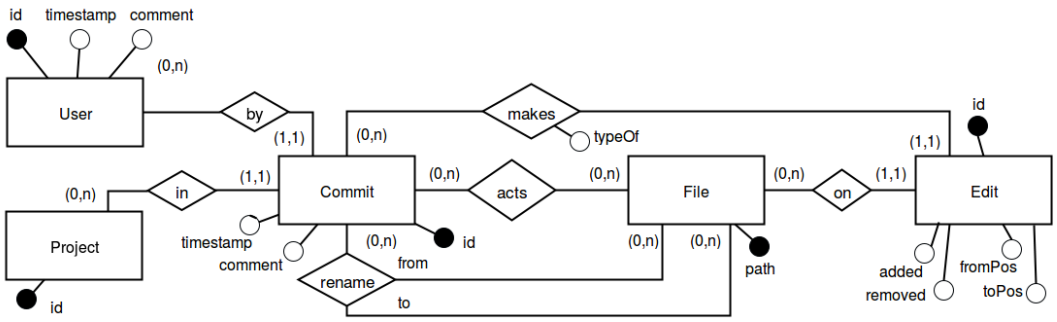


Figure 1 Entity-Relationship Diagram from a VCS

can be of different types, e.g. it can store a merge between two commits or it can store file changes. Files, users, and edits on files are considered as separate entities in our model.

File represents a file that is present in the repository. A file is typically changed by a commit. We take into account also renames that are made to the file as a triple relation among two files and one commit.

Edit stores a modification of a *part* of the file. We use this entity to record where and how much a file was changed. This allows for fine grained analysis as we will see later in Section 2.2.

User is the resource who commits on the repository. A user has an id, a name and an email. Users of a project may issue an arbitrary number of commits.

Project is used to keep track of the project to which the set of commits belongs.

2.2 Querying data

With the data modeled as in Figure 1 it is possible to plan for several types of queries. We have used this model to store data into a MySQL[2] database. Database users are hence enable to issue suitable queries for their requirements. As an example, we show three possible queries. To this end we fed into our model the Camunda BPM[3] repository. We make the following queries.

1. Which are the contributions to the project of a particular user X? (Listing 1)
2. What changes have been made to a particular file over time? (Listing 2)
3. Who are the users that have worked on feature development? (Listing 3)

The queries have been written in SQL as follows.

[2] https://www.mysql.com/
[3] https://github.com/camunda/camunda-bpm-platform

```
SELECT name, Commit.id as CommitId, timeStamp, comment,
   linesAdded, linesRemoved, linesAdded-linesRemoved
FROM User, Commit, Edit
WHERE User.id = Commit.user_id
AND User.id = 1
AND Edit.commit_id = Commit.id
GROUP BY Commit.id
ORDER BY timeStamp ASC
```

Listing 1 Timeline of a user's contributions

```
SELECT sum(`linesAdded`),sum(`linesRemoved`), sum(
   linesAdded)-sum(linesRemoved) as delta,`timeStamp`,`
   comment`,`Commit`.`id`,`file_path`
FROM `Edit`,`Commit`
WHERE `Edit`.`commit_id`=`Commit`.`id`
AND `file_path`="engine/pom.xml"
group by timeStamp
order by `timeStamp` ASC
```

Listing 2 Changes to the camunda engine configuration file

```
SELECT name, Commit.id as CommitId, timeStamp, comment,
   sum(linesAdded), sum(linesRemoved), sum(linesAdded-
   linesRemoved) as Delta
FROM User, Commit, Edit
WHERE User.id = Commit.user_id AND Edit.commit_id =
   Commit.id AND Commit.comment LIKE '%fix%'
GROUP BY Commit.id
order by timeStamp ASC
```

Listing 3 Users who worked on feature development

Listing 1 returns all the changes made to the repository from the user who has `id = 1`. Such user might have made multiple edits in one single commit. Thus, we aggregate by commit the numbers of lines added, lines removed and their difference which is also referred to as *delta*. Lastly we order these edit event from the oldest to the most recent.

Listing 2 shows the changes to a particular file. We chose the file named `"engine/pom.xml"` which is a configuration file that is supposed to change frequently due to updates, new releases (alpha, beta, etc) or tests. Here we want to see how the file changes over time. This can help to identify patterns that

might exist in the development. For instance, we expect that when certain task is approaching completion, then the file changes become stable. This also why we order the results chronologically.

Listing 3 returns a list of user who have been working on a specific task. Here we exploit the commit-comments of the Camunda history log. Camunda comments include special tags like `fix(engine)`, `fix(platform)`, etc, for indicating what the changes were about. In this query we select all commit where the users have mentioned the word *fix* somewhere in the comment. Furthermore, we select the amount of changes and order the users chronologically.

2.2.1 Using timelines for project analysis

The queries of previous Section 2.2 can be exploited further to obtain more information. The extracted results can be exported and fed into other tools or data mining algorithms can be applied on top of them. As a showcase, we have analyzed our data with R[4] and have plotted the results for different queries.

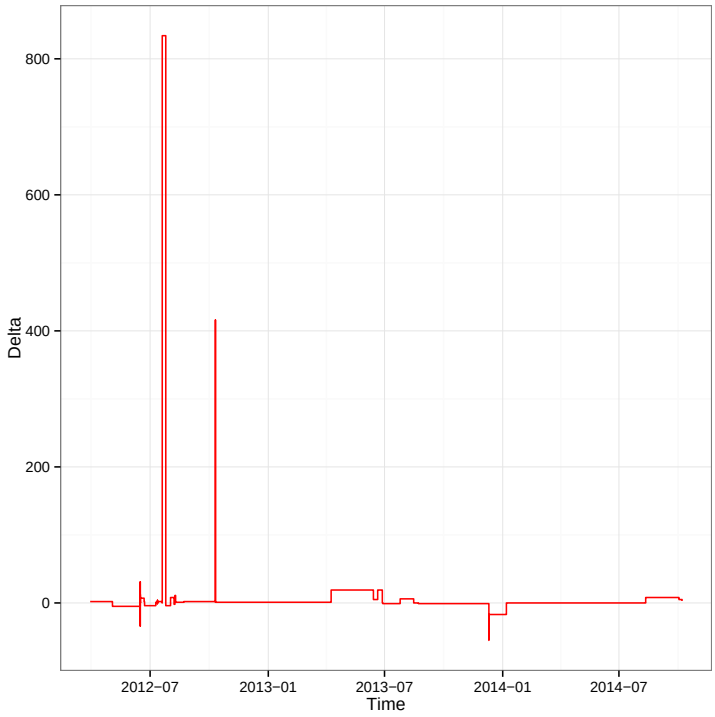
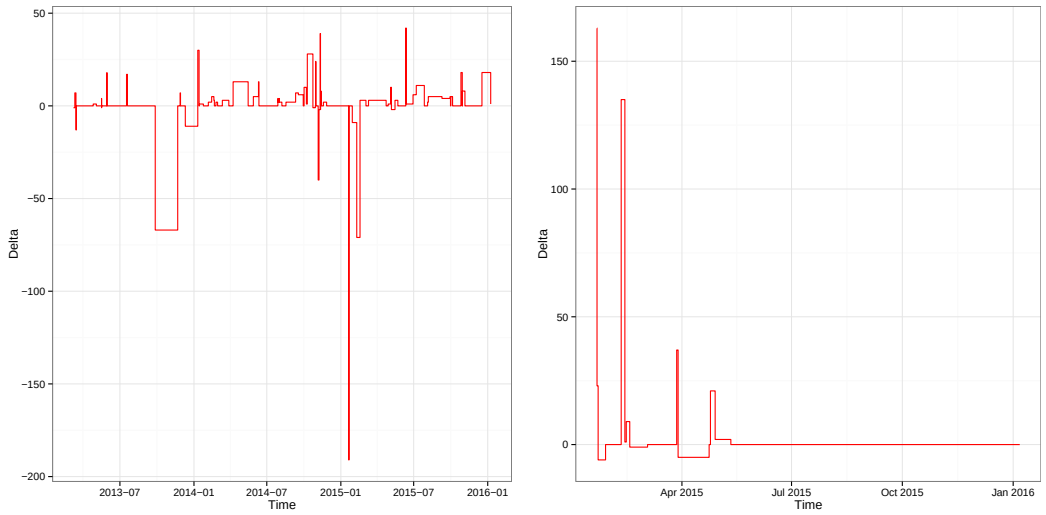


Figure 2 Profile of the changes made by one user. In the Y-Axis is reported the total amount of change that has been done by the selected user

[4] https://www.r-project.org/



(a) File under development **(b)** Database configuration file

Figure 3 An example of two different file-change patterns

Figure 2 shows the changes of a specific user over time on the repository. The Y-Axis is the amount of change *delta*, which is the difference between lines added and the lines removed from all files changed by the selected user. The data comes from the results of the query in Listing 1. It is now easy to see that user with `id = 1` has been generally not very active with two high peeks around his/her first appearance.

Figure 3 shows how we can use the query in Listing 2 to distinguish different types of work reflected by file changes. As an example, we run the query for two different types of files where the distinction is evident. Figure 3a reports the change over time of a file that belongs to the core development of Camunda. Figure 3b plots changes of a database configuration file of Camunda. The graphs make it clear to see such change patterns.

We can use this kind of data to classify the type of work, the users and also gather further insights about the extent to which rules and regulations may be followed. Irregular patterns or artifacts that should be created, such as for instance validation reports, or comments about extensive testing that are not reflected by the changes in the VCS logs may be indicators of deviations. In Section 2.3 we give an example of rules and guidelines in software development projects that use VCSs.

2.3 Rules and guidelines

Rules and regulations have to be taken into account when it comes to check for process compliance. European Standards such as the EN 50128 software development process mus be followed in order ensure safety and reliability. Developer teams typically rely on VCS to manage their workflow. Thus, these systems can help to discover possible deviation from the desired flow. In the following, we will describe practical ways of using VCS for development. For details we point to online resources such as Driessen [2010] and Atlassian [2016].

2.3.1 Typical Workflows in VCS

Let us see an overview of four typical workflows on Git and discuss how they can be a) identified in a repository b) used to support mining.

2.3.1.1 Centralized Workflow

The centralized workflow uses a central point-of-entry for all the changes to the project. It works as follows. Each user has a local copy of the repository. All their edits to files get stored locally when they commit. To make a change visible in the main project streamline, users have to publish their changes (e.g. through commands `git push` or `svn commit`). Conflicts can occur when users try to push content that is already present in the main repository under another version. In such a case, users need to first `pull` the possible changes from the main trunk and resolve possible conflicts. Afterwards users can upload their edits. This workflow maintains a linear history.

2.3.1.2 Feature Branch Workflow

The feature branch workflow is based on the idea that the main repository, also called `master` branch, should contain only clean artifacts. In case of software development, it means that the features are developed in a dedicated branch and later merged into the main codebase. When a feature is ready, it is possible to issue a pull request. This triggers user review and conversation on the possible improvements of the feature before it gets accepted. The feature branch workflow is very flexible, allowing users to create as many branches as they want. However, sometimes this has the drawback that a common way of work is difficult to identify. The Gitflow Workflow discussed below provides a common pattern for managing feature development, release preparation, and maintenance.

2.3.1.3 Gitflow Workflow

The Gitflow Workflow defines a strict branching model designed around the project release. It assigns very specific roles to different branches and defines how and when they should interact. It uses individual branches for preparing, maintaining, and recording releases. Like in other models, Gitflow still uses a main `master` branch for developers to publish their local changes. The difference lies in the branch structure.

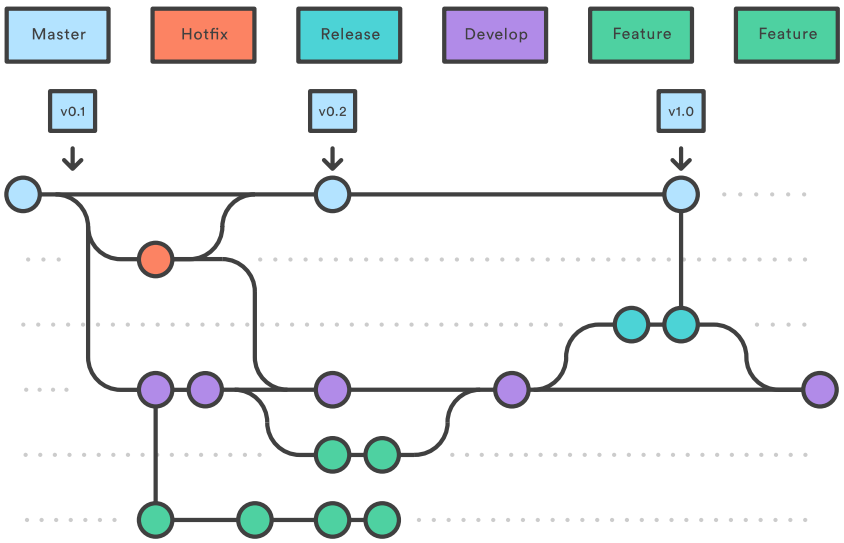


Figure 4 The Gitflow branches. Each branch serves to a specific purpose. Picture from Atlassian [2016]

Figure 4 shows how dedicated branches are created for a specific use. The main branch `master` tracks the official software releases. The most important branch after master is `develop`. `develop` serves as an integration branch for new features. Each feature is developed in its own branch that forks off `develop`. When a number of features are merged again into `develop`, the users may want to prepare a new release. To this end, they fork off a `release` branch from `develop`. A pull request to `master` may then trigger code review and discussions among developers. If the new release is accepted, it can then be published on the `master` branch and tagged with a version number. In case the end user discovers a bug and a quick fix becomes necessary, then a `hotfix` branch is created directly from `master`. As soon as the issue is solved, a new version is committed both in `master` and `develop`. In this way urgent problems are solved without going through the whole release cycle.

2.3.1.4 Forking workflow

In the workflows introduced above, users had a local copy of a central repository, namely a clone. Here, instead of a single server-side repository, each user has its own server-side copy of the official repository, namely a fork. Users have also a local copy derived from cloning their own server-side copies. The advantage here is that each user is concerned only with publishing on its own repository. A project maintainer can then pull from different projects into the official one, without having to give write permissions to the users. It works as follows.

- The project maintainer initializes the official repository
- Developers fork the official repository
- Developers clone their forked repositories
- Developers work on their features
- Developers publish their features
- The project maintainer integrates their features
- Developers synchronize with the official repository

Forking workflow implements a totally distributed workflow where developers are not tied to one team but they can share code with any other developers and any change can be merged into a project at any time.

3 Exploiting text mining techniques for mining projects

This deliverable uses text mining to exploit unstructured data that may give evidence on compliance deviation or gather new insights into the engineering projects of SHAPE. This is the case of e-mails and commit messages from VCS logs. In this section we consider two different approaches to text mining *i)* topic modeling; and *ii)* semantic annotation. We have evaluated both approaches with our data in order to better understand the possible future improvements of these techniques that fit our goals.

3.1 Topic models

Topic models are a set of methods that aim to discover a common theme in document collections. These document collections may talk about several topics. Topics are a set of words which are distributed over a vocabulary. Topic modeling algorithms are able to find topics and group them together into clusters. A simple yet extensively used topic model is given by Latent Dirichlet Allocation (LDA), published in Blei et al. [2003].

Table 2 Four topics extracted from SHAPE emails

Topic 1	Topic 2	Topic 3	Topic 4
axel	cristina	shape	saimir
saimir	cristinacabanillas	axel	deliverable
shape	shape	polleres	bala
polleres	next	information	tudor
stefan	information	simon	shape
week	claudio	business	architecture
cristina	meeting	meeting	review
will	deliverable	date	please
giray	agenda	steyskal	alois
paper	site	missing	project

3.1.1 Topic modeling on SHAPE emails

In this section we show an application of LDA to the email collection from SHAPE. We have downloaded all the emails from the SHAPE mailing list and applied the approach described in Feinerer et al. [2008]. To this end we have created a single text file for each email. The data have then been imported into R. The procedure involves classic text-mining preprocessing steps such as stop-words removal, tokenization, etc, which are already provided by the `tm` and `topicmodels` libraries.

Table 2 shows the topics for the emails of SHAPE. We cluster our documents into `k = 4` topics of which the ten most popular words are reported. Topics represents words which are highly correlated. For instance, in this case, people who have more closely worked together have exchanged more emails with each other. Hence, they get clustered under the same topic.

An advantage of topic models is that no annotation on the text is required as a prior step. However, sometimes topics alone are not very informative. Moreover, choosing the right number of topics is not straightforward. This work-package plans to investigate further on combining topic models with the approach from Bala et al. [2015] that mines Gantt charts. Topics can be mined only on the emails that where exchanged during active times slots that our found by process mining VCS logs. Then, such topics may be used to label the activities on the Gantt chart.

3.2 Semantic model

Here we present another text mining approach to learn from unstructured data. While topic models are statistical methods that do not take into account the relations among words or the structure of a sentence, semantic models allow us to exploit words hierarchies or similarities. Text from software repositories or emails

can be semantically annotated and analyzed using such semantic approach. One interesting task that allows us to learn more about comments that people write in VCSs is to categorize their commits. We take inspiration from Leopold et al. [2012], who classify process labels into categories, to classify commit-styles. In this section we investigate to what extent it is possible to understand the commit-styles by analyzing them as possible labels of business process activities. We adapt the label-styles to commit-styles as shown in Table 3.

Given the diversity of projects that can be retrieved from open source repositories and the fact that comments may have worse wording with respect to activity-labels, we expect the results to change from a repository to another.

We applied the rules in Table 3 to two different repositories.

- ProM - from academia
- Camunda - from industry

Table 4 shows the results of our classification of user comments. We can see that there are many comments that are not classified. This is due to their sentence structure. One big difference between activity labels VCS and comments is their length. As suggested by Table 4, in order to classify more comment-styles we need to take into account an average sentence size of at least ten words. However, the results still show how Camunda uses a better commit-style with respect to ProM, which gives more freedom to its users. In future work we plan to improve our classification method to be able to deal with the length of the commits from VCSs. It will be then possible to use commit-styles to classify users, work, or possible deviation (e.g. ambiguous description of work that may lead to process deviations).

Table 3 Commit styles

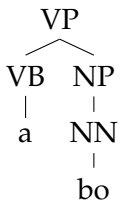
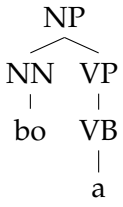
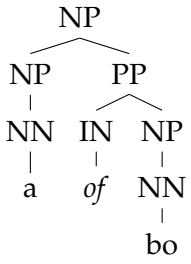
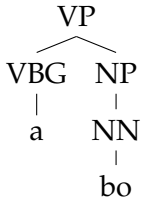
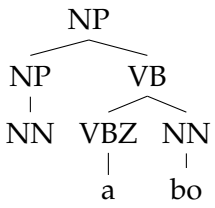
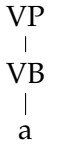
Commit style	Structure	Example
Verb-Object VO		update README.md
Action-Noun AN (np)		jar update
Action-Noun AN (of)		Creation of launchers
Action-Noun AN (ing)		reviewing code
Action-Noun AN (irregular)	`<irregular structure>`	(CAM-A14) bugfix#11
Descriptive DES		A left-over is removed now.
Only-Action OA		fix
Only-Object OO		feature

Table 4 Commit styles in Camunda and ProM

Property	ProM	Camunda
VO	28.00%	48.38%
AN	10.09%	4.66%
DES	.02%	.02%
OO	.91%	.64%
OA	.09%	.02%
N/A	60.89%	46.27%
Number of comments	19675	4208
Number of revisions	27574	5149
Avg No. of Sentences per Commit	0.71	0.82
Avg No. of Words per Comment	10.71	6.91
VCS	SVN	Git

4 Mining software engineering projects from unstructured data

Projects are common work-plans that companies use to achieve their business goals. A project consists of an endeavor that involves several phases and disciplines, and is undertaken with time and resource constraints. Computer engineering firms typically deal with software projects, in which the resources are programmers and the final product is a software. In order to ensure quality and in-time delivery for the software product, companies break down the project into phases and its related tasks into work packages. After the work packages have been defined and the resources have been assigned, the project should be implemented according to the predefined plan.

There are a number of aspects that project managers may want to know about their projects in order to assess their quality. For instance, managers may want to know whether the project has been carried out according to the plan. This is particularly useful when the project must account for safety-critical activities, e.g. installation of a railway interlocking-system. Because these projects must also comply with safety-rules and regulations, their phases must be clearly documented in order to provide evidence of compliance. Nevertheless, work documentation in large project is sparse among several technical reports, software code comments, emails, and other artifacts that are produced by specific tools.

This section aims at analyzing the user comments in software repositories during the implementation phase of the project. The goal is to verify whether the project phases are reflected by lexical elements in the comments made by users

when they check in parts of their tasks. Useful insights are gathered by comparing a corpus of open-source projects to the common English and to the a corpus taken from software engineering books (in particular we use the RUP corpus as it represent the basis for software engineering terminology). Further analysis is conducted by observing concordances and keywords that point to particular software phases.

4.1 Mining project phases

This section aims at discovering which Rational Unified Process (RUP) phases and related disciplines are reflected in software repositories. The RUP is an iterative software development process framework. Although many alternatives exist, RUP is still a reference when it comes to breaking down a software development project into phases and disciplines. Here we give an overview on these concepts and show how we can use linguistic tools to analyze these repositories.

4.1.1 The Rational Unified Process for Software Development

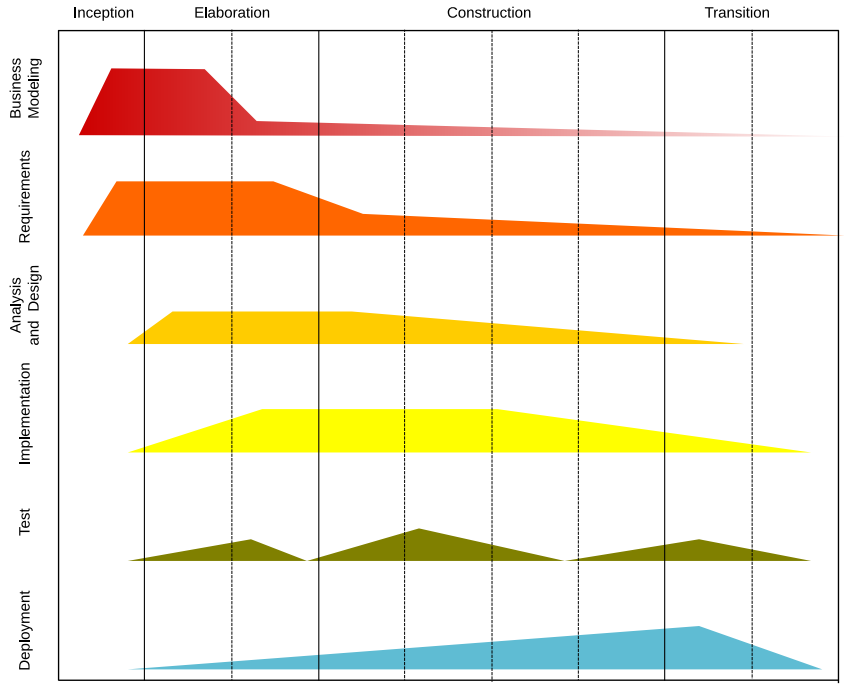


Figure 5 Example of the RUP software development model.

A simple RUP model is illustrated in Figure 5. The horizontal axis represents

the time from when the project incepted to when it is over, that is, the final product is delivered to the customer. The vertical axis divides the disciplines that are employed during each phase. Disciplines and phases may be divided in a more fine-grained fashion, however, the main phases are typically four and disciplines are six. They are described as follows.

Inception. The requirements are collected and goals are specified;
Elaboration. The collected material is analyzed and targets are formally defined;
Construction. The software is implemented.
Transition. The software is migrated to the final execution environment.

During each phase one or more disciplines are employed. RUP includes the following disciplines.

Business modeling. The work-flow, resources and the strategy are defined.
Requirements. Requirements are identified and software features are planned.
Analysis and design. Challenges and software architecture are identified.
Implementation. The software components are written in a programming language.
Test. Software functionalities are tested.
Deployment. The software is shipped to the customer.

4.1.2 Analyzing the GitHub corpus

This work uses corpus linguistics methods to analyze user comments about changes in software artifacts. In particular, we use corpora comparison, keywords and collocations to understand whether it is possible to find references to the various project phases. We follow the corpus-driven approach [Tognini-Bonelli, 2002] to understand which words are actually used by software project members.

4.1.2.1 Method

The general method in this work falls into the corpus-driven approach. We use data and tools from the mining software repositories (MSR) field in order to extract a corpus. This corpus is then analyzed with standard methods from corpus linguistics (CL). In order to better understand the discovery of the project phases, this approach proceeds in four steps. First, we use the GHTorrent[5] from the work

[5] http://ghtorrent.org/

of Gousios [2013] to extract all the GitHub projects from 2012 to 2016. Second, we preprocess the data by selecting only the information about user comments. Third, we use the AntConc tool from Anthony [2014] and Leech et al. [2014] corpus to analyze the GitHub corpus from the research questions' point of view. Fourth, we manually inspect the styles of typical commits that reflect the RUP phases.

4.1.2.2 Getting the corpora

GitHub has been widely studied by researchers in the MSR community. The work of [Gousios, 2013] provides a tool, namely GHTorrent, which has been continuously gathering GitHub events since 2012. The data is available for download and it comes as a set of tables from a relational database. After the download, such schema can be recreated on the local machine and the data stored accordingly. We used the MySQL database management system (DMBS) to restore the downloaded dumps. Using the SQL language, it was possible to query for and filter out all the user comments of all the available projects. The result was a one-column table with as many rows as the total number of commits. Through a simple JAVA program, the content of each row was appended into a text file. This file is the resulting GitHub corpus.

The RUP corpus was obtained from books and guidelines on the topic. Altogether eight fundamental books have been taken as a base. Each book dedicates one chapter to each of the RUP disciplines. The content of the relevant chapter was therefore pasted into a text file. The final result is a set of files that can be grouped by topic. This set of files constitutes the working RUP corpus.

4.1.2.3 Data processing.

With the corpora available as texts files, we used AntConc [Anthony, 2014] corpus software to analyze the data. The corpus methods are described as follows.

M.1: GitHub corpus vs. BNC written. In order to understand the comments of the project participants, we first analyzed their vocabulary with respect to the common English language. As a reference corpus for common English, we used the BNC written corpus [Leech et al., 2014]. This choice is justified by the fact that GitHub's corpus is extracted from written user comments.

M.2: GitHub corpus vs. RUP corpus. The aim of this work includes understanding how different types of work are reflected by user text. The types of work are described by the six RUP disciplines. Thus, a comparison between the GitHub

and the RUP corpora is required. This allows identifying whether the words that define the disciplines are actually used in real world projects.

M.3: Corpus analysis. Once the GitHub corpus was characterized with respect to both the common English and the RUP corpora, we further analyzed the corpus itself for collocations and frequently occurring patterns. This analysis further hints on possible specialized words and terms that are used together with the RUP keywords.

4.1.3 Results

This section presents the results and discusses the findings from the applying the aforementioned corpus methods. The results will be presented in short tables. The reader is referenced to the appendices for more detailed results and tables with up to one hundred rows.

4.1.3.1 Corpora comparison

	BNC written		GitHub	
rank	freq	word	freq	word
1	5529513	the	1936305	the
2	2820005	*of*	1285929	to
3	2321161	to	1212853	**i**
4	2316623	and*	1064976	a
5	1942423	a	916708	it
6	1772037	in	890624	**this**
7	877366	*that*	821212	is
8	858945	is	634698	in
9	795808	*for*	594955	**you**
10	788928	it	578323	and*

Table 5 Top 10 most frequent words in GitHub and BNC (common English) written

M.1: GitHub corpus vs BNC written. The BNC corpus consists of around 85 million tokens, clustered into approximately 300 thousand word types. The GitHub corpus consists of around 56 million tokens, clustered into around 1 million word types. This translates into a type-token ratio of 0,0039 (four per thousand) for BNC written and a type-token ratio if 0,018 (around two percent) for GitHub. This difference in the order of magnitude between the two corpora shows that there is actually a substantial variation in writing style of software workers. In fact, it is often the case that comments in GitHub are simply composed of one

or two words, e.g., "fix bug", in contrast to longer sentences used in the BNC. Table 5 list the ten most frequent words in the GitHub and in the BNC written corpora. The words appearing in italic (i.e., *of*, *that*, and *for*) occur only in the BNC written corpus. Conversely, the word marked in bold (i.e., **i**, **this**, **you**) occur only in the top ten list of GitHub. It is also interesting to see the rank distance of the conjunction word "and". While it ranks 4th in common texts, it ranks only 10th in GitHub (with a frequency almost 4 times lower with respect to the most frequent word). This, together with the absence of certain words that serve as a connector between clauses, e.g. *that*, may be indicators that GitHub is mainly composed of short sentences. Putting it all together, this first comparison hints that, differently from common English, software developers' text contain mostly short phrases that use a higher number lexical words. As a last observation, in line with the words "i","you" and "this" point to a less formal communication between project members.

	GitHub		RUP	
rank	freq	word	freq	word
1	1936305	the	8400	the
2	1285929	to	3944	**of**
3	1212853	*i*	3658	a
4	1064976	a	3305	to
5	916708	*it*	3049	and*
6	890624	*this*	2511	is
7	821212	is	2462	in
8	634698	in	1598	**that**
9	594955	*you*	1267	**are**
10	578323	and*	1127	**as**

Table 6 Top 10 most frequent words in GitHub and RUP corpora

M.2: GitHub corpus vs RUP corpus. The RUP corpus consists of 130 thousand tokens and around 8 thousand word types. This corpus contains the study material for engineers who learn the software development process model. Therefore, comparing this corpus to the GitHub one, we can point out differences on how projects are supposed to be carried out and how they are actually done. Table 6 helps to see these differences more clearly[6]. Here, we find words like the ones

[6] Note that the single letters appearing in the table are the results of a tokenizer that break the quote (') character. This is informative as it hints at the type of discourse. For instance in a non formal genre words like *haven'**t***, *don'**t***, *isn'**t***, etc., appear more often than in formal writing.

marked in bold, which give clues of a more formal type of discourse. Again, there is a notable difference in the frequency of "and", hinting to a higher similarity of the RUP corpus to common English than to the GitHub corpus. Another interesting fact is that the words that are present in GitHub but not in the RUP corpus are almost the same, with Table 6 having in addition the word "in". This reinforces the clue about the genre, with GitHub being more informal.

4.1.3.2 Corpus inspection

This section reports the results of step **M.3: Corpus analysis**. In particular, we have inspected the keywords and the collocations of symbolic words that may represent project phases and disciplines.

Rank	Freq	Keyness	Word
1	1212853	4745.013	i
2	238349	1106.858	https
3	235398	1093.154	github
4	256178	1064.277	http
5	419625	1022.381	t
6	916708	938.423	it
7	250139	889.463	com
8	236759	865.046	d
9	173664	761.574	org
10	163970	747.571	commit

Table 7 GitHubs's top ten keywords using RUP as reference corpus

Keywords analysis. The goal is here to understand whether significant words emerge that may refer to project phases or disciplines. Given that GitHub is mostly used by software engineer and programmers, knowledge on the standard RUP model is expected. The hypothesis is that we do not find information about the first two disciplines of the RUP, namely Business Modeling and Requirement Analysis, but we find evidence of the Analysis and Design, Implementation, Test, and Deployment phases.

Using the AntConc software, we generated three keyword lists. First, we took the BNC corpus as the reference (see Table 14 in Appendix A). As expected, words like *design*, *requirements*, *software*, *class*, *analysis*, *system*, *modeling*, *test*, *code*, *implementation*, etc, were unusually frequent in the RUP corpus with respect to common English. Second, we generated the keyword list from the GitHub corpus, still using BNC as the reference (see Table 15 in Appendix A). Apart from the unusually high frequency of the words *i*, *http*, *this*, *com*, *https*, *github*, *org*, which are clearly

common parts of the set of the project URLs, the words *build*, *code*, *tests*, *file*, *api*, *test*, *add*, *use*, *fix*, occurred in the top 50 keywords. These words are related to project disciplines. For instance, the words *test*, and *fix* are related to the *Implementation* discipline.

By comparing the two above mentioned keyword-lists, we found only partial support to our hypothesis. In particular, not only there is no reference to the first two RUP disciplines (Business Modeling and Requirement Analysis), but also reference to Deployment is missing in the first one hundred. The first occurrence of *deploy* is ranked in position 950. Moreover, the Analysis and Design is not reflected by any keyword in the top 100. The first word related to this phase is the word *class* (from *design of classes*, meaning that programmers are creating objects to represent concepts). This word is ranked at position 172.

As a further validation step we created a keyword list for GitHub, using the RUP corpus as a reference. This is reported in Table 7. An extended list can be found in Appendix A in Table 16. A part from the words indicating parts of a URL, the word *commit* occurs in the top ten list. This is no surprise as this word specifies the use of a VCS. By looking at the extended table, it is possible to notice two things. First, a big number of words pointing to a informal genre (e.g., *I*, *think*, *ok*, *me*) appear. Second, words that were highly ranked as keywords with respect to BNC are still highly ranked when RUP is taken as a reference. This hints at the fact that GitHub users focus extensively on particular parts of the RUP software development model.

Summing up, by inspecting the keywords, the following conclusions can be drawn. First, not all the disciplines are sufficiently represented in GitHub. Keywords suggest that the GitHub comments are mostly about the Implementation and the Testing phase, while the Analysis and Design is present but not frequent. The other phases are instead not directly represented in the corpus, with the Deployment phase being surprisingly low in the ranking. Looking further into the difference between the GitHub and the RUP corpora, we can observe that users most likely use GitHub for Implementation and Testing of software projects, as particular keywords like *build* and *check* continue to be highly ranked when using RUP as a reference.

Collocations and patterns. The keyword analysis hints at the fact that GitHub is used mainly for Implementation and Testing, and Design, although the latter is less frequent. Here we delve into the details on how users write when they talk about the most represented RUP disciplines. To this end we used the collocation method from AntConc, to inspect how project members use the words *design*,

implementation and *test* in context. Results are reported in Appendix B in three tables. The tables are sorted by the keyword occurrence frequency. From the tables and from further inspection (using also other methods like clustering), it appears that the words do refer to the same concepts as in the RUP. Design is often used together with the word *pattern*. Design patterns are well-known engineering best practices. Implement appears mostly as a verb. However, a term referring to a software artifact (e.g. implement functionalities) is almost always present in the context. Test is mostly used as a lexical word (in most of the cases as a noun). It also clearly refers to the testing discipline, e.g., test output, test case, etc. An idiomatic usage of the inspected words is almost never noticed in the corpus.

4.2 Story mining from software repositories

Story Mining aims at extracting business process activities from natural language text Gonçalves et al. [2011]. The approach of story mining is shown in Figure 6 and comprises three main steps. First, it requires that the process participants tell their stories, which consist of their vision of the process. This has to be done on a daily basis. Second, once the stories have been collected for all the participants, the mining phase starts. The mining phase includes a first step of preprocessing of the natural language text using classic natural language processing (NLP) techniques in a pipeline, such as tokenization, stop word removal, lemmatization, etc. Third, it proposes a set of possible processes and choose the most appropriate on as the result.

The method used in story mining would generate models for every textual input given. Hence, we can use story mining also for the user comments in a VCS. To do this, we used the data model presented in the previous version of this deliverable Cabanillas et al. [2016b], to import processes from the real world into a database. We then query the comments for each user, and store them into separate files. These files represent the user stories to be fed to the story mining algorithm.

The next step is to cluster similar activities that emerge from the output of Story mining. To do this we used semantic textual similarity (STS), drew from the SemEval workshop series[7]. We chose the DLS@CU Sultan et al. [2015] that performed best in the workshop. The similarity measure is based on how much parts from two sentences can be aligned by looking them up in a so-called paraphrase database. The method is explained in Ganitkevitch et al. [2013] and is based on the intuition that two groups of words have the same meaning if they can be trans-

[7] https://en.wikipedia.org/wiki/SemEval

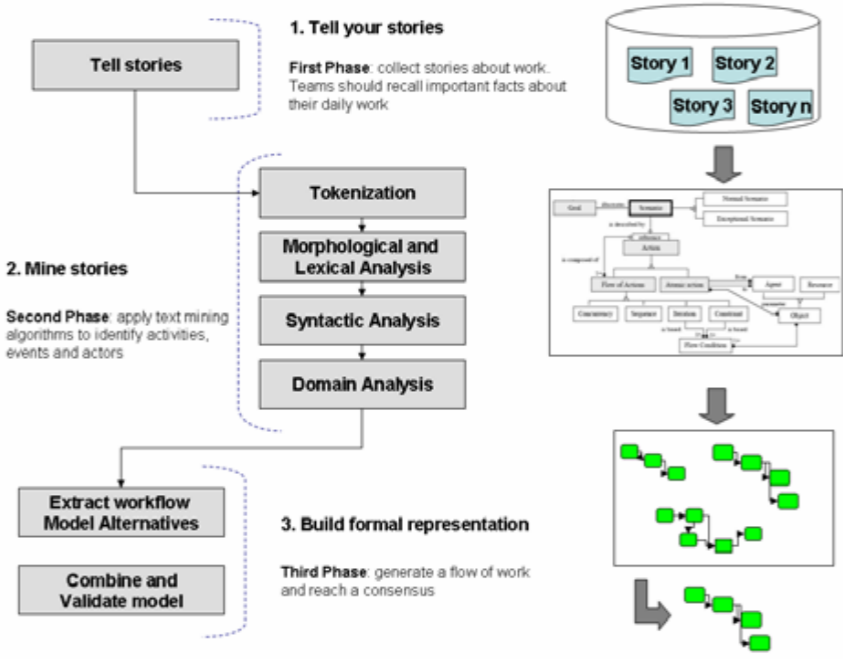


Figure 6 The story mining approach to extract business process activities from stakeholders' stories Gonçalves et al. [2011]

lated in the same set of words in a foreign language. For example *thrown into jail* and *imprisoned* are both translated into the German word *festgenommen*. This means that they have a high similarity.

The work is ongoing and we expect to improve the identification of activities from project-mining approach described in Bala et al. [2015] by identifying another view on the activities. Discovering textual activities could provide a step forward towards labeling the activities of the Gantt chart extracted by Bala et al. [2015] with semantics.

5 Combined methods for mining business processes

This section presents a method and tool to handle large process logs by storing them into a relational database and allowing for the execution of process mining algorithms. This has been developed as part of a Master Thesis Fischbach [2016]. More specifically, we designed an approach to create a common basis for imperative and declarative process mining on relational databases. This approach is based on the following four steps:

1. A database is created according to the RXES schema.
2. The database is populated with event log information.

3. The event log information is used to calculate Support and Confidence metrics for several declarative constraints in the DECLARE language Pesic et al. [2007] using SQL Miner queries. The results are stored in the database.
4. Using the information we obtain by mining the declarative constraints from process logs, we are able to create Figure 7. It shows the ordering relations for that we can use as a basis for mining imperative models out of DECLARE constraints.

Ordering Relation	**Constraints / Relations needed for Derivation**
All Algorithms/Miners	
Direct Successor ($>$)	Activities where the $ChainResponse(A, B)$ constraint has a Support of greater than zero
Exclusiveness ($\#$)	$>$ - relation
Alpha Algorithm	
Causality ($\rightarrow$)	$>$ - relation
Concurrency ($\|\|$)	$>$ - relation
Alpha+ Algorithm	
Length-2-Loop ($\triangle$)	Activities where the custom $Length-2-Loop(A, B)$ constraint has a Support of greater than zero
Symmetric Length-2-Loop ($\diamond$)	$\triangle$ - relation
Causality ($\rightarrow$)	$>$ - relation, $\diamond$ - relation
Concurrency ($\|\|$)	$>$ - relation, $\diamond$ - relation
Alpha++ Algorithm	
Length-2-Loop ($\triangle$)	Activities where the custom $Length-2-Loop(A, B)$ constraint has a Support of greater than zero
Causality ($\rightarrow$)	$>$ - relation, $\triangle$ - relation
Concurrency ($\|\|$)	$>$ - relation, $\triangle$ -relation
XOR-Split ($\triangleleft$)	$\#$ - relation, $\rightarrow$ - relation
XOR-Join ($\triangleright$)	$\#$ - relation, $\rightarrow$ - relation
Indirect Successor ($\gg$)	$>$ - relation, $\triangleleft$ - relation, $\triangleright$ - relation; many "sub process executions" have to be analyzed
Succession ($\succ$)	$>$ - relation, $\gg$ - relation
Heuristics Miner	
Causality ($\rightarrow$)	$>$ - relation
Concurrency ($\|\|$)	$>$ - relation
Length-2-Loop ($>>$)	Activities where the custom $Length-2-Loop(A, B)$ constraint has a Support of greater than zero
Long-Distance-Dependency ($>>>$)	Activities where the $Response(A, B)$ constraint has a Support of greater than zero

Figure 7 Mapping between order relations and declare constraints Fischbach [2016]

The result of this approach is a database that contains event log information stored according to the RXES schema and declarative constraints including Sup-

port and Confidence values calculated from the event log information that can be used to derive ordering relations used by several imperative process mining algorithms. Those ordering relations can then be used as an input for adapted algorithms to create imperative process models. Therefore, it can be concluded that the resulting database is an infrastructure for different forms of process mining because it is possible to extract necessary information for different process mining algorithms from it.

We implemented this approach in a Java application called RXESuite. With this application it is possible to create and query such an infrastructure for process mining on relational databases. RXESuite contains features for storing event log data in relational databases according to the RXES schema and extracting declarative constraints as well as imperative ordering relations from this data. Specifically, with RXESuite it is possible to:

- Import XES files (using a special library called OpenXES [10], described in Section 4.3) into a relational database according to the RXES schema.
- Derive specific declarative constraints from the event logs stored in the database using (adapted) SQL Miner queries and display them in a convenient way.
- Derive imperative ordering relations for imperative miners using the declarative constraints as a basis.
- Export the derived constraints and/or ordering relations to a structured CSV-file that can be used as an input for other tools (e.g. special mining tools that use the created file as an input).

A summary of a possible usage procedure of RXESuite is presented in Figure 8. The source code of RXESuite is publicly available in an online repository[8]. All required libraries are already included in that repository. To run the application it is necessary to compile it using the Java SE Development Kit 8 for the respective underlying platform.

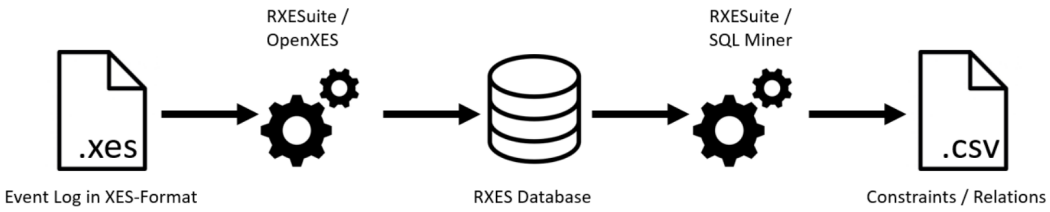


Figure 8 Summary of the Derivation Procedure using RXESuite Fischbach [2016]

[8] Accessible at https://bitbucket.org/FroZzy18/rxesuite

We use SQL Miner queries to derive the declarative constraints because the approach already provides templates for mining DECLARE constraints. At the point where the SQL Miner comes into play, the event log is already stored in a relational database according to the RXES schema that can be queried with SQL. Therefore, we just adapted the existing SQL Miner queries of Schönig et al. [2015] to derive the constraints. Although in principle it would be possible to use any declarative mining approach, these approaches would require the extraction of the event log data from the relational database as a first step, only to be able to extract the final constraints in a second step. With SQL Miner it is possible to directly derive the constraints from the relational database and return the resulting constraints.

We decided to include the feature of deriving the Indirect Successor[9] relation of the Alpha++ algorithm with RXESuite, although the procedure is more processing-intensive than the derivation of the other relations. However, we want to prove that it is actually possible to derive the Indirect Successor relation with the underlying RXES structure and neglect therefore performance issues. In general, the tool is mainly designed to act as a proof-of-concept. Therefore, the focus of this work was to create a functional infrastructure that is capable of achieving the points presented above. Additionally, we designed RXESuite as flexible as possible so that it can be easily adapted and reused in future projects.

6 Combined methods for mining user roles from VCS logs

This sections shows an approach to mine resource-roles combining information from both text and quantitative data from Version control system (VCS).

6.1 Background

Here we discuss the problem and related work.

6.1.1 Problem description

The problem we address in this section is the discovery of the roles that members of a software project may actually play in a collaborative setting. Each member plays a particular role in the project. Roles are decided in the project planning

[9] The Indirect Successor relation is defined as the existence of two events a and b such that for all the traces of a log, b does not immediately follow a, but it follows a eventually

phase. This phase also involves the definition of project tasks and the assignment of suitable tasks to the various roles.

Projects follow clear guidelines. Guidelines may come from internal policies or from rules and regulations of the working domain. For example, software systems in the railway domain must make sure that their development process and resources comply to safety requirements imposed by the European standard EN50128. This is why project managers need to track how they distribute work to different people and whether the project members effectively contribute according to their role and their task.

Software configuration management (SCM) systems are a valuable source of information to investigate on the behavior of project members. There systems are used for tracking and controlling changes in the software. If a change produces a wrong or undesired outcome, it is always possible to revert to an older configuration of the system. Artifacts' versions are automatically managed by VCSs. It is always possible to follow the evolution of each artifact, along with information about the resources who changed it and their comments, by looking into the VCS logs.

Figure 9 illustrates how people work in a project. They are assigned to defined tasks to which they contribute. Their contributions are part of generated artifacts. As time goes by, the number and the content of the artifacts on a project grows. Changes in the artifacts themselves and in the structure of the project reflect the development process that project workers follow. The evolution of the repository goes along with the progress of the project.

VCS logs provide rich and fine grained information about the changes in the project. A change may consist of a file being modified or new files being added in or removed from the repository. When changes are complete, it is possible to store them in the repository through a commit. Each commit contains a unique revision number, the identity of the resource who issued the commit, a timestamp, statistical information about the changes for each affected file, and a comment from the person who committed.

Let us now see an example of how people use a VCS to collaborate in a project. Bob is a software engineer at Abc. ltd. He is working on the collaborative 'Project 1' with his colleagues. He adds a new module to 'demo' file, updates the file 'rule' and leaves respective comment. After performing the commit, the VCS log entry for this instance would look like the first log entry in the table with log ID 'abc123'. Alice is the UX designer working on the same project. Her job is to look after the user satisfaction from a design perspective. She makes changes in the 'setup' interface file and leaves the respective comment. After committing, the

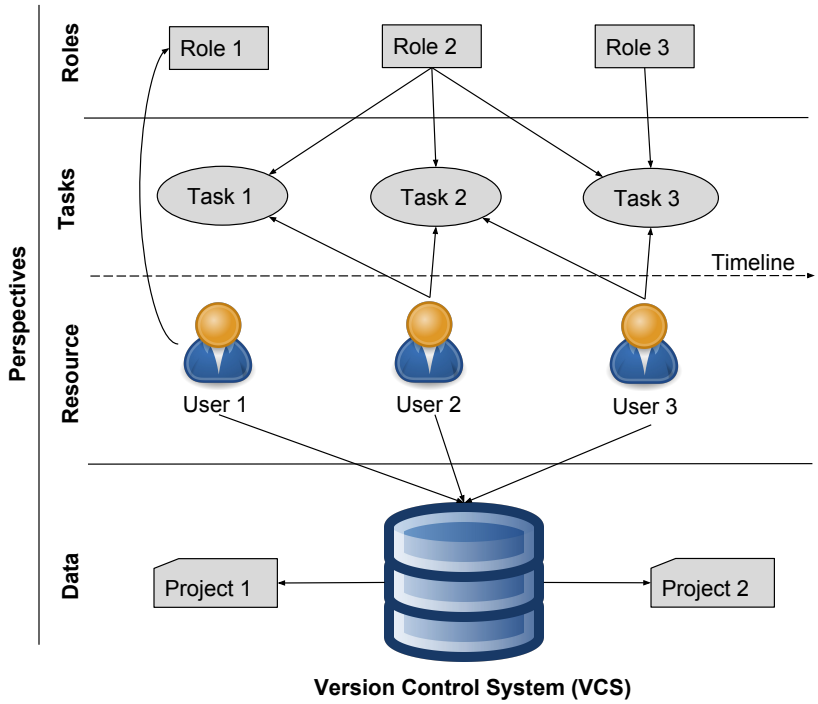


Figure 9 Software project and resources

VCS log for this entry is represented in table with log ID 'jsh567'. Consequently, Bob had to make changes in the 'setup' programming module represented by the log ID 'aof082'. Some important insights into the VCS logs are: *a)* log ID is a unique id generated by VCS and changes each time a commit is performed by the user; *b)* user ID in VCS is universal and remains the same; *c)* comments, though useful for classification of users, are optional and committers might avoid it or write just a single word. We can clearly see here how VCS gives us granular information which can help us classify the users. However, it should be kept in mind that this is a very simple example of a VCS, and as the projects grow both in size and numbers VCS can get very complex.

Table 8 illustrates our running example. A unique log ID for each commit performed by the user. Furthermore we can see the information on which repository the commit was performed, at what time and date, and which files were subsequently updated. User comments are dependant on many variables like personality, requirements etc and therefore sometimes they can be blank or contain all sorts of irrelevant information user might mention.

Table 8 Example of a VCS log

Log ID	User ID	Repository	TimeStamp	Updated Files	Comment
abc123	bob14	Project 1	2014-10-12 13:29:09	Demo.jar rule.txt	Added new module to demo and updated rules
jsh567	alice01	Project 1	2014-11-01 18:16:52	Setup.exe	Modified the setup interface
547	kriss	Project 2	2015-06-14 09:13:14	Todo.doc	Update the application interface
anfn876	s_tony	Project 3	2015-07-12 15:05:43	graph.svg todo.doc	Define initial process diagram & listed remaining tasks
aof082	bob14	Project 1	2014-11-05 10:12:47	setup.exe	Updated

6.1.2 Related Work

Role discovery has been addressed by literature in different settings and from several points of view. Here we classify existing efforts from a data perspective.

6.1.2.1 Structured data approaches

This class of methods includes algorithms that make use of quantifiable data. We divide them into: *a)* MSR approaches; and *b)* process mining (PM) approaches.

Mining software repositories approaches. In the area of MSR, Yu and Ramaswamy [2007] use a hierarchical clustering based on user interactions to identify two categories of users: *core member* and *associate member*. Core members are those users whose interaction frequency is higher than a given threshold. Associate members are instead users whose interaction frequency is below the threshold. Alonso et al. [2008] use a rule-based classifier that maps file types onto categories and hence each author who modified a file is linked to the files' category. Gousios et al. [2008] classify developers contribution based on lines of code (LOC) changes and map infer activities from them. Begel et al. [2010] developed the Codeboook software tool a utility for fining experts. They use a social network approach that combines sources from people, artifacts, and textual allusions to other people. Ying and Robillard [2014] study developer profiles in terms of their interaction with the software artifacts to understand how they modify files and to further recommend changes based on history from VCS logs. Füller et al. [2014] investigate user roles in innovation-contest communities. They use quantitative methods to analyze user activity logs and interpretative to categorize qualitative comments into classes.

Process mining approaches. Efforts have been done to analyze software repositories with process mining techniques. Rubin et al. [2007] implement a multi-perspective incremental mining that is able to continuously integrate sources of evidence and improve the software engineering process as the user interacts with the documents in the repository. Their approach allows for mining other perspective, such as roles, by applying social network analysis. However, only statistical methods can be applied to their output, since it lacks the comments that are associated to file changes. In the same setting, Poncin et al. [2011] developed FRASR, a framework for analyzing software repositories. FRASR can be used in order to transform VCS logs data into the XES (Verbeek et al. [2010]) data format that can

be further analyzed with process mining tools like ProM[10]. Song and van der Aalst [2008] focus on three types of organizational mining *i)* organizational model mining, *ii)* social network analysis, and *iii)* information flows between organizational entities. Schönig et al. [2015a] propose a mining technique to discover resource-aware declarative processes.

6.1.2.2 Unstructured data

Maalej and Happel [2010] use NLP for automating descriptions of work sessions by analyzing developers' informal text notes about their tasks. Developers are then classified into two classes based on their behavior: developers who use problem information to refer to their current activity and developers who refer to task and requirements. Kouters et al. [2012] developed an identity merging algorithm based on Latent Semantic Analysis (LSA) to disambiguate user emails. Licorish and MacDonell [2014] mined developer comments to understand their attitudes.

6.1.2.3 Other related work

The term *role mining* often points to role mining algorithms based on role-based access control (RBAC) systems. These algorithms takes as input predefined roles that are given as a matrix, where each users are assigned to access permissions. A number of algorithms have been developed to mine roles from RBAC systems alone (Lu et al. [2015], Frank et al. [2013]) or combining their data with process history logs (Baumgrass et al. [2012]). A survey of existing techniques and algorithms can be found in Mitra et al. [2016]. Our work is disjoint from this class of algorithms as VCS does not contain access control information. Bhattacharya et al. [2014] propose a contributor graph-based model. By constructing both a source-based profile and a bug-based profile, they are able to identify seven roles: *patch tester*, *assist*, *triager*, *bug analyst*, *core developer*, *bug fixer*, and *patch-quality improver*. Hoda et al. [2013] use a grounded theory (GT) approach to study agile teams. Their work unfolds the roles of *mentor*, *coordinator*, *translator*, *champion*, *promoter*, and *terminator*.

This work builds upon existing literature in that it gather insights on organizational level like in Rubin et al. [2007] and Song and van der Aalst [2008] but it takes into account unstructured data. Differently from the literature that works with unstructured data, we explicitly consider the problem of role discovery, i.e.

[10] http://www.promtools.org/doku.php

attribute roles to resources. Lastly, this approach differs from Bhattacharya et al. [2014] and Hoda et al. [2013] since we further adopt NLP techniques.

6.1.3 Research Questions

As mentioned in the previous section, the focus of this work lies on mining and analyzing properties of the users of VCS. The users belong to certain classes and these classes have different commit message styles. Based on this assumption, the following research questions are defined:

RQ1 *What classes can be assigned to users of VCS?* The first research question scrutinizes if a classification of the users of a VCS is possible only based on the information provided by the log files. The classification separates the users into clusters. To this extent the most expressive features and combinations of these features have to be identified. These clusters are then further analyzed by means of the next research question.

RQ2 *How can we map users types to classes?* Based on the created clusters meaningful classes are derived. The chosen features influence the types of classes that can be created and there has to be an intelligible distinction between those classes. This research question also answers how detailed this distinction can become and it creates behavioral profiles for the class members, based on the analyzed features.

RQ3 *What are the main differences found for these classes?* The last research question examines the differences between the created classes. It compares the results of the log files and identifies the reasons for dissimilarities. Therefore, it evaluates the quality of the classification and its applicability for different version control systems.

6.2 Approach

Commit messages contain metadata and content. For analysis of users and roles both parts are important. Metadata includes the author, which is essential for gaining a unique identifier of the committer. The content gives some indications on roles. Even if data is fetched already in the right format, it is still not ready to be analyzed. Information cleaning is as important as the extraction. Changed roles, one time committers, or several identities for the same use, are some of the challenges that need to be handled.

Once data is prepared, the analysis can start with a direct approach of connecting one commitment message to one user and his/her corresponding role. While

this 1:1 relationship is not always available, there are other possible analyses like the commit based approach. It is an indirect approach because in it a list of roles to every commit message. By linking the lists of roles to the authors the role of the user is predicted.

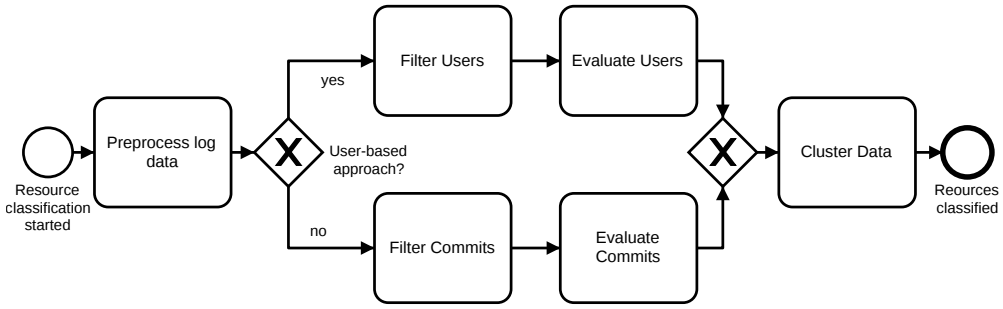


Figure 10 Approach to role classification from VCS logs

Figure 10 illustrates the steps of our approach through a BPMN diagram. In the first step we preprocess the data and parse the SVN log file. Then we account for two different types of classification: user-based and commit-based. These approaches require a prior step where we filter the data according to user or to commits. Both the approaches include an evaluation step where we map keywords and information on file types to users or commits, respectively. The last step is the data classification. Section 6.3 describes both the approaches in detail.

6.3 Implementation and evaluation

One solution for the outlined problem is an algorithmic approach in form of building a script using Python. This script automatically fetches, processes and analyzes the log files and creates a classification model that is based on the extracted information. The following tools are used for implementation:

Technology. Python is a general-purpose, high-level programming language. It is open source, easy to use and offers various third party modules[11].

Machine Learning. The Scikit Learn module is utilized for machine learning. It is built on matplotlib, numpy and scipy. It offers algorithms and graphical representations for classification, regression, clustering, dimensionality reduction, model selection and preprocessing[12]. We used decision treess (DTs) for classification and regression. DTs are supervised learning method whose goal

[11] https://www.python.org/
[12] http://scikit-learn.org/stable/

is to create a model that predicts of a target feature of variable by inferring existing rules from the data.

Natural Language Processing. For natural language processing the Natural Language Toolkit (NTLK) offers methods to extract additional information out of everyday communication. The "Bag of words" technique analyzes word occurrences, but also more sophisticated methods, which develop a human understanding of communicated messages, can be utilized [13].

After choosing the tools, a very important step required prior to the actual implementation is the selection of data. While the number of accessible code repositories available via repository hosting services is quite huge, picking a dataset of appropriate size and quality proves rather challenging. There have to be some considerations to be made for choosing the right repositories:

- Number of repositories: Classification on a single repository might not produce a generally applicable result, whereas using more repositories increases complexity.
- Size of individual repositories
- Role information: This is important for certain steps in the classification task and for verification of results.
- Differences in organisational aspects of the projects: The type of the development team (professional or hobby) type of software (proprietary or open-source) and type of platform (private server or public repository hosting service) might have influences on the way repositories are used.
- Differences in version control systems

For this project three repositories are analyzed, each with multiple thousand commits:

1. Main code repository of the company Infinica. Proprietary software. VCS: Mercurial
2. ProM Sourceforge project repository. Open source software. VCS: SVN
3. Camunda GitHub project repository. Open source software. VCS: Git

These three are chosen to cover a broad range of the above mentioned differences. The necessary role information was reviewed by interviews with the main contributors for these projects. While two repositories were publically available, the third was granted a direct access from within the company.

[13] http://www.nltk.org/book/ch00.html

First of all, browsing through the commit messages created an understanding of the structure of the log files, the type of information available for extraction and the way VCS commits are typically used in the different projects. They contain the most useful information but are also the least predictable factor, because there are no general rules on how to formulate such a message. From the messages certain words and phrases can be extracted, which can be linked to a specific class. In a next step, the log files were preprocessed removing commits and/or users with faulty information, merging users with several accounts and anonymizing the data if required.

The first step of creating the algorithm, once the the log files were obtained and their basic structure was known, is to transform them into a common in-memory object model. Three different functions, providing the same output for different input formats, were required to cover all three systems (Git, Svn, Mercurial). This model consists of one object for each commit in the log, consisting of an id, an author, a message, a timestamp and lists of all added, modified and deleted files. From this it is possible to derive an additional model consisting of the users, by aggregating the information for each author name occurring in the commits.

For further analysis and classification steps, efforts have been split up into two approaches. The first one focuses purely on the users and the features deductible for them. We soon realized though that this method, while providing some promising insights, had certain limitations and left out some potentially valuable information. Therefore a second approach was incorporated taking a closer look at the individual commits. The two approaches are outlined in more detail in the following sections.

6.3.1 User-based Approach

The main idea of this approach is to use clustering in order to find potential user classes and build a classification model based on those clusters, which represent a comprehensible, existing class of users. As a clustering method we used k-means.

The first step is finding and calculating features for the users where useful differences between classes might be occurring. An explorative approach is required to find the meaningful ones. The calculated and analyzed features are:

- Total number of commits
- Timeframe: The time between the first and last commit of the user (approximates the time a user has been working on a project)
- Commit frequency: Total number of commits divided by the time frame. Represents the number of commits a user makes within a certain period of time

(e.g. day, month)

- Commits message length: Average length of commit messages in number of words or characters
- Occurrence numbers of certain keywords: How often a certain word (e.g. "test", "fixed", etc) is used by a user, relative to the total number of commits.
- Number of added/modified/deleted files
- Occurrence number of file formats: How often a file with a certain format (e.g. .java, .py, .html...) are modified by a user, relative to the total number of modified files.

In order to find useful clusters, an experimentation with different combinations of features is necessary. For each of these combinations the optimal cluster number has to be found. While the clustering and optimizing tasks can be done programmatically, the clusters have to be evaluated manually. This makes it impossible to test all possible combinations of features, especially when including the keyword occurrence numbers, as the list of keywords with potential value was far too large. Combinations that promise useful results are selected. Evaluation is done using plots as well as looking at the raw data to find connections between the identified clusters and the existing classes in the sample data.

From the clusters generated with this method classification models are built using the decision tree classification method. The decision tree models are trained for the three data sets individually. For verification of their quality, the models are cross validated with the other data sets respectively.

6.3.2 Commit-based Approach

There are multiple reasons leading to the decision of adopting a second approach in addition to the user clustering. As already mentioned above, the research led to the conclusion that in many cases a user can have multiple roles or executes many tasks not belonging to his primary role. This kind of use case is hard to cover using only the simple clustering method. Another reason is that a lot of information is lost when aggregating the commit information for users. This lead to the idea of classifying individual commits.

The algorithm iterates over commits and tries to assign types to them. The types are assigned based both on the analysis of the commit message, and the file extensions. The message is searched for certain keywords and phrases which are connected to types. The file extensions are searched for known file types fitting to a commit type. There are certain overlaps between commit types, for example the type addition can be a development or test commit. In those cases where one

identified type is a more specific description for another, only the more specific one is included into the further analysis. The identified commits and the related keywords and file types are listed in Table 9.

Table 9 Keywords used to classify the type of work

Test	Development	Web	Tool	Maintenance	Refactor	Documentation	Design
test	implement	web	tool	bugfix	refactor	documentation	style
tested	implemented	spring	library	bugfixes	refact	documented	styles
testing	implementation	http	libraries	fix	refactored	javadoc	styling
tests	implementing	https	framework	fixed	refactoring'	readme	icon
testcase	implements	rest	dependency	fixes	refactorings	userguide	icons
testcases	improved	html	dependencies	fixing	rename	user	font
test	improving	css	upgrade	patch	renamed	guide	layout
case	update	servlet	upgrade to	patched	renaming	tutorial	layouts
test cases	updated			cleanup	move	faq	layouted
cases	updating			cleanups	moved	translation	layouting
unittest	script			clean up	revert	translate	file types:
unit	scripting			clean	reverted	translated	png
test				cleaned	reverting	translating	svg
integrationtest				cleaning		doc	jpg
integration						i18n'	
test						file types	
fitnesse						txt, docx, text, tex, pdf	

Build	Data	Backend	Addition	Removal	vcsManagement	Automated	Merge
build	data	engine	added	delete	svn	(starts with)	(starts with)
building	database		add	deleted	git	Automated Release	Merge
compile	sql		adding	deleting	mercurial	Automated Nightly	merge
compiled	postgres		new	remove			#merge'
compiling	postgre		create	removed			
release	postgresql		created	removing			

After assigning the commit types, the commits can be aggregated for the individual users resulting in the absolute occurrence numbers of each type for each user. Dividing each of these values by the total number of commits of the user, we get percentages for each type. These percentages tell how the work of a user is distributed among the different kinds of tasks which appear in a VCS. These user profiles are a form of classification, which is not as simple and concise as assigning one definite class for each user, but has the advantage of covering secondary roles and minor tasks. They can be useful for analyzing smaller project teams with multiple roles for one user.

The classification task is done on user profiles. A table of user profiles is created and a role for each user manually inserted, based on the role information of the data sets. For this part the Infinica data set is used, because it is the one with the most extensive information base and it has the most diverse and comprehensive set of roles. Four roles are assigned to the Infinica users: Web developers, other developers, testers and support. The last one is an aggregation of users who have different oficial roles but contribute in the VCS mainly in form of minor, supportive tasks.

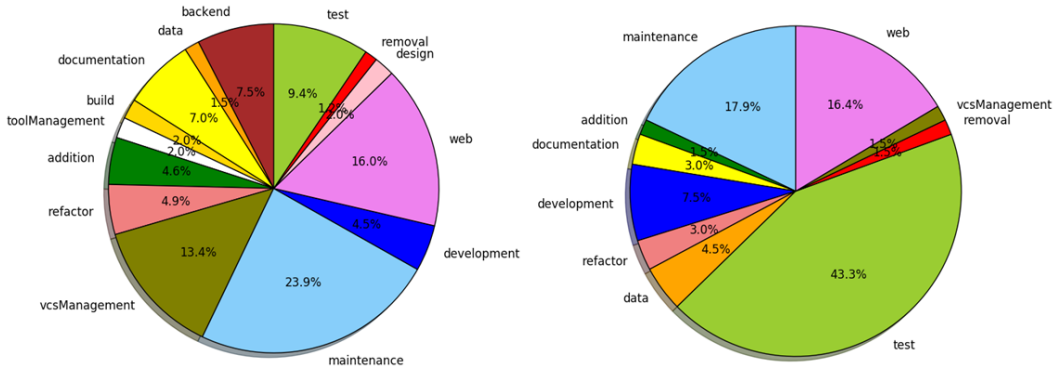
The classification task was done in two ways: 1. Manual analysis of the table and derivation of rules by looking for similarities between users with the same role. 2. Automated classification in line with our original concept. For the latter the decision tree method is used. In this case the commit type percentages are used as features and the manually assigned roles as classes. The resulting decision tree model was cross validated against the ProM and Camunda data sets.

The final algorithm including retrieval, processing and analysis of data as well as classification and verification, consists of roughly 700 lines of code. The results for the two approaches are discussed below.

6.3.3 Results

In the first step of the user-based approach, the most expressive features of the Infinica data set were identified. The solution was tested on the Infinica data set due to the immediate availability of detailed information about the log file. The best results were achieved with the combination of the commit frequency and the occurrence numbers of test related keywords.

The commit frequency is a strong indicator of developers. Moreover, it also differentiates between the working preferences of the developers. The group with the lower frequencies tends to work locally on their machines and pushes their work when it is finished. Whereas, the other developers group pushes every small



(a) User profile for a backend developer **(b)** User profile for a tester

Figure 11 Commit distributions of the Infinica dataset

change into the repository separately so that everybody is updated and works with the latest code.

Test-related keywords can not only identify testers, but also differentiate between their expertise. The sum of the words "test", "tested", "testing", "tests" is already enough to make a distinction between testers and developers. Additionally, manual testers use these words less often than their technical counterparts, who are more focused on automation.

The first part of the commit-based algorithm implementation, i.e., the commit classification, was successfully tested on all three data sets combined. For all three sets we achieved a similar coverage (percentage of commits which could be assigned some type). For Infinica the coverage was 88,94%, for Camunda it was 90,25% and for ProM 86,65%.

While the user profiles were originally intended as an intermediary step and a means to verify and improve the quality of our approach, they proved to be a quite useful perspective on user roles, beyond the simple categorisation using a single class. Especially when using a graphical representation such as the pie charts which can be seen in Figure 11a and Figure 11b, the commit distribution provides an interesting insight in the actual roles and tasks of users.

The Infinica dataset is divided successfully into expressive classes with k-means clustering with a k=4 shown in Figure 12. The developers (red and dark blue) are split off based on their higher frequency in the first step of the decision tree and the following classes are derived:

- **Developers - frequent committers:** The red cluster is comprised of developers which push every change to the repository.
- **Developers - heavy commits:** The dark blue cluster contains developers who work on their local machine and push less frequently. However, they have

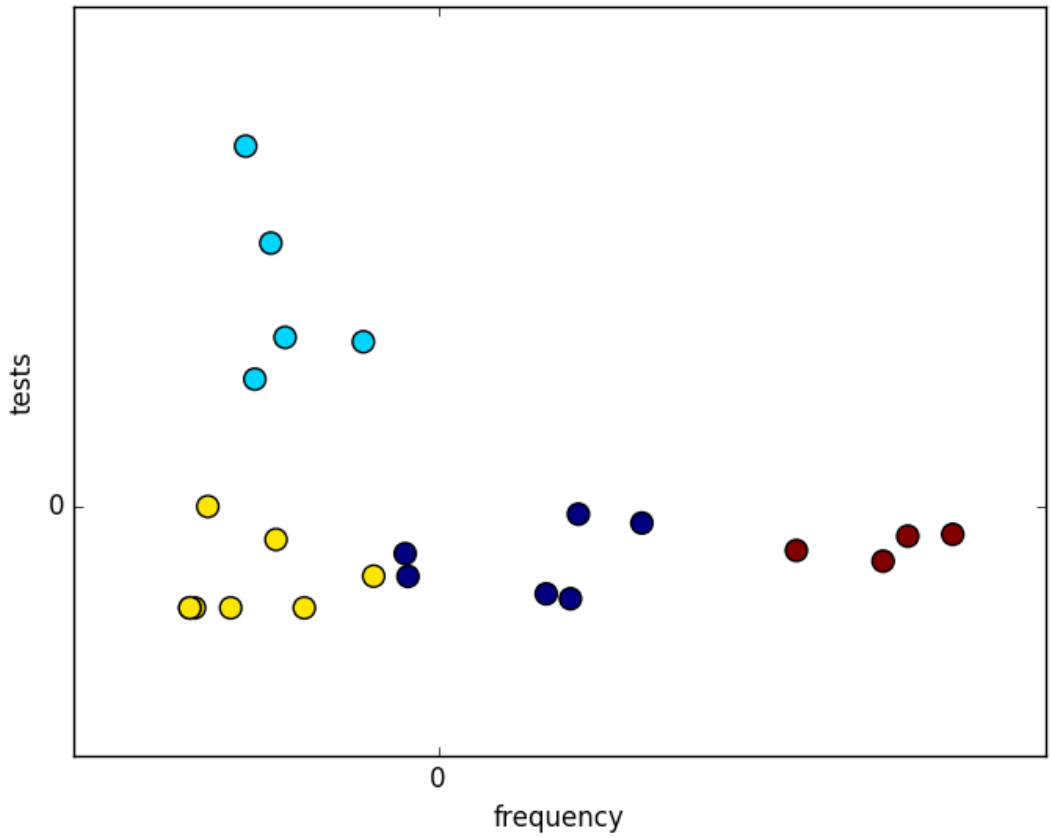


Figure 12 Scatterplot with colorcoded clusters (Infinica data set)

bigger commits containing more changes.

- **Testers - technical:** Testers focused on automating the testing process are in the light blue cluster. They are solely senior developers but not all of them are situated in this group.
- **Testers - non technical:** The yellow cluster is comprised of less technical testers which tend to do more manual testing. Additionally, it includes the special service team.

Table 10 shows an excerpt of the user profiles and role assignments for the Infinica data set. The commit types development, backend, maintenance and refactor have been merged to a single type development, as the other, more specific types provided no additional value in this case. Also the types addition, removal and merge have been left out here due to a lack of meaningfulness. The data visible in the table was used for the manual classification as well as the creation of the decision tree model.

We identified 4 role-based classes which were visible from the Infinica data. Those were testers, with more than 20% of their commits being of type test and

Table 10 Excerpt from user profiles with roles for Infinica data set

test %	development %	web %	documentation %	vcsManagement %	build %	toolManagement %	data %	design %	class name
6.07	39.49	26.40	7.94	0.23	3.27	0.70	0.23	11.21	dev
9.41	40.90	15.99	7.00	13.37	1.96	2.04	1.46	2.00	dev
3.83	26.78	44.70	2.90	4.70	3.93	0.60	2.95	6.01	webdev
0.00	28.07	52.28	2.46	0.00	5.26	1.40	0.35	8.42	webdev
43.28	28.36	16.42	2.99	1.49	0.00	0.00	4.48	0.00	tester
23.99	27.73	15.26	7.79	0.00	2.49	1.56	1.25	4.67	tester
0.00	7.94	38.10	0.00	38.10	1.59	4.76	0.00	9.52	support
1.19	3.39	46.15	1.61	46.15	0.08	0.17	1.19	0.08	support
...	...	...	...	...	...	...	...	...	...

between 30% and 50% of type development developers with above 50% development commits, web developers with more than 40% of type web and 20% of other development and non-technical users with less than 20% development. In addition to those, we found some less obvious hints for additional classes, represented by minor, secondary roles of users. These were not represented as actual existing roles in our data, so we could not verify our assumptions. The corresponding classes we suggest for those are technical writer with more than 40% documentation commits, designers with above 30% design commits, users with various administration tasks, represented by high numbers of the types build, vcsManagment and toolManagement and database experts with a large percentage of data commits.

When applying the optimal features and clusters deduced from the Infinica data set on the ProM data set, different but still meaningful user classes are derived. This is due to the fact, that the Infinica data set represents the development process within a company, whereas ProM is an academic project where researchers continuously join and leave. However, it still required to get invited for working on the project which creates an entry barrier. All members are developers without any designated testers. The ProM data set can be divided into the following four classes as shown in Figure 13.

- **Core developers:** The employed users of the ProM project are situated in the light blue cluster to the right. Their commit frequency and absolute number commits is far above the others. There is also a system user in this class. Its main task is building the project.
- **Engaged developers:** The dark blue cluster contains developers which also write tests. They are more committed and they put more effort into the development.
- **One-time developers:** The enagement of this group ends with the addition of their required functionality. The majority of the users falls into this class and it is represented by the red cluster at the bottom left.
- **Testers:** Despite the lack of testers in the ProM data set two have been identified as such.

For the ProM data set which consisted of 42 users, 35 (83%) were classified as developers. This is not surprising as we knew before that the contributors of open source projects are mostly developers, which was also confirmed by our contact persons for ProM and Camunda. Six users (14%) were regarded as testers, which fits to our results in the user-based approach. One (2%) remaining user was classified as a web developer. As the ProM project has no web component,

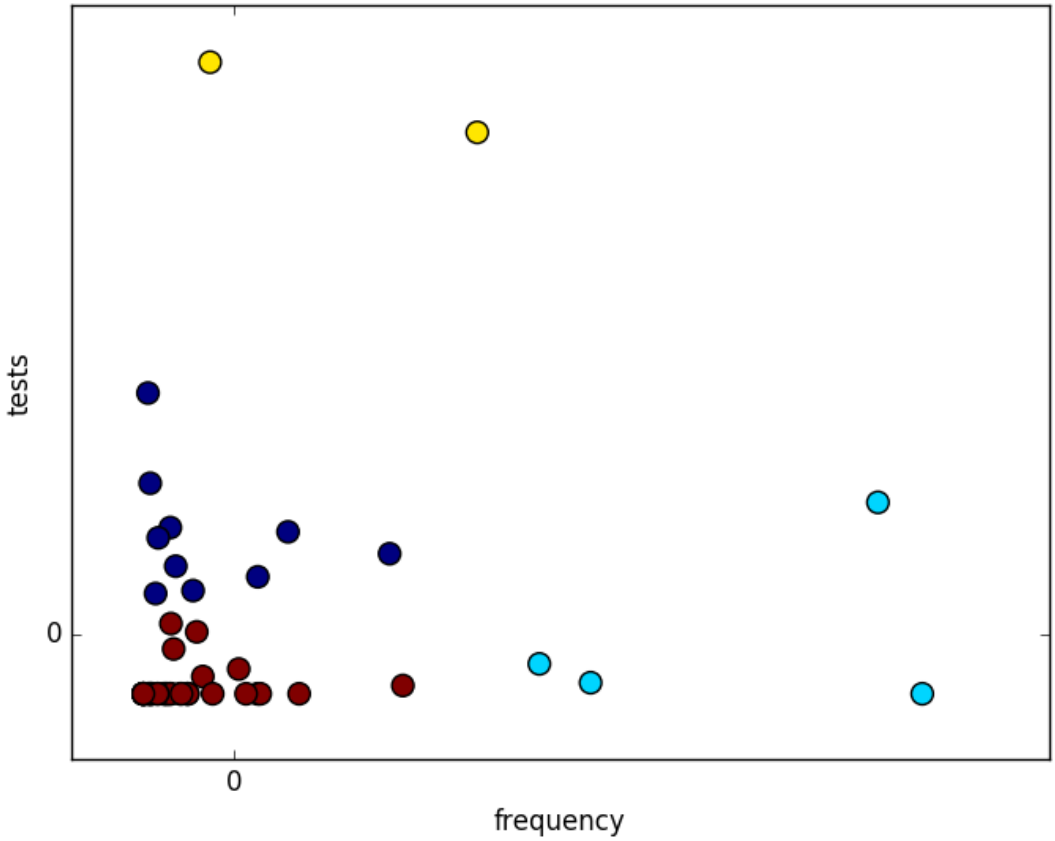


Figure 13 Scatterplot with color-coded clusters, ProM data set

it makes sense that there are no web developers identified. The one we found has only 5 commits, 2 of them web commits so this (potential) misclassification is ignored.

The Camunda data set was analyzed based on the optimal features and classes derived of the Infinica data set. The majority of the users are developers just like in the ProM data set. However, it is an open source project and users can commit anything at any time without restrictions. There is a permanently appointed core team which evaluates, selects and integrates commits for the product. The clustering creates similar classes like in the ProM data set and it can be divided into the following four classes as shown in Figure 14.

- **Core developers:** The red cluster contains the employed users of the Camunda project. Their commit frequency and absolute number commits is far above the others.
- **Engaged developers:** Developers which stay longer with the project are situated in the light blue cluster. They are refining their own committed code or they are extending the product in different areas.

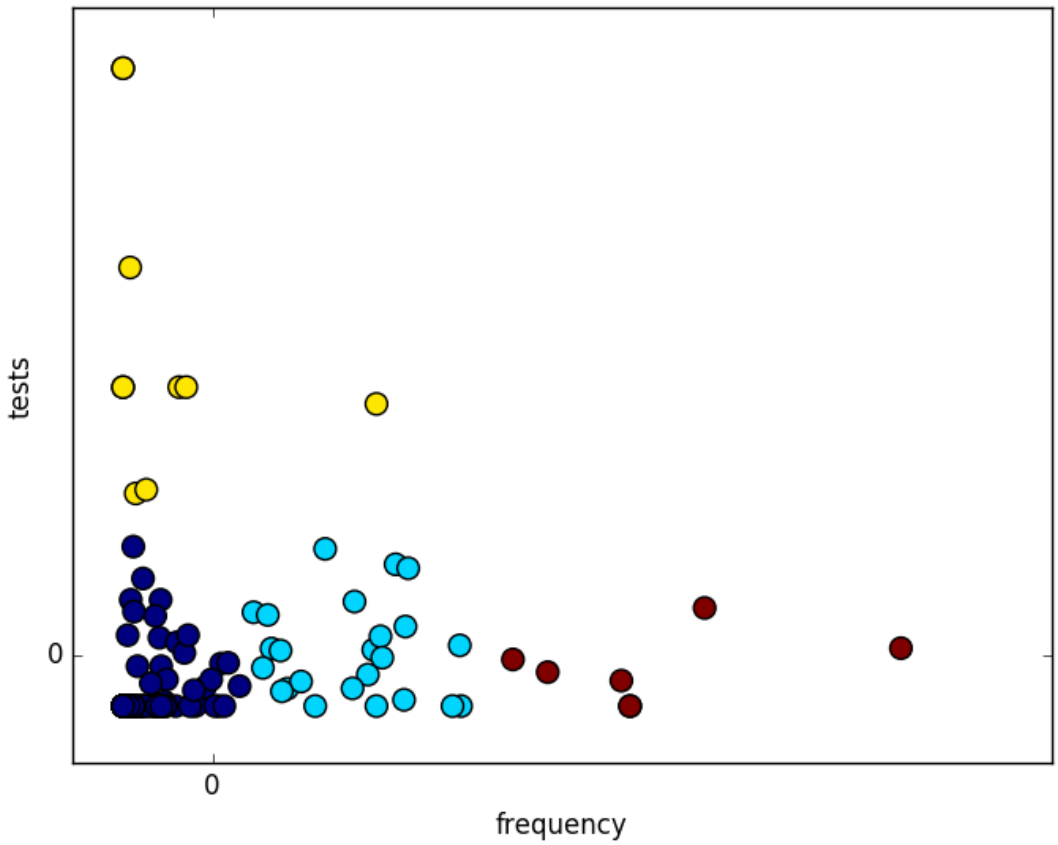


Figure 14 Scatterplot with colorcoded clusters, Camunda data set

- **One-time developers:** Similar to the ProM data set the majority of the users belongs to this group and it is represented by the dark blue cluster. Code usefull or not is committed typically once and there is no lasting commitment.
- **Testers:** Also testers have been identified in the yellow cluster. They already have a lower frequency than developers and therefore it is difficult to make an estimation about their commitment.

For the Camunda data the case is similar. Out of 66 total users, 49 (74%) were regarded as developers, the high number being again explained by the type of project and confirmed by the contact person we spoke to. Ten (15%) testers were found, mostly in line to our knowledge the data. Contrary to ProM, there is a web part in the analyzed Camunda project reflected by the fact that our algorithm found 7 (11%) web developers.

For the user-based apparoach no further distinction can be made based on expertise, time in the company, teamwork, development area, project membership, room allocation, etc. In the case of a development from junior to a senior role in the course of the coverage of the log file the respective persons stay within their

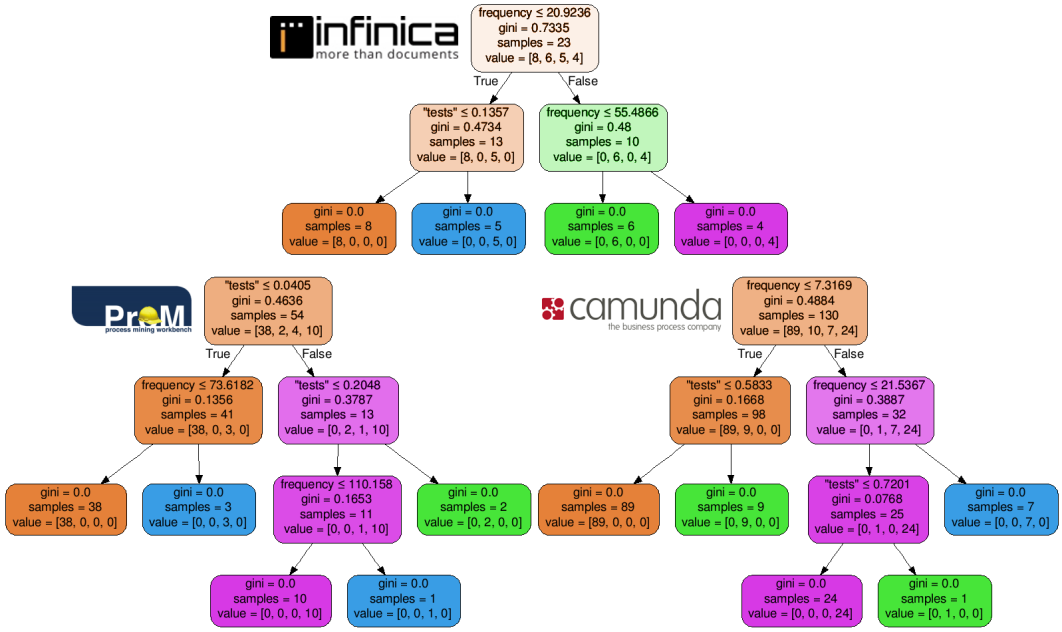


Figure 15 Decision trees, all three data sets

previous clusters. The associated increase or decrease in commit frequency can not be linked to the development.

For the commit-based approach in neither of the two validation data sets a support user was found. This might be due to the differences in the organisational structure between Infinica and the other two or it could stem from the classification rule based on the vcsManagement commi type, for which we already expressed our doubts.

The corresponding decision trees for the previous presented k-means clustering shown in Figure 15 display the boundaries of the clusters. Due to the differences in the structure of the projects, business vs open source, the data sets share only little similarities.

The Infinca decision tree initially splits the developers and the testers apart. In the second step these 2 classes are then divided into subclasses. On the contrary, the other two trees start separating the subsets right away in different orders.

The decision tree model is depicted in Figure 16. When comparing its rules with those of our manual classification there are some clear parallels. In the tree the differentiation between web developers and other developers is done with a border value of 36% for web commits, close to the 40% threshold of the manual table. Testers are identified having more than 17% test commits, similar to the 20% boundary. The decision tree separates the support users from the rest using the vcsManagement type. Looking at the data, this rule is definitely valid for the

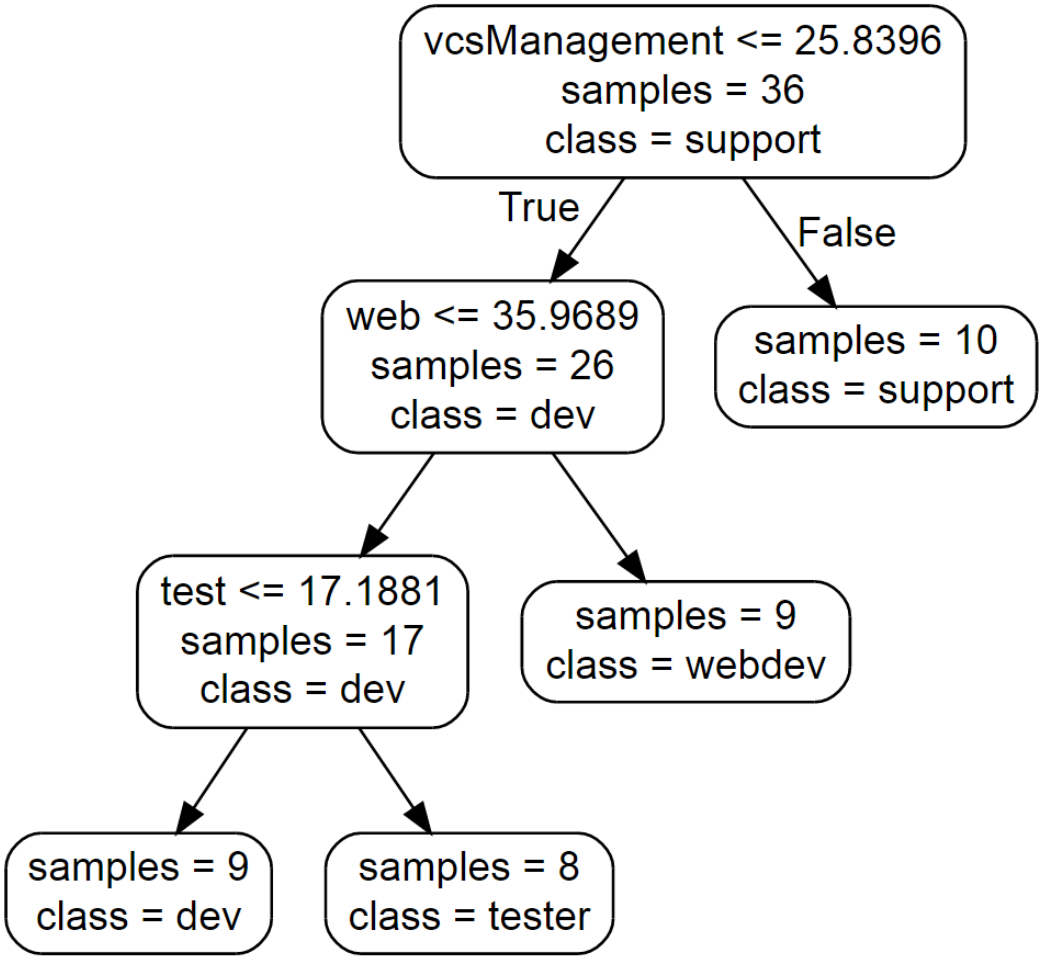


Figure 16 Decision tree, commit based model

Infinica data, however as we were not able to provide a definite explanation for this class having higher percentages for that particular type, we can not say if this rule would also apply to other data sets. One possible explanation is that all users have a similar number of vcsManagement commits within a given timeframe, but because the support role has on average a significantly lower number of commits, they account for a higher percentage in the distribution, however this is just an assumption.

In general the decision tree is more precise, but our manually created rules capture more factors of differentiation and we were able to identify some potential classes which could not be included in the programmatic classification with reasonable effort. A clear advantage of the automated model is of course that it can be applied to other data sets very efficiently. Doing that with the Camunda and ProM data sets provided some verification for our decision tree model. We

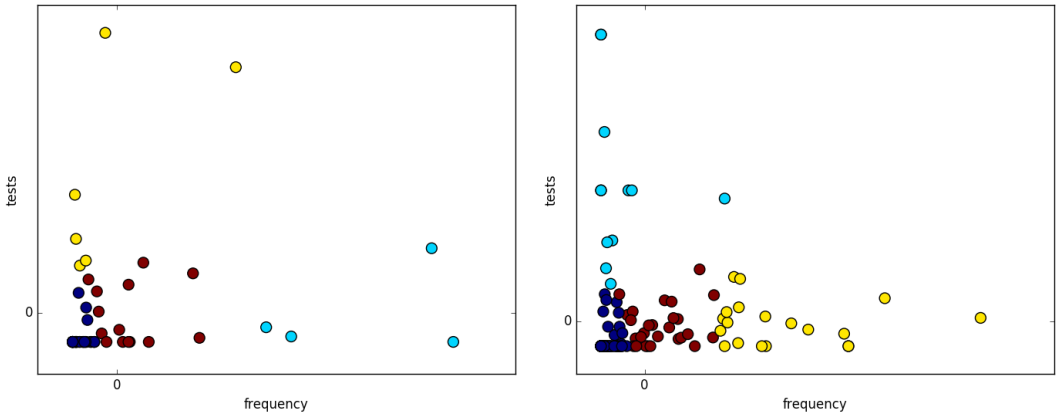


Figure 17 Classication based on Infinica training set: Prom, Camunda

only included users with 5 or more commits into the cross validation.

When validating the model with the Infinica data set and then predicting the ProM and Campunda data set only moderate results can be achieved with the user-based approach as shown in Figure 17. For the Infinica data set meaningful classes can not be conveyed to the other data sets due to the structural differences of the projects.

For ProM (Figure 17 left) the trained classes are less representative than the ones created when clustering on its own, especially since the ProM data set does not include any designated testers. The prediction of Camunda (Figure 17 right) on the other hand performs better. However, the core developer class now also includes very committed contributors.

Because of the missing designated testers in both test data sets, the initial differentiation between technical and non-technical testers is not possible. Due to the moderate results of user based approach the commit based approach was initiated.

6.3.4 Discussion

Throughout this project we discovered a number of research directions to go follow, methods to use and features to analyze, which made our research much more exploratory than originally planned. For our research we picked from a vast selection of potential methods and tools, a small set which seemed promising to us. The user-based approach, which was our initial plan, provided a useful insight into the data but fell short of providing generally applicable results in terms of a classification algorithm. The commit-based approach seems more valuable to us, due to its results and the much larger spectrum of information it delivers for analyzing users. The commit classification method seems quite robust and could

with some extensions and refinements become a useful tool for analyzing arbitrary VCS repositories. The user profiles created based on this classification are definitely an interesting source of information and building a classification model on top of them has proven to be a legitimate approach. Our first research question can be answered with: Yes, it is possible to classify users based on the information usually found in VCS logs. Our results prove that at least for some typical roles in software development there can be enough information in commit messages and file types, to make certain statements about the users who created them. However this is not true for all users, especially because the ways commit messages are used are very different.

The other two questions are more difficult to answer. The classes to be found can differ between repositories. From our experience, developers can be distinguished from other roles quite easily due to high numbers of commits and/or modified files as well as the usage of certain words and phrases. Testers can also be set apart in most cases, mainly through key words. Looking at the file types of added, modified and deleted files also reveals users in the field of web development. Beyond that there are certainly more differences and classes to be found but they are less obvious and require more research.

6.3.5 Limitations

While our algorithms and models work well for describing our three data sets, it still needs to be tested for other repositories, especially from other domains. This would require testing with more data sets, more verification information and more manual analysis. While fetching more VCS repositories and running them through our algorithms could be done in a short amount of time, obtaining the necessary role information is challenging and the manual analysis time consuming.

Acquiring the right data proved to be more challenging than expected. While there is a large selection of open source projects, acquiring useful information for potential classes can be difficult. Furthermore we have to assume that there are significant differences between open source and other repositories, which need to be further investigated for making a clear statement of the general representativeness of our data and our results.

A definite limitation for the whole topic of analyzing VCS logs is the fact that commit messages are used differently between users, repositories and teams. We have to assume that it is practically infeasible to find one classification model, which produces valid results for any VCS repository, because there can be contradictions in what certain statements mean. Even worse, some users even leave the

commit messages empty, which means no classification is possible for this case.

6.4 Future Work

The exposed approach is the preliminary stage on which we plan to build upon a more comprehensive solution of the problem of role discovery from VCS. There are many ways to extend our approaches as well as other potential methodologies for classification. On the technical level, different methods of clustering and classification from the machine learning domain could be used. More conceptual additions would be for example the use of additional features and feature combinations.

There is also much potential for further refining our proposed methods. For instance, we would use more extensively the natural language toolkit for processing the commit messages. So far we only used the more basic functionality of the language processing tools. In this research we focused mainly on the user roles for classification but there might be other classes, for example discriminated by the experience of users. Significant improvements would be possible by analyzing different categories of VCS repositories. Especially log files of large software companies, where many different roles are formally defined and executed would be interesting.

7 Implementation in a BPMS

We have refined the prototype presented in the previous version of this Deliverable by proof-of-concept implementation. Specifically, we have implemented the main components of the framework discussed in Deliverable D2.4 (Havur et al. [2016]). In this prototype, we aim to bring together functionality for reasoning, process mining and document generation. Fig. 18 shows the software architecture that we use. It considers four main components which inter-operate during the execution of a process activity.

Camunda running process. We use the Camunda BPM engine as our Business Process Management System (BPMS). Camunda is an open source platform that allows for defining new components and for interacting with its APIs in a custom way. All the process instances that run into Camunda and their data are stored in log files. Camunda uses two main databases to store its logs: *i)* a database for processes that are currently executing; and *ii)* a database for historical information. These two databases can be queried through provided

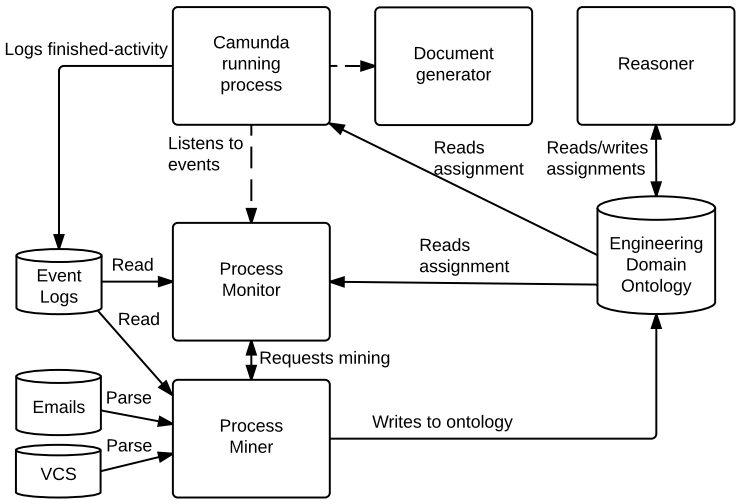


Figure 18 Software architecture of the prototype

Java or REST APIs. The results are returned as either a set of plain old java objects (POJOs) or in the JSON format, respectively.

Before an activity starts to run, it first fetches the ontology which contains the set of assignments of existing resources to activities. Consecutively, a resource is assigned to the activity and thus can appear on their task list. When the resources complete their tasks, an event is triggered. This event is listened by the process miner and the document generator components, who can react accordingly. At the same time, the event is stored into the Camunda database of the running instances. The running processes database and the history database record similarly-structured data. Furthermore, they can be accessed using the same technology, i.e. the Camunda REST APIs. Hence, we abstract both these databases as a single database in Figure 18 and denote it as *Event Logs*.

Reasoner. The reasoner module is implemented as a Java application connected to the Camunda process engine as an asynchronous service. We use Sesame, an open source framework for creating, parsing, storing, inferencing and querying over our ontology data. With respect to the request, the reasoner either performs resource allocation by first translating the RDF data into the ASP language, solving the problem instance using the ASP solver *clasp*, and then writing the allocation results back to the triple store; or it validates all contained SHACL constraints and returns potential violation result back to the process engine.

Process Monitor. This component is in charge of querying the status of the running processes in Camunda. In case a deviation occurs, for example, a process instance cannot be completed within the assigned schedule, the process

monitor must signal out the anomaly. The process adaptation module can use this output to learn the status of the system and subsequently apply an adaptation. This component is implemented as a web client that can read execution logs through the Camunda REST API. Results are returned in the JSON format which are then parsed into POJOs and can be processed by customized monitoring algorithms. In this case the communication happens through periodical queries to the database. An alternative to this is to implement an activity listener that notifies the process monitor whenever a task is completed.

Miner. The miner is in charge of running a number of mining algorithms on the logs from Camunda and from VCSs. Emails and commit messages can also be analyzed by using the approaches discussed in Cabanillas et al. [2016a] and summarized in the document at hand. This component is implemented as a web service, which can be called by the process monitor in order to understand how the activities being monitored have performed in the past. Mining algorithms can give new insights into the processes, like for instance actual execution times and several performance indicators of the process. This can contribute to the domain knowledge. Thus, they are stored again into the ontology as RDF triples.

Document generator. The document generator is in charge of listening to activity submissions and collecting information from them with the final goal of creating textual documents. This component uses customizable event handlers to process changes of process variables and forms compiled by the users. It is implemented in Java and can be imported as a Java library into several other modules that require document generation from events.

The prototype has been published as a demo in the BPM 2016 conference. For more information on the prototype and a video demonstration of its functionalities, the reader is referred to Bala et al. [2016].

7.1 Limitations

The architecture is currently under implementation. The components have been only individually tested. There is the need for a comprehensive software solution that integrates the single software components into one.

SQL console for querying Camunda logs. We are developing a tool for process monitoring. This tool will allow for SQL-like queries on top of Camunda logs. The approach involves mapping Camunda's database schema to RXES **?**. In addition to this we are also developing a tool that can map from RXES

to XES Verbeek et al. [2011] and we plan to use this tool with the approach from Schönig et al. [2015b] in order to make it fully compatible with the RXES standard.

Process adaptation. The process adaptation module that we describe in the framework is yet to be implemented. This module will be developed as an intermediate component between the process monitor and the Camunda engine. It will act as a middle layer that is able to correct slight deviations in the running process, without stopping the workflow. Deviations that are not adjustable may occur. In this case, this component will communicate the need for a schedule to the reasoner.

Connection to ontology. Our ontology is currently under improvement. We are planning to complete it with all the data from the engineering domain ontology. Furthermore, its connections to the various components are yet to be implemented.

User interfaces. We support for mining and monitoring techniques whose results are models that are generated out of data. User interfaces to visualize these data are required in order to easy the understanding of the mining results. Analogously, we plan to provide a fully fledged user interface for the reasoner component.

8 Conclusions

In this deliverable we have studied and implemented several approaches to gather insights from project by combining data from structured and unstructured sources. We showed how we can capture project data through a schema. This approach is flexible: it allows to gather different insights by simply changing the query. We also showed that we can further elaborate the query results and get better understanding of the project. Text mining approaches have also been evaluated. Using semantic models for VCS seems promising. We plan to improve on this by working on a better categorization of the comments. Given the final goal of discovering project phases, we first study the language that is used by people who collaborate together in software projects. This allows us to find out that a few of the main phases of the RUP model are represented in the VCS logs. We then consider Story mining as a preprocessing approach that can extract activities form user stories. The method is applied considering the user comments grouped by user as a single user story. We also deal with structured data such as process logs. Here, we provide a methods that can automatically map XES logs onto RXES. This is particularly useful, as it also allows for applying process mining algorithms on

top on a DMBS. Combined techniques that take into account additional data to the comments have been presented. We used this method to classify users according to roles. The results have been validated with experts that work on the projects under analysis. In order to integrate the approaches from different work packages, we are developing a tool that combines functionalities from Cabanillas et al. [2015a] and Cabanillas et al. [2015b] and use the Camunda BPMS platform. The resulting prototype will be presented in the next SIMPDA 2016 conference. Furthermore, we are developing two approaches that will allow to easily query the Camunda history log for complex insights. These methods are parts of two master theses. In future work, we want to cluster the stories according to their semantic similarity and obtain clusters of similar stories, then compare the to the Gantt chart of the project and see how are stories related to the activities.

References

Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from CVS. *Proc. 2008 Int. Work. Min. Softw. Repos. - MSR '08*, page 125, 2008. ISSN 02705257. 10.1145/1370750.1370780. URL http://portal.acm.org/citation.cfm?doid=1370750.1370780.

Laurence Anthony. AntConc (Version 3.4.3) [Computer Software]. Tokyo, Japan: Waseda University. http://www.laurenceanthony.net/software/antconc/, 2014.

Atlassian. Comparing workflows, 2016. URL https://www.atlassian.com/git/tutorials/comparing-workflows.

Saimir Bala, Cristina Cabanillas, Jan Mendling, Andreas Rogge-Solti, and Axel Polleres. Mining project-oriented business processes. In Hamid Reza Motahari-Nezhad, Jan Recker, and Matthias Weidlich, editors, *Business Process Management*, volume 9253 of *Lecture Notes in Computer Science*, pages 425–440. Springer International Publishing, 2015. ISBN 978-3-319-23062-7. 10.1007/978-3-319-23063-4_28.

Saimir Bala, Giray Havur, Simon Sperl, Simon Steyskal, Jan Mendling, and Axel Polleres. SHAPEworks: A BPMS Extension for Complex Process Management. 2016.

Anne Baumgrass, Sigrid Schefer-Wenzl, and Mark Strembeck. Deriving process-related rbac models from process execution histories. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 421–426. IEEE, 2012.

Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *2010 ACM/IEEE 32nd Int. Conf. Softw. Eng.*, volume 1, pages 125–134, 2010. ISBN 978-1-60558-719-6. 10.1145/1806799.1806821.

Pamela Bhattacharya, Iulian Neamtiu, and Michalis Faloutsos. Determining Developers' Expertise and Role: A Graph Hierarchy-Based Approach. *2014 IEEE Int. Conf. Softw. Maint. Evol.*, pages 11–20, 2014. ISSN 1063-6773. 10.1109/ICSME.2014.23.

David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

Cristina Cabanillas, Saimir Bala, Jan Mendling, and Axel Polleres. Mining processes, resource consumption and witnesses for task completion from logs. Deliverable D3.3 (M3), 2015a.

Cristina Cabanillas, Giray Havur, Jan Mendling, Axel Polleres, Vadim Savenkov,

and Alois Haselboeck. Unified semantic model and reasoning techniques for mining and monitoring process-relevant data. Deliverable D3.3 (M3), 2015b.

Cristina Cabanillas, Jan Mendling, Axel Polleres, and Saimir Bala. Requirements for process, resource and compliance rules extraction from text. Technical Report 1, 2015c.

Cristina Cabanillas, Saimir Bala, Jan Mendling, and Axel Polleres. Combined method for mining and extracting processes, related events and compliance rules from unstructured data. Technical report, WU Vienna, 2016a.

Cristina Cabanillas, Saimir Bala, Jan Mendling, and Axel Polleres. Combined method for mining and extracting processes, related events and compliance rules from unstructured data. Deliverable, WU Vienna, 2016b.

Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976. ISSN 0362-5915. 10.1145/320434.320440. URL http://doi.acm.org/10.1145/320434.320440.

Vincent Driessen. A successful git branching model. *URL http://nvie. com/posts/a-successful-git-branching-model*, 2010.

Ingo Feinerer, Kurt Hornik, and David Meyer. Text Mining Infrastructure in R. *J. Stat. Softw.*, 25(5):1–54, 2008. ISSN 15487660. citeulike-article-id:2842334. URL http://www.jstatsoft.org/v25/i05.

Lukas Fischbach. *Concepts and Methods for Providing a Data Infrastructure for Process Mining on Relational Databases*. Master thesis, WU Vienna, 2016.

Mario Frank, Joachim M Buhman, and David Basin. Role mining with probabilistic models. *ACM Trans. Inf. Syst. Secur.*, 15(4):15, 2013.

Johann Füller, Katja Hutter, Julia Hautz, and Kurt Matzler. User Roles and Contributions in Innovation-Contest Communities. *J. Manag. Inf. Syst.*, 31(1):273–308, 2014. ISSN 07421222. 10.2753/MIS0742-1222310111.

Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. PPDB : The Paraphrase Database. *Proc. NAACL-HLT*, (June):758—-764, 2013. URL http://cs.jhu.edu/{~}ccb/publications/ppdb.pdf.

João Carlos De a. R. Gonçalves, Flávia Maria Santoro, and Fernanda Araujo Baião. Let Me Tell You a Story - On How to Build Process Models. *J. Univers. Comput. Sci.*, 17(2):276—-295, 2011. ISSN 0948695X.

Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1.

Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *Proc. 2008 Int. Work. Conf. Min. Softw. Repos.*, pages 129–132. ACM, 2008.

Giray Havur, Cristina Cabanillas, Saimir Bala, Simon Steyskal, Jan Mendling, and Axel Polleres. Resource and data management service prototype. Technical report, WU Vienna, 2016.

Rashina Hoda, James Noble, and Simon Marshall. Self-organizing roles on agile software development teams. *Softw. Eng. IEEE Trans.*, 39(3):422–444, 2013.

Erik Kouters, Bogdan Vasilescu, Alexander Serebrenik, and Mark G J Van Den Brand. Who's who in Gnome: Using LSA to merge software repository identities. pages 592–595, 2012. 10.1109/ICSM.2012.6405329.

Geoffrey Leech, Paul Rayson, et al. *Word frequencies in written and spoken English: Based on the British National Corpus*. Routledge, 2014.

Henrik Leopold, Sergey Smirnov, and Jan Mendling. On the refactoring of activity labels in business process models. *Information Systems*, 37(5):443–459, 2012.

Sherlock A. Licorish and Stephen G. MacDonell. Understanding the attitudes, knowledge sharing behaviors and task performance of core developers: A longitudinal study. *Inf. Softw. Technol.*, 56(12):1578–1596, 2014. ISSN 09505849. 10.1016/j.infsof.2014.02.004.

Haibing Lu, Yuan Hong, Yanjiang Yang, Lian Duan, and Nazia Badar. Towards user-oriented RBAC model. *J. Comput. Secur.*, 23(1):107–129, 2015. ISSN 0926227X. 10.3233/JCS-140519.

Walid Maalej and Hans-Jörg Happel. Can Development Work Describe Itself? *7th IEEE Work. Conf. Min. Softw. Repos. (MSR 2010)*, pages 191–200, 2010. 10.1109/MSR.2010.5463344.

Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. A Survey of Role Mining. *ACM Comput. Surv.*, 48(4):1–37, 2016. ISSN 03600300. 10.1145/2871148.

Maja Pesic, Helen Schonenberg, and Wil M P Van Der Aalst. DECLARE: Full support for loosely-structured processes. *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pages 287–298, 2007. ISSN 15417719. 10.1109/EDOC.2007.4384001.

W. Poncin, A Serebrenik, and M. van den Brand. Process Mining Software Repositories. *2011 15th Eur. Conf. Softw. Maint. Reengineering*, pages 5–14, 2011. ISSN 1534-5351. 10.1109/CSMR.2011.5.

Vladimir A. Rubin, Christian W. Günther, Wil M. P. Van Der Aalst, Ekkart Kindler, Boudewijn F. Van Dongen, and Wilhelm Schäfer. Process mining framework for software processes. In *Softw. Process Dyn. Agil.*, volume 4470, pages 169–181. Springer, 2007. ISBN 978-3-540-72425-4. 10.1007/978-3-540-72426-1_15.

Stefan Schönig, Cristina Cabanillas, Stefan Jablonski, and Jan Mendling. Mining

the organisational perspective in agile business processes. In *Enterprise, Business-Process and Information Systems Modeling*, pages 37–52. Springer, 2015.

Stefan Schönig, Cristina Cabanillas, Stefan Jablonski, and Jan Mendling. Mining the Organisational Perspective in Agile Business Processes. In *BPMDS*, pages 37–52, 2015a. 10.1007/978-3-319-19237-6_3.

Stefan Schönig, Cristina Cabanillas, Stefan Jablonski, and Jan Mendling. Mining the Organisational Perspective in Agile Business Processes. In *BPMDS*, volume 214 of *LNBIP*, pages 37–52. Springer, 2015b. 10.1007/978-3-319-19237-6_3.

Minseok Song and Wil M P van der Aalst. Towards comprehensive support for organizational mining. *Decis. Support Syst.*, 46(1):300–317, 2008. ISSN 01679236. 10.1016/j.dss.2008.07.002.

Md Arafat Sultan, Steven Bethard, and Tamara Sumner. DLS @ CU : Sentence Similarity from Word Alignment. *SemEval2015*, (SemEval):241–246, 2015.

Elena Tognini-Bonelli. Corpus Linguistics at Work, 2002. ISSN 0891-2017.

B F van Dongen and Sh Shabani. Relational XES : Data Management for Process Mining. *BPM Cent. Rep. BPM-15-02*, 2015. URL BPMcenter.org.

H M W Verbeek, Joos C A M Buijs, Boudewijn F van Dongen, and Wil M P van Der Aalst. Xes, xesame, and prom 6. In *Inf. Syst. Evol.*, pages 60–75. Springer, 2010.

H. M W Verbeek, Joos C A M Buijs, Boudewijn F. Van Dongen, and Wil M P Van Der Aalst. XES, XESame, and ProM 6. In *Lect. Notes Bus. Inf. Process.*, volume 72 LNBIP, pages 60–75. Springer, 2011. ISBN 3642177212. 10.1007/978-3-642-17722-4_5.

Annie T T Ying and Martin P. Robillard. Developer profiles for recommendation systems. *Recomm. Syst. Softw. Eng.*, pages 199–222, 2014. 10.1007/978-3-642-45135-5_8.

Liguo Yu and Srini Ramaswamy. Mining CVS repositories to understand open-source project developer roles. In *Proc. - ICSE 2007 Work. Fourth Int. Work. Min. Softw. Repos. MSR 2007*, pages 7–10, 2007. ISBN 076952950X. 10.1109/MSR.2007.19.

A Copora comparisons

Table 11 Top 100 keywords in GitHub

Rank	Freq	Keyness	Word	Rank	Freq	Keyness	Word
1	1212853	517488.538	i	51	101650	1489.434	only
2	916708	114516.665	it	52	101346	153112.089	file
3	890624	468819.394	this	53	101320	6191.12	your
4	821212	57731.182	is	54	99219	78942.457	change
5	594955	114537.346	you	55	99138	11260.855	see
6	470949	11531.196	be	56	98911	54.706	some
7	419625	172664.97	t	57	98350	187635.858	teamcity
8	393774	41752.597	not	58	97033	25047.237	get
9	327041	51697.196	we	59	96008	24907.336	good
10	311008	83091.706	if	60	95147	133462.402	add
11	264235	409.665	but	61	94837	923.251	could
12	256178	488746.1	http	62	94211	179739.317	artsmia
13	255928	166004.114	should	63	93912	21817.985	make
14	250139	473005.767	com	64	93716	121407.133	x
15	248839	47482.863	can	65	92157	3913.986	me
16	238349	454731.258	https	66	85517	157170.396	id
17	236759	188578.457	d	67	82899	533.86	new
18	235398	449101.228	github	68	82812	8472.168	work
19	222431	289867.254	e	69	80642	65970.871	using
20	202718	229875.524	c	70	78736	5196.082	because
21	195483	17258.064	so	71	78603	148463.986	api
22	190001	48798.324	do	72	74859	63299.856	sure
23	188724	84766.431	just	73	74807	28965.684	does
24	182176	292912.214	build	74	73758	25234.115	ll
25	177067	3217.884	will	75	72940	54782.572	name
26	174618	126391.167	use	76	72738	3348.803	how
27	173664	331026.652	org	77	72427	83848.625	please
28	169245	189941.183	b	78	71648	135996.767	io
29	167700	251285.863	f	79	69874	81256.099	page
30	163970	299089.188	commit	80	69804	14262.959	same
31	157942	109267.606	here	81	69320	105731.118	error
32	154196	5593.506	what	82	69203	25102.281	want
33	151203	30293.068	like	83	68142	5350.697	right
34	149213	84831.75	think	84	68142	1594.704	way
35	148506	892.176	would	85	67958	85395.922	function
36	146943	239408.589	code	86	67343	79257.146	passed
37	145415	238340.12	tests	87	65778	66554.412	instead
38	135298	1679.696	no	88	65778	7187.732	re
39	129005	62613.252	don	89	65549	113139.049	fix
40	127570	45132.065	m	90	62996	120186.157	buildid
41	125407	78562.239	need	91	62975	120146.092	buildtypeid
42	118936	226910.61	html	92	62973	120142.277	viewlog
43	117826	8233.327	my	93	62470	77458.796	version
44	116805	11194.652	now	94	61890	7579.612	used
45	112272	141362.02	test	95	60663	77764.588	check
46	111053	174217.483	thanks	96	60615	28377.29	better
47	110026	8655.042	also	97	60041	1058.466	know
48	108421	72222.388	why	98	59739	15692.208	case
49	106328	100087.887	line	99	59368	57334.546	simple
50	102923	175359.501	ok	100	58950	16644.435	something

Table 12 GitHub versus BNC. Top 100 most frequent words

	GitHub		BNC			GitHub		BNC	
1	1936305	the	5739934	the	51	151203	like	194887	up
2	1285929	to	2913193	of	52	149978	an	190155	what
3	1212853	i	2438445	and	53	149213	think	188294	when
4	1064976	a	2435448	to	54	148506	would	186689	who
5	916708	it	2034829	a	55	147101	there	186398	no
6	890624	this	1840393	in	56	146943	code	180936	out
7	821212	is	981651	that	57	145415	tests	169651	about
8	634698	in	906927	is	58	135298	no	168862	said
9	594955	you	881422	it	59	131610	all	156131	its
10	578323	and	827489	for	60	129005	don	153152	some
11	558000	that	802982	was	61	127570	m	149806	into
12	535551	of	697204	s	62	125407	need	148945	time
13	506379	for	675096	on	63	118936	html	144493	them
14	470949	be	667192	i	64	117965	one	144074	other
15	419625	t	623251	as	65	117826	my	143529	two
16	393774	not	621414	with	66	116805	now	140275	him
17	330042	on	609545	be	67	112593	when	140055	do
18	328147	s	571916	he	68	112272	test	138207	only
19	327041	we	496597	by	69	111053	thanks	137641	than
20	311008	if	495596	you	70	110026	also	132942	could
21	281833	with	486944	at	71	108421	why	130558	then
22	264235	but	425407	are	72	106328	line	130282	my
23	256178	http	416125	this	73	104633	by	125782	over
24	255928	should	405253	from	74	102923	ok	124019	like
25	250139	com	400792	but	75	101650	only	123581	may
26	248839	can	400074	have	76	101346	file	120450	now
27	244599	as	395909	not	77	101320	your	119126	new
28	239346	have	392864	his	78	99219	change	118701	also
29	238349	https	388807	had	79	99138	see	116545	first
30	236759	d	355725	they	80	98911	some	114964	your
31	235398	github	350717	which	81	98350	teamcity	114679	these
32	222431	e	346865	or	82	97033	get	112451	any
33	216583	was	322839	an	83	96693	more	111108	me
34	202718	c	311655	she	84	96008	good	110153	people
35	200536	are	292329	we	85	95147	add	107985	after
36	195483	so	288651	were	86	94837	could	106680	well
37	190001	do	287218	her	87	94211	artsmia	104932	very
38	188724	just	272428	there	88	93912	make	104104	such
39	182176	build	266654	one	89	93716	x	100038	just
40	177067	will	250834	all	90	93210	about	97993	should
41	174618	use	243443	their	91	92157	me	97879	see
42	173664	org	243140	been	92	87673	which	94430	most
43	169245	b	239753	has	93	85517	id	94138	where
44	167947	at	236726	t	94	83758	time	89018	made
45	167700	f	236217	will	95	82899	new	88630	way
46	167136	from	224979	if	96	82812	work	88228	back
47	163970	commit	214187	would	97	80642	using	88046	because
48	161147	or	207983	can	98	79781	then	87472	between
49	157942	here	205541	so	99	78736	because	86617	how
50	154196	what	197587	more	100	78603	api	85375	our

Table 13 GitHub versus RUP frequencies

	GitHub		RUP			GitHub		RUP	
1	1936305	the	8400	the	51	151203	like	326	other
2	1285929	to	3944	of	52	149978	an	326	which
3	1212853	i	3658	a	53	149213	think	316	these
4	1064976	a	3305	to	54	148506	would	303	there
5	916708	it	3049	and	55	147101	there	300	will
6	890624	this	2511	is	56	146943	code	296	they
7	821212	is	2462	in	57	145415	tests	293	when
8	634698	in	1598	that	58	135298	no	284	has
9	594955	you	1267	are	59	131610	all	284	page
10	578323	and	1127	as	60	129005	don	279	implementation
11	558000	that	1094	you	61	127570	m	278	am
12	535551	of	1092	this	62	125407	need	278	some
13	506379	for	1082	be	63	118936	html	274	example
14	470949	be	1027	for	64	117965	one	273	each
15	419625	t	951	an	65	117826	my	269	but
16	393774	not	919	design	66	116805	now	258	up
17	330042	on	894	or	67	112593	when	257	what
18	328147	s	883	it	68	112272	test	256	process
19	327041	we	818	can	69	111053	thanks	256	testing
20	311008	if	772	class	70	110026	also	254	package
21	281833	with	770	system	71	108421	why	252	such
22	264235	but	688	by	72	106328	line	245	also
23	256178	http	662	use	73	104633	by	242	should
24	255928	should	645	object	74	102923	ok	239	its
25	250139	com	640	analysis	75	101650	only	234	s
26	248839	can	636	software	76	101346	file	231	many
27	244599	as	634	requirements	77	101320	your	231	used
28	239346	have	633	classes	78	99219	change	229	into
29	238349	https	615	on	79	99138	see	227	how
30	236759	d	615	with	80	98911	some	222	new
31	235398	github	569	have	81	98350	teamcity	220	two
32	222431	e	568	model	82	97033	get	216	interface
33	216583	was	551	we	83	96693	more	214	then
34	202718	c	541	figure	84	96008	good	210	set
35	200536	are	454	from	85	95147	add	208	development
36	195483	so	449	one	86	94837	could	208	operations
37	190001	do	438	test	87	94211	artsmia	206	must
38	188724	just	437	may	88	93912	make	206	time
39	182176	build	430	chapter	89	93716	x	203	only
40	177067	will	430	objects	90	93210	about	203	very
41	174618	use	368	state	91	92157	me	201	name
42	173664	org	363	case	92	87673	which	200	diagrams
43	169245	b	354	not	93	85517	id	197	patterns
44	167947	at	353	more	94	83758	time	195	uml
45	167700	f	350	if	95	82899	new	191	any
46	167136	from	345	between	96	82812	work	189	about
47	163970	commit	338	all	97	80642	using	189	product
48	161147	or	329	at	98	79781	then	189	workflow
49	157942	here	327	code	99	78736	because	187	using
50	154196	what	326	business	100	78603	api	184	user

Table 14 RUP keywords using BNC as reference corpus

Rank	Freq	Keyness	Word	Rank	Freq	Keyness	Word
1	919	5265.796	design	51	184	715.492	user
2	633	4169.816	classes	52	59	711.807	ejb
3	634	4142.086	requirements	53	67	710.798	stakeholders
4	645	4041.867	object	54	117	707.347	programming
5	636	3549.507	software	55	175	693.351	elements
6	772	3364.497	class	56	102	679.194	dependency
7	640	3213.087	analysis	57	72	678.871	iteration
8	568	2713.496	model	58	326	676.73	business
9	430	2687.702	objects	59	161	675.931	tests
10	195	2497.087	uml	60	51	661.214	bankaccount
11	770	2402.769	system	61	363	658.22	case
12	541	2323.235	figure	62	818	641.107	can
13	189	2156.263	workflow	63	144	628.838	program
14	163	2031.289	modeling	64	87	619.956	deployment
15	438	1849.759	test	65	278	612.291	am
16	327	1771.47	code	66	109	608.344	instances
17	279	1762.52	implementation	67	256	606.542	process
18	200	1759.637	diagrams	68	177	592.331	pattern
19	430	1755.852	chapter	69	189	586.653	product
20	662	1545.132	use	70	115	578.506	transition
21	256	1498.752	testing	71	82	577.303	realization
22	216	1411.267	interface	72	95	564.386	inheritance
23	118	1268.46	behavior	73	59	552.771	nested
24	254	1226.465	package	74	106	551.383	associations
25	167	1206.329	diagram	75	41	531.564	registrationmanager
26	145	1166.036	interfaces	76	144	531.231	relationships
27	162	1136.316	attributes	77	1267	516.984	are
28	284	1133.367	page	78	64	510.285	functionality
29	105	1067.774	subsystems	79	173	488.737	specific
30	104	1024.926	oo	80	87	481.325	collaboration
31	110	1007.531	java	81	41	477.935	classifier
32	99	1003.896	artifacts	82	274	472.698	example
33	165	948.81	define	83	103	472.398	component
34	96	943.606	subsystem	84	55	470.234	refine
35	163	942.057	jump	85	89	466.135	domain
36	125	924.251	attribute	86	108	462.596	languages
37	208	917.704	operations	87	69	457.78	syntax
38	91	910.671	isbn	88	44	455.817	subclasses
39	136	876.196	specification	89	68	455.156	composite
40	148	865.285	functional	90	35	453.774	addcourse
41	197	858.293	patterns	91	50	446.897	addison
42	2511	852.612	is	92	34	440.809	statechart
43	96	840.015	aggregation	93	60	440.13	coupling
44	131	802.212	packages	94	77	431.168	documentation
45	99	797.031	semantics	95	33	427.844	multiobject
46	151	781.856	architecture	96	33	427.844	submachines
47	71	744.776	dependencies	97	141	419.112	instance
48	368	731.656	state	98	165	413.13	operation
49	147	725.787	components	99	89	409.959	interaction
50	75	721.874	int	100	33	398.267	modeled

Table 15 GitHub keywords using BNC as reference corpus

Rank	Freq	Keyness	Word	Rank	Freq	Keyness	Word
1	1212853	593890.895	i	51	58315	80529.844	comment
2	256178	472946.795	http	52	125407	80041.322	need
3	890624	467442.461	this	53	69874	80025.254	page
4	250139	458380.288	com	54	55384	78242.885	remove
5	238349	440031.524	https	55	54186	77467.306	images
6	235398	434583.492	github	56	327041	77163.451	we
7	173664	320407.237	org	57	99219	76729.639	change
8	163970	289588.017	commit	58	67343	76251.15	passed
9	222431	284645.346	e	59	60663	76212.421	check
10	182176	284071.219	build	60	48550	76210.672	collections
11	167700	246838.28	f	61	62470	74500.682	version
12	146943	231185.757	code	62	42703	73795.266	builds
13	145415	230291.597	tests	63	108421	73385.886	why
14	202718	228654.809	c	64	129005	73021.45	don
15	118936	219575.452	html	65	39120	71580.935	oss
16	419625	199857.038	t	66	37776	69740.72	href
17	236759	195078.741	d	67	47314	69442.823	summary
18	169245	187476.415	b	68	39932	65353.748	default
19	98350	181570.304	teamcity	69	74859	64534.593	sure
20	94211	173929.028	artsmia	70	65778	64215.998	instead
21	111053	171255.971	thanks	71	39752	64169.967	update
22	102923	169396.327	ok	72	34715	64089.61	ubuntu
23	594955	167120.848	you	73	80642	63788.679	using
24	255928	162316.584	should	74	35103	63150.499	git
25	85517	152529.732	id	75	34104	62902.609	php
26	101346	148781.95	file	76	56991	62749.071	looks
27	78603	143620.453	api	77	34042	62395.444	js
28	112272	136482.109	test	78	51148	61465.131	fixed
29	916708	136367.554	it	79	36599	60878.691	pr
30	71648	131579.753	io	80	190001	59989.718	do
31	95147	131047.368	add	81	44131	59985.004	outcome
32	174618	123262.624	use	82	54558	59433.832	method
33	93716	119434.804	x	83	32930	59226.34	python
34	62996	116300.995	buildid	84	51071	59138.549	maybe
35	62975	116262.226	buildtypeid	85	49604	58850.629	master
36	62973	116258.533	viewlog	86	56751	58367.789	doesn
37	157942	113344.541	here	87	43513	57559.51	pull
38	65549	109663.405	fix	88	821212	57555.778	is
39	149213	107136.557	think	89	40319	56790.59	files
40	69320	102113.252	error	90	40221	56524.006	string
41	52666	97168.512	jpg	91	48662	56105.925	user
42	106328	97006.802	line	92	38395	55799.561	guest
43	188724	95385.642	just	93	59368	55386.022	simple
44	49464	91318.694	linux	94	29851	55109.864	B
45	311008	91045.423	if	95	58666	55005.096	success
46	72427	83981.958	please	96	29574	54598.477	nrel
47	47493	83920.967	rust	97	72940	53648.842	name
48	45529	83584.9	notifications	98	248839	52098.218	can
49	67958	83052.943	function	99	28117	51908.614	he
50	46846	82081.643	auto	100	31801	51342.29	builders

Table 16 GitHub keywords using RUP as reference corpus

Rank	Freq	Keyness	Word	Rank	Freq	Keyness	Word
1	1212853	4745.013	i	51	52666	244.573	jpg
2	238349	1106.858	https	52	190001	244.153	do
3	235398	1093.154	github	53	65549	241.286	fix
4	256178	1064.277	http	54	74859	239.335	sure
5	419625	1022.381	t	55	169245	238.536	b
6	916708	938.423	it	56	311008	237.697	if
7	250139	889.463	com	57	51071	237.166	maybe
8	236759	865.046	d	58	67343	232.36	passed
9	173664	761.574	org	59	49604	230.354	master
10	163970	747.571	commit	60	56991	228.675	looks
11	167700	708.529	f	61	47493	220.551	rust
12	216583	622.345	was	62	45529	211.43	notifications
13	890624	558.524	this	63	55448	208.713	yes
14	182176	515.802	build	64	97033	206.255	get
15	111053	515.714	thanks	65	43513	202.068	pull
16	117826	505.373	my	66	65778	201.302	re
17	328147	503.752	s	67	53866	195.799	did
18	118936	479.675	html	68	58666	190.157	success
19	157942	458.951	here	69	55384	186.135	remove
20	98350	456.723	teamcity	70	49464	182.294	linux
21	149213	453.473	think	71	39120	181.668	oss
22	393774	449.327	not	72	101320	181.262	your
23	94211	437.502	artsmia	73	53593	178.475	issue
24	127570	423.843	m	74	38395	178.301	guest
25	222431	390.287	e	75	94837	177.91	could
26	102923	382.191	ok	76	60663	175.851	check
27	92157	380.411	me	77	37776	175.426	href
28	108421	372.56	why	78	37692	175.036	didn
29	93716	341.684	x	79	47654	174.336	ve
30	71648	332.723	io	80	36599	169.96	pr
31	78603	319.059	api	81	38559	168.076	nice
32	85517	317.38	id	82	55535	167.756	since
33	202718	310.157	c	83	47176	166.588	probably
34	72427	306.189	please	84	46846	165.154	auto
35	188724	305.57	just	85	42737	164.784	wrote
36	62996	292.544	buildid	86	51148	163.204	fixed
37	62975	292.447	buildtypeid	87	35103	163.013	git
38	62973	292.437	viewlog	88	73758	161.437	ll
39	116805	290.851	now	89	34715	161.211	ubuntu
40	106328	283.214	line	90	34571	160.543	sorry
41	101346	283.028	file	91	34042	158.086	js
42	151203	280.035	like	92	95147	155.37	add
43	129005	273.647	don	93	44131	153.382	outcome
44	255928	270.021	should	94	65778	151.158	instead
45	58315	258.992	comment	95	48213	150.874	seems
46	195483	252.294	so	96	31801	147.679	builders
47	54186	251.632	images	97	60615	145.132	better
48	148506	249.75	would	98	30705	142.59	yeah
49	53097	246.575	de	99	37424	141.173	failed
50	264235	245.825	but	100	56751	140.825	doesn

B Concordances

Hit	KWIC		File
1	not if (res insert-pan-fader-xmit ![new	design why does datasource require a name? Thx for	ccfi.txt
2	ecture14.pdf "don't anticipate the future" "[use	design that] contains the fewest possible classes	ccbx.txt
3	:+1: -1 Think about this sentence: "API	design for mathematics". Doesn't it sound strange	ccbz.txt
4	the design. do you think using a '"list":[]'	design helps clear up the design for JSON-ites [g]	ccbd.txt
5	om sample gcenv.h? Yep, file this one under "Poor	Design That's A Minor Annoyance". This function wa	ccfg.txt
6	"game perspective" is gaining over the "software	design perspective", which is not something that I	ccce.txt
7	ignored: 250 Buil We cannot force the "tdp"	design anytime. By the way there can be other	ccge.txt
8	wrong with the text before? Something like: "The	design incorporates a monitoring port per pipeline	ccas.txt
9	workaround is to load I'd reword to "the	design of the session specification requires..." (	ccfs.txt
10	all and process thi AAAAAAAAAAAAAAAAAA :100: ##	Design - All receivers share a symmetric 32-b	ccfm.txt
11	smooth as it should at this point. ###	Design and Code Changes * Consistent use of impr	ccac.txt
12	We read a text file. What for "rb"? ####	Design reference source A then we write a text	ccfe.txt
13	Read the comments bro! xP Code Smells &	Design Pattern Violations: This code was not DRY.	ccbg.txt
14	was already up to date. Code Smells &	Design Pattern Violations: This is another example	ccbg.txt
15	the repo, it comes "ready to go" (a	design principle for the whole Vagrant box experie	ccfq.txt
16	itance and make events handle themselves (command	design pattern). Skulle det ha vært ::Trust::Inher	ccac.txt
17	in spring), with unknown added value IMO (don't	design something when we have no use case, Can	ccae.txt
18	-level requirements for the data itself? (Formal	design and weird edge cases can come later) Awesom	ccgc.txt
19	2) It is that way by design (my	design :)). I'm thinking you should have the same	ccbx.txt
20	espace ;-) Note that because of the (perhaps-bad)	design of non-linear matchine, equality involves a	ccbu.txt
21	test and by using integers as keys (solid	design choice) -Iterating through parsed data -Dyn	ccaw.txt
22	) I'm looking forward to it :) (The	design includes at least one house boat, right?) H	ccbx.txt
23	condition of a while loop like this (this	design choice was made because too many people mix	ccbl.txt
24	to be a remnant from the original (v1)	design of Boost.Filesystem. In v2 that method was	ccar.txt
25	hould name it "config". Layout "strategy" (yeah..	design pattern :)) would position all "widgets".	ccfc.txt
26	lear in the most recent commit. :heart: **LOGO:**	Design Justification - C and S are stacked on	ccae.txt
27	file. @IanMayo your changes makes perfect sense -	design is cleaner with the canHandle responsibilit	ccfo.txt
28	[x"$install = xyes]; then "' 1.2.2 -Neues	Design -Karte von Orten -Eigene Armbänder und Bild	cccb.txt
29	wayah e jatah ku ki sing upload >.<	Design pattern for Cost is not really a design	ccae.txt
30	@aron should talk about this and plan /	design / spec this better. I'm reading this and	cccj.txt

Table 17 Concordance list of the word *design*

Hit	KWIC		File
1	view.chromium.org/317783008/ 2. First line "[IAP]	Implement query and buy functionalities." r+ no ,	ccbs.txt
2	In this case, I think "refactor" means "actually	implement the functionality." Feel free to submit	ccau.txt
3	e it. @NotNull annotation is forgotten. >"Always	implement things when you actually need them, neve	ccba.txt
4	in "foregone" which is the keyword for "don't	implement this". >(PECL pecl >= 0.10.0) >http —	cccc.txt
5	etc) I can also imagine queries like "extensions:	implement the thing" whic Add a rainvow color func	cccy.txt
6	now. Can you make a note like "May	implement in the future" inside of the code? we	cccr.txt
7	introduce a stub-function there, that says "TODO	implement me" and has documentation saying what it	ccbf.txt
8	fix for request "Get Build by Id #61"	implement DownloadTestsCsv #386 hp.target changed	ccbk.txt
9	to get it to work in MPAS. #ifdef	implement our own 'strcasestr()'. #endif This is r	cccj.txt
10	way. They come in pairs: /Int and %Int	implement t-division (which I think is the most co	ccbd.txt
11	other group need to think about that &	implement I think I'll move the script over to	ccfz.txt
12	Slds. El abr. 2, 2016 10:50, "diegorc22" (8)	Implement a SASS-JSON translator. [Did you find a	ccgj.txt
13	always present, but it's the most reliable (could	implement failover chaining). Oh and when you cli	ccga.txt
14	ist.github.com/zeffii/53e618454372798c171c (could	implement special unpacking for gist compressions)	ccck.txt
15	no cache, you should store it explicitly. (Or	implement a cache. :) Any other 'method' should ra	ccaf.txt
16	- so, please, upgrade to Rails 4 (or	implement Rails 3 support in a fork until you	ccas.txt
17	# Server commands * Implement "uptime" command *	Implement "add-superuser"/"re header s/constantly/	ccga.txt
18	## Server commands *	Implement "uptime" command * Implement "add-superu	ccga.txt
19	will work on this next. I will *	Implement an abort mechanism. Perhaps an animated	ccao.txt
20	will work on this next. I will *	Implement an abort mechanism. Perhaps an animated	ccao.txt
21	g items) # ololord.js v1.0.0-beta ## General *	Implement different access levels for different bo	ccga.txt
22	reason doesn't work, there are two options: *	Implement fallbacks (so for example the localisati	ccas.txt
23	the Git Bash console. 2 options here: *	implement 'Iterator', the downside is that we can'	ccfp.txt
24	s problem and implement relative gui interface *	Implement Meteo settings gui interface it also nee	ccar.txt
25	notify can wait a while.. ;) TODO: *	Implement save and restore activated plugins. * Co	ccat.txt
26	figuration support to ServiceContractGenerator. *	Implement Se Moet ik dan nog iets anders doen	ccae.txt
27	error now when I try to insta *	implement semaphore instead of a mutex to reduce o	ccam.txt
28	nt different access levels for different boards *	Implement superuser ## Server commands	ccga.txt
29	table.Map' with 'java.util.Map' and call RxJava *	Implement 'toMap' and 'toMultimap' directly in RxS	ccbs.txt
30	steps 3 through 6 into something like *	Implement your feature and make sure it is covered	

Table 18 Concordance list of the word *implement*

Hit	KWIC		File
6	ehind? "keyboard cat!" Again no ERROR check(s) !	test ACTUALSIZE | | ACTUALSIZE=1 something like t	ccca.txt
7	Bytecode‘ allow portable location. try this one !	test another Looks good, thanks a lot broski! temp	ccca.txt
8	:+1: , if zou want :shipit: !	test comment The reason I’m not handling loginComp	cccy.txt
9	It’s my fault and I AM SORRY !	test comment Aber wieso steht da nothing to do	ccgf.txt
10	allow spaces in it... a size()-misusage bug !	Test documented with PR https://github.com/wso2-de	cccr.txt
11	was a typo here :(self.opts.sstp = 1 !	test This commit still contains multiple variables	cccu.txt
12	but indeed you were quicker than me !	Test your stuff on VAAPI. Works fine as it	ccfo.txt
13	the docsite one Don’t do this please !!	test 8 It’d better follow the import convention us	cccs.txt
14	nie ! nie podoba mi się to !!	test @theophani fair suggestion! Is it ok to do	ccai.txt
15	x86 OK (329 of 329 tests passed) ![	test page up Deze hoort in ‘Domain‘, niet	ccco.txt
16	9/6b49f612-cfa2-11e2-8009-1f237179c4b1.png) ![201	Test Result: 1. This is palette addition result o	ccbk.txt
17	level description of what’s going on. ![20140503	test comment This variable is not well named. It	ccbu.txt
18	9.. no fear next time. asesome ???? ![20160402	test https://docs.oracle.com/javase/8/docs/api/jav	ccgk.txt
19	84 80334374 (evanweaver) - x86 Build Failed ![Bu	Test comment. ‘Code’ First, a line is repeated	cccm.txt
20	b7b-11e4-82ea-9eba8e887209.PNG) Test Run ![final	test run](https://cloud.githubusercontent.com/asse	ccck.txt
21	/70e96f38-ab1e-11e2-92eb-21bec957bb8a.png) ![kpca	test output](https://f.cloud.github.com/assets/361	ccah.txt
22	Test Test Looks good. Test Test Test ![tumblr	Test Test Test Test Test Test Test Test Test	ccbz.txt
23	7293’ Check the screenshot for reference. ![wrong	test case codeigniter date helper gmt to local](ht	ccau.txt
24	one half of the res == -1 && !feof(pfp)	test is redundant. Yes we have been hunting down	ccaw.txt
25	the door. GET//schema.org/:pwz3n0/:4d1b6b6 !join	test class desperado amigo No guys, just me. Proba	ccfb.txt
26	es #1 ">alert("XSS") ">alert(1) alert(1) "	TesT ">">">0); Cute, but this will	ccaw.txt
27	) ">alert(1) ’">alert(/pwned/) alert(1) "	TesT ">">1 rather than n in the	ccbd.txt
28	a ‘return‘. In this test case "	test " no anything for testing then what is	ccex.txt
29	and see how they fare. if ["$	TEST = "phpunit-coverage.xml"] Same here My bad.	ccam.txt
30	app - application output \n" +	test - test output \n debug - tracebacks and othe	ccga.txt
31	a I would replace this sentence with "	test fails if no video is visible or if	ccav.txt
32	ated. http://web.mit.edu/gnu/doc/html/autoconf "	Test programs should exit, not return, from main,	ccfm.txt
33	»http://web.mit.edu/gnu/doc/html/autoconf >"	Test programs should exit, not return, from main,	ccfm.txt
34	=>/+""+virt+test virt/ =>/+virt+""+	test /virt/ =>/+""+virt+""+test “‘ That’s why	

Table 19 Concordance list of the word *test*