Master Thesis Project

Multi-version software quality analysis through mining software repositories

Author: Illia Kriukov
Supervisor: Welf Löwe
Examiner: Shiyan Hu
Reader: Shiyan Hu
Semester: HT 2017
Course Code: 4DV50E
Subject: Computer Science
Abstract

The main objective of this thesis is to identify how the software repository features influence software quality during software evolution. To do that the mining software repository area was used. This field analyzes the rich data from software repositories to extract interesting and actionable information about software systems, projects and software engineering. The ability to measure code quality and analyze the impact of software repository features on software quality allows us to better understand project history, project quality state, development processes and conduct future project analysis. Existing work in the area of software quality describes software quality analysis without a connection to the software repository features. Thus they lose important information that can be used for preventing bugs, decision-making and optimizing development processes. To conduct the analysis specific tool was developed, which cover quality measurement and repository features extraction. During the research general procedure of the software quality analysis was defined, described and applied in practice. It was found that there is no most influential repository feature and the correlation between software quality and software repository features exist, but it is too small to make a real influence.

**Keywords:** software quality analysis, mining software repositories, software repository features, quality measurement, quality metrics.
Abbreviations

MSR – Mining software repositories.
KPI – Key performance indicator.
VCS – Version control system.
SCM – Source code management.
CVS – Concurrent Versions System.
SVN – Apache Subversion.
## Contents

1. Introduction ................................................................................................................. 6  
   1.1 Background and Motivation .................................................................................. 6  
      1.1.1 Background .................................................................................................... 6  
      1.1.2 Motivation ...................................................................................................... 9  
   1.2 Problem Statement ................................................................................................. 9  
      1.2.1 Research Questions ..................................................................................... 9  
   1.3 Contributions ......................................................................................................... 10  
   1.4 Report Structure .................................................................................................... 11  
2. Background .................................................................................................................. 12  
   2.1 Software repositories ............................................................................................. 12  
      2.1.1 Functional and non-functional requirements ............................................... 30  
      2.1.2 Use cases ....................................................................................................... 29  
   2.2 Version Control Systems (VCS) ............................................................................ 29  
      2.2.1 Types of VCS systems .................................................................................. 13  
      2.2.2 Version Control Software tools ..................................................................... 16  
   2.3 Mining software repositories (MSR) ..................................................................... 18  
   2.4 Software quality metrics and KPIs ....................................................................... 19  
   2.5 Quality analysis tools ......................................................................................... 22  
      2.5.1 VizzAnalyzer™ .............................................................................................. 22  
   2.6 Existing research ................................................................................................... 23  
3. Method ......................................................................................................................... 25  
   3.1 Scientific Approach ............................................................................................... 25  
   3.2 Method Description ............................................................................................... 26  
   3.3 Ethical Considerations ........................................................................................... 28  
4. Implementation ............................................................................................................ 29  
   4.1 General overview and requirements .................................................................... 29  
      4.1.1 Use cases ....................................................................................................... 29  
      4.1.2 Functional and non-functional requirements ............................................... 30  
   4.2 Software Architecture .......................................................................................... 31  
      4.2.1 Components ................................................................................................. 31  
   4.3 Programming Language and Tools ....................................................................... 34  
      4.3.1 Java ............................................................................................................... 34  
      4.3.2 Apache Maven ............................................................................................... 35  
      4.3.3 Git .................................................................................................................. 35  
   4.4 Data analysis and visualization ............................................................................. 35  
   4.5 User guide for usage and future development .................................................... 36  
5. Evaluation .................................................................................................................... 38  
   5.1 Experiments .......................................................................................................... 38  
      5.1.1 Measurement approach .................................................................................. 38  
      5.1.2 Source projects for measurements .................................................................. 39  
   5.2 Data analysis and visualization .......................................................................... 39  
   5.3 Findings and discussion ...................................................................................... 44  
6. Conclusions and Future Work .................................................................................... 46
1 Introduction
This chapter gives a general overview of the topic and research, which was conducted. It contains motivation description, problem statement with research questions and contributions to the quality analysis domain.

1.1 Background and Motivation
The aim of this research project is a *multi-version software quality analysis through mining software repositories*. The two corresponding domains are mining software repositories (MSR) and multi-version software quality analysis. The main objective of this thesis is to identify how software repository features influence software quality during software evolution.

1.1.1 Background
The MSR field analyzes the rich data from software repositories to extract interesting and actionable information about software systems, projects and software engineering [1]. MSR is a broad class of investigations into the examination of software repositories [2].

Software repositories refer to artifacts that are produced and modified during the evolution of software. Such repositories include a variety of sources, namely the data and metadata stored in a source code repositories, information from requirements/bug tracking systems, communication archives, etc. [2].

MSR is about evolving software versions, changes, data (source code, bug reports, etc.), meta data (who, when, where, why). Mining software repositories includes data analysis, data measurement, data mining, data recovery, static source analysis, statistical analysis (correlation) [2]. In this thesis MSR will be used for collecting data about development processes during software evolution.

![Software repositories diagram](image)
There are various types of the software repositories (see figure 1.1). The most popular ones are listed below [3]:

- Source code repositories managed by version control systems (VCS) like Git, Subversion, Mercurial.
- Issue tracking systems such as JIRA by Atlassian, Team Foundation Server, Redmine, YouTrack.
- Bug tracking systems such as Bugzilla and FogBugz.
- Message and communication archives.

Due to the large number of repository types, this thesis focuses only on source code repositories. The figure below [3] illustrates the mining process for such type of repositories.

![Figure 1.2: MSR process](image)

Each VCS has features. Features constitute the interesting information that can be extracted and analyzed for the different purposes. For example below are the some possible features of existing source code repositories. During the experiments several features from this list will be considered:

- Software versions (minor and major),
- Code changes (added, edited, removed lines),
- Count of changed files,
- Commit messages,
- Meta data:
  - Frequency of commits,
  - Date of commit,
  - Author of commit.

To achieve the objective of the research it is necessary to also assess quality of software captured in the repositories. It will be measured for each selected feature. For example, in case of using software versions, software quality should be measured for each version of the project. In this thesis for the software quality measurement 8 key performance indicators (KPIs) with 21 quality metrics will be used. They are presented on the figure below:
Measurements will be conducted for Java projects. Below is the description of the quality metrics. All of this KPIs and metrics will be measured with VizzAnalyzer™ (see section 2.5.1).

1. Cloning:
   - Cloning - Cloned fraction of class/interface,
   - Similarity - Similarity to other classes/interfaces,
   - Similar Files - Number of similar classes/interfaces.

2. Cohesion:
   - LCOM - Lack of Cohesion in Methods,
   - ILCOM - Improvement of Lack of Cohesion in Methods,
   - LD - Locality of Data,
   - TCC - Tight Class Cohesion.

3. Complexity:
   - CC - Mc Cabe Cyclomatic Complexity,
   - WMC - Weighted Method Count.

4. Coupling:
   - CBO - Coupling Between Objects,
   - DAC - Data Abstraction Coupling,
   - MPC - Message Passing Coupling,
   - RFC - Response For a Class.

5. Inheritance hierarchy:
   - DIT - Depth Of Inheritance Tree,
   - NOC - Number Of Children.

6. Readability:
   - LEN - LEngth of Names,
   - LOD Classes - Lack Of Documentation in Classes.

7. Size:
   - LOC - Linces Of Code,
   - NAM - Number of Attributes and Methods,
NOM - Number Of local Methods.

8. Suspicious code patterns:
   – CYC Classes - Cyclic Dependencies Between Classes.

1.1.2 Motivation

There are several reasons to work on this topic. First, we assume that software repository features (e.g. software versions, source code changes, frequency of commits) influence code quality and can thus affect the number of bugs in software products. This will lead to system failures and costs for fixing them.

Existing work in this area does not correspond software quality analysis with software repository features. As a result, valuable information that could be used to prevent bugs, make decisions, optimize development processes, is missed. See section 2.6 for more details.

The second reason is that the ability to measure code quality and analyze the impact of software repository features on software quality allows us to better understand project history, project quality state, development processes, and to conduct future project analysis.

1.2 Problem Statement

The main objective of the presented research is to identify how software repository features influence software quality during software evolution. To achieve this objective, it is necessary to conduct software quality measurement. Quality will be measured for each selected feature. For example, software quality could be measured for each version of the project, in case of using the software versions feature. For code changes and changed files, it could be measured for each commit in the VCS.

When all KPIs have been obtained they should be merged into one single quality value [4] for each project version (in case of using project versions repository feature). After all measurements have been finished, correlation between repository features and value of the software quality can be calculated. To obtain more reliable results it is necessary to make software quality measurements and correlation calculation for multiple projects. Analysis will be conducted for Java projects.

As the result of our research, we strive to automate quality analysis process, find the relationship between software repository features and value of the software quality. After that, we aim at identifying the most influential repository features.

1.2.1 Research Questions

Considering all of the above there are several research questions in this research project (table 1.1).
Table 1.1 – Research questions

<table>
<thead>
<tr>
<th>RQ1.</th>
<th>How do software repository features influence software quality during software evolution? In other words, is there any correlation between software repository features and software quality?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ2.</td>
<td>Which repository features have the greatest impact on the software quality?</td>
</tr>
<tr>
<td>RQ3.</td>
<td>Is it possible to automate process of the multi-version software quality analysis through MSR, which contains of the quality KPIs and metrics measurement and extraction of the software code repository features?</td>
</tr>
</tbody>
</table>

1.3 Contributions

The main contribution of the presented research is expected to be a procedure of software quality analysis through MSR and a judgment about influence of the software repository features on the software quality. Existing works in this area, which will be discussed in Chapter 2, do not allow us to do such analysis and make such a judgment.

Also, one of the main values of this work will be the developed tool, which can automate whole procedure of software quality analysis through MSR and generate user-friendly XLSX reports. This tool will provide ability for another future researchers to conduct their own more detailed, complex research and analysis. During research a few projects will be analyzed using developed tool. Then, based on the results and additional data analysis, it will be possible to answer the research questions and make judgment about influence of software repository features.

As a result of our research, three types of results (according to Mary Shaw [5]) will be obtained. There are:

1. A procedure for software quality analysis via MSR during software evolution. This procedure will describe and define the whole process of the analysis started from the cloning of the repository to the results obtaining and analysis phase. It will prove that such type of analysis is possible and allow future researchers to easily reuse such procedure to make their own research and build even more complicated analysis.

2. Answer / judgment about:
   a. The correlation between software quality and software repository features. It will allow us to understand how this features influence software quality. Should software developers pay attention to them or they are so small that could be neglected? Results of the conducted research will provide detailed answer and justification about that.
   b. The most influential repository features. That is also very important to understand, which features have biggest impact on the software quality. It will allow developers to prevent future issues about the lower software quality and pay attention to the most significant features.
3. The implemented tool for multi-version software quality analysis through MSR. This is one of the most valuable contributions of this thesis. It will allow future researchers and us to use defined procedure to analyze software quality via MSR during software evolution. This is one of the most difficult parts of the thesis, given that it will require implementation of the whole flow, from cloning the Git repository, extracting software repository features, running software quality analysis, to collecting all the data and generating the XLSX reports. At the end this tool could also be used for:
   a. Decision making by project managers in the IT organizations.
   b. Projects and development processes analysis.
   c. Costs optimization.

1.4 Report Structure
This thesis is organized as follows. Chapter 2 describes the background to the problem area of the quality analysis and mining software repositories. It also contains detailed descriptions of the different types of the source code repositories, version control systems, quality KPI’s and metrics, and existing research in the area. The next chapter describes the scientific approach, selected research method, and ethical considerations. Chapter 4 contains detailed description of the implemented tool, which is necessary to conduct multi-version software quality analysis. Chapter 5 presents everything that is related to the experiments, measurements, data analysis and visualization. It also contains findings and results of the experiments. Last chapter summarizes all the results and presents final conclusion of the thesis work. Finally, we also discuss future work and improvements over the current version of our tool.
2 Background
This chapter covers general and necessary definitions and theoretical knowledge about Software Repositories, Version Control Systems, Mining Software Repositories and Software Quality metrics and tools.

2.1 Software repositories
In information technology and software engineering a repository is a central place, where in an organized way data is stored and maintained. The term *repositorium* is from the Latin. It means a vessel or chamber in which things can be placed. Another meaning is a place where things are collected.

A repository could be directly accessible to users or may be a place for specific databases, files, or documents, which can be obtained for further relocation or distribution in a network [6]. In software development, a repository is a central place to store files. It is managed by VCS to keep multiple versions of files. Usually repositories are stored on a server, where multiple users can access them, but repository can also be configured on a local machine for a single user [6].

Each repository has three main elements – a trunk, branches, and tags. Current version of a software project is stored in the trunk. It could be multiple source code files, necessary resources, etc. Branches are used for development of the new features and therefore contain new versions of the program. Developers create a new branch each time they are doing major revisions to the program. When all the necessary changes have been made, a branch can be merged into the trunk as the latest version. Otherwise, it could be discontinued, if it contains unwanted changes [6]. Tags are not related to an active development and usually used to save versions of a project. For example, each time a new version of the software is ready to be released, developers may create a tag for it.

A repository provides a structured and organized way for developers to store source code files. It could be helpful for any project size, but it is especially important for large ones [6]. Any changes to a project are made through a commit. If recent updates produce bugs or other issues, it is also possible to quickly revert to a previous state of a project.

A lot of VCS also support comparison of different versions of files. It could be helpful during source code investigation. Additionally, if a repository is stored on a server, it is possible to make a check out of all the files for editing and in such way prevent them from being edited by more than one user at a time [6].

2.2 Version Control Systems (VCS)
Version Control System or VCS is a software tool that keeps track of any changes to a file or set of files in a special kind of database. It helps manage changes over time and allow access to a specific version on demand [7]. Version control is a component of software configuration management. It is also known as source
control or revision control. It corresponds to the management of modifications to source code, documentation, web sites, etc. Usually, changes are identified by "revision level", "revision number", or just "revision" [8].

If you are a software developer, who is writing code and want to have every version of the project or selected files then VCS is a most helpful tool in that case. [9] describes that it allows to:

- Revert changes in specific files back to one of the previous states,
- Revert the entire project,
- Compare changes in files from different versions,
- See, who modified specific piece of code.

VCS also allows to recover files if they for some reason were screwed up or lost.

Each version control system has features. Features are some interesting information that can be extracted and analyzed for the different purposes. For example below are the some possible features of the source code repositories:

- Software versions (minor and major),
- Code changes (added, edited, removed lines),
- Count of changed files,
- Commit messages,
- Meta data:
  - Frequency of commits,
  - Date of commit,
  - Author of commit.

2.2.1 Types of VCS systems

There are three types of Version Control Systems – Local Version Control Systems, Centralized Version Control Systems and Distributed Version Control Systems. Let us examine all of them.

1) **Local Version Control Systems.** One of the common approaches, if you need simple version-control, is to just copy files into a new time-stamped directory. If that were the case, a great amount of problems could appear in the future, namely it would be highly probable that needed directory will be forgotten or lost, and some of the files will be broken by a wrong records.

To resolve such problems, developers created local Version Control Systems, which allow keeping all the changes under version control and store them in a database [9]. The figure 2.2 represents local version control system.

One of the most popular tools for that was a system called RCS. Even now it is still distributed with many computers. The RCS approach is based on patch sets. Each of such sets represents differences between files. RCS keeps them on disk in a special format. On demand it can easily re-create state of any file at specific time in the past by applying all the patches [9].
2) **Centralized Version Control Systems.** The next step in development of the VCSs is the Centralized Version Control Systems (CVCSs). They were created to solve collaboration problems, when developers want to work together on the same project. An example of such systems could be CVS (Concurrent Versions System), Subversion (SVN), Perforce, etc. [9]. The figure 2.3 represents centralized version control system.

Main concept of CVCSs is that all of them have one server and (could have) many clients. Server stores all the versioned files and clients can check out them. Centralized Version Control Systems were a standard for version control during many years.

With comparison to the local VCSs, this way of handling version control offers many advantages. For example it is possible to check on what each of the developers is working on. Administrators could manage access and permissions. In general, it is much easier to administrate such systems rather than work with local databases on each client.
However, centralized VCSs also have a few problems. One and maybe the most evident is that all the data is stored only on the centralized server. That means that if the server goes down for some period of time, then nobody can save their changes and collaborate at all. Furthermore, if the hard disk of central server will become corrupted, then all the data will be lost except local snapshots, which developers may have. The same situation with local VCS – "whenever you have the entire history of the project in a single place, you risk losing everything" [9].

3) **Distributed Version Control Systems** (DVCSs), such as Git, Mercurial or Bazaar, work in another way. Developers fully mirror the repository, including the full history, instead of checking out the latest snapshot of the files. Therefore, given that each clone is a full copy of all the data, there is no danger if any of the central servers dies. Any of the client repositories can be pushed to the server to restore it [9]. The figure below represents distributed version control system.

![Distributed version control system](image)

**Figure 2.4: Distributed version control system**

Besides that, Distributed Version Control Systems also can work pretty well with several remote repositories. It allows developers simultaneously collaborate in different ways with different groups of people on the same project. It also possible to set up several types of workflows, such as hierarchical models, that is not possible in VCSs [9].
2.2.2 Version Control Software tools

Let us examine the major and most popular Version Control Software tools, which are currently in use. All of them are very helpful in every-day work and are necessary to organize the multi-developer projects [8].

1) **Git**. One of the most popular and de facto standard version control system in the world currently is Git [10]. Git is a VCS for tracking changes in different types of files and synchronizing the work of multiple people on those files. It is basically used in software development to manage source code, but can be also used in any set of files to keep track of changes [11]. Git was created in 2005 by Linus Torvalds to simplify the development of the Linux kernel.

Git has a distributed architecture and therefore it is a good example of a Distributed Version Control System. Unlike most client–server systems, like CVS or Subversion, where the full version history is stored only in one single place (central server), in Git, each developer can have on his computer full-fledged repository with complete history of all changes. All versions tracking functionality is available even without access to network or a central server [11].

Pros: full history tree available offline; distributed, peer-to-peer model; cheap branch operations; increase in operation speed [12]. Cons: learning is required for those, who use Subversion; not optimal for single developers [12].

There are a lot of different web-based services, which uses Git and provide possibility to use Git as a service. Most popular of them are GitHub, GitLab and Bitbucket. Let us examine these services in details.

**GitHub** – is a web-based hosting service for version control that uses Git. Basically used by software developers for programming code. GitHub offers source code management (SCM) and distributed version control functionality of Git [13]. Besides that it has a lot of additional features, which were developed above the main Git functionality. Nowadays GitHub is one of the most powerful web-based Git services. It provides:

- Access control,
- Collaboration features, such as comments,
- Pull requests,
- Bug tracking and feature requests,
- Task management,
- Wikis.

People usually use GitHub for open-source software projects. Year ago it was around 57 million repositories created by more than 20 million users. It makes GitHub the largest source code host in the world [13].

**GitLab** – is a web-based open source Git repository manager and one of the main competitors to the GitHub. Nowadays it has more than 1400 open source contributors. Main part of code is written on Ruby and Go [14].

GitLab was developed by Dmitriy Zaporozhets and Valery Sizov from the Ukraine. Main goal was to create free open source service, which can compete with GitHub, provide high level of usability and be free. In additional to the main
functionality, it has wiki, issue tracking and almost all of the GitHub features, mentioned above, plus a few own developed by themselves.

**Bitbucket** – is a web-based version control repository hosting service for source code and development projects owned by Atlassian (being acquired in 2010). Previously it was using Mercurial as a revision control system. Nowadays it uses Git. One of the advantages of Bitbucket is that it is deeply integrated with other Atlassian software like Jira, Confluence, Bamboo, etc. [15]

The whole solution is very similar to GitHub and GitLab. It also offers both free accounts and commercial plans [15]. The main target audience for Bitbucket is professional software developers with private proprietary software code. On September 2016 it was more than 5 million developers and 900,000 teams in Bitbucket.

2) **Apache Subversion** (SVN) was created as an alternative to CVS and takes the best its features. Like CVS, SVN is also free and open source solution. To prevent repository database from being corrupted, SVN uses the atomic operations concept. It means that either all of the changes made to the source are applied or none are applied and therefore no partial changes will break the original source [12].

SVN has resolved one of the biggest problems of CVS – high costs of branch operations. It easily supports large, forked projects with many directions [12]. At the same time SVN also has a few issues – slower comparative speed and the lack of distributed revision control. It uses a peer-to-peer model rather than using a centralized server to store code updates. Also, if the server is down, no clients are able to access the code. It is the main downside of a dedicated server approach. Below there are lists of pros and cons of SVN:

**Pros:**
- Includes atomic operations,
- Cheaper branch operations,
- A lot of different plug-ins for IDEs,
- Based on CVS.

**Cons:**
- Insufficient amount of the repository management commands,
- Contains bugs, which are related to renaming files and directories,
- Slow comparative speed.

3) **Mercurial** is a “distributed revision-control tool for software developers. It is supported on Microsoft Windows and Unix-like systems, such as FreeBSD, macOS and Linux [16].” In [16] we can find a short description about design advantages and user experience with Mercurial:

“Mercurial's major design goals include high performance and scalability, decentralized, fully distributed collaborative development, robust handling of both plain text and binary files, and advanced branching and merging capabilities, while remaining conceptually simple. It includes an integrated web-interface.

Mercurial has also taken steps to ease the transition for users of other version control systems, particularly Subversion. Mercurial is primarily a
command-line driven program, but graphical user interface extensions are available, e.g. TortoiseHg, and several IDEs offer support for version control with Mercurial. All of Mercurial's operations are invoked as arguments to its driver program hg (a reference to Hg – the chemical symbol of the element mercury).

4) GNU Bazaar is a distributed and client–server revision control system. Bazaar can be used by teams collaborating across a network or by single developer working on multiple branches of local content [17]. Bazaar has very similar commands to CVS or Subversion ones. That makes it easy to learn and understand.

With Bazaar it is possible to create and start new project even without a remote repository server. Everything that is needed is just to invoke init command in a directory, which should be under version control [17]. Bazaar can work with and without a central server, rather than regular distributed VCS, which doesn’t support central servers. It is even possible to use both methods with the same project at the same time [17].

Bazaar also allows working with other revision control systems, such as Subversion, Mercurial, Git. Developers can check-out a new branch from another system, make changes locally and commit them to a Bazaar branch. Then later on merge them back into the main (source) system. This possibility is available for Subversion. Git and Mercurial has read-only access [17].

2.3 Mining software repositories (MSR)
Mining software repositories (MSR) is a software engineering field where software researchers applying data mining techniques analyzes rich data in software repositories to extract useful and actionable information about software systems, projects and software engineering produced by developers during the development process [18][1]. MSR is a broad class of investigations into the examination of software repositories [2].

MSR is about evolving software versions, changes, data (source code, bug reports, etc.), metadata (who, when, where, why). During mining software repositories, the obtained information, according to [18], can be used to “discover hidden patterns and trends, support development activities, maintain existing systems or to improve decision-making for the future software development and evolution. Usually the data is used to better manage software and to produce higher-quality software systems by analyzing past software development projects.”

Software repositories comprise of different types, such as source code repositories, bug repositories, mailing lists, wiki pages, etc. Mining software repositories includes data analysis, data measurement, data mining, data recovery, static source analysis, statistical analysis (correlation) [2]. The figure below presents the general flow of mining software repositories:
To start looking for insights first of all we need source raw data in which search can be conducted. The most interesting are the historical data. It could be found in different systems, depends on the type of software repository:

- For VCS systems data could be extracted from CVS, SVN, Git, Mercurial.
- For bug trackers – from Bugzilla, JIRA, YouTrack.
- For Communication systems – from e-mails, chat logs, wiki pages.

Next step is the data mining process on which, using special tools, all the necessary information, such as number of commits, number of changes, author of commit, etc. (in case of source code repositories), will be extracted and organized in understandable way.

Finally, in the last stage all the retrieved information will be analyzed, either manually or automatically, using special visualization and data mining tools. In both cases researchers will have an opportunity to find new useful information and hidden insights. Results of the analysis could be used in wide range of different areas. Most popular of them are:

- Quality assurance,
- Architecture analysis,
- Bug prediction,
- Developer feedback.

### 2.4 Software quality metrics and KPIs

In order to attain our goal of the research it is necessary to conduct the software quality measurement. To do that, 21 quality metrics have been used. The most important of them are presented on the figure below.

Metrics are classified as supporting mainly statements on software Complexity, software Architecture and Structure, and software Design and Coding [19]. Below are the list and the figure of the quality metrics.

1. **Complexity:**
   - **Size:**
     - Lines of Code (LOC)
   - **Interface Complexity:**
     - Number of Attributes and Methods (SIZE)
     - Number Of local Methods (NOM)
   - **Structural Complexity:**
     - McCabe Cyclomatic Complexity (CC)
2. Architecture and Structure:
   - Inheritance:
     o Depth of Inheritance Tree (DIT)
     o Number Of Children (NOC)
   - Coupling:
     o Coupling Between Objects (CBO)
     o Data Abstraction Coupling (DAC)
     o Message Passing Coupling (MPC)
   - Cohesion:
     o Lack of Cohesion in Methods (LCOM)
     o Improvement of LCOM (ILCOM)
     o Tight Class Cohesion (TCC)

3. Design Guidelines and Code Conventions:
   - Documentation:
     o Lack Of Documentation (LOD)
   - Code Conventions

Let us deeply examine each of the quality metrics [19], which will be used on analysis stage. Lines of code simply counts the lines of source code (line break characters) of a certain software entity. It is a simple yet powerful metric to assess the complexity of software entities. Since it is depending on code conventions and format, it is critical to use it in generated codes since it may lack of line breaks. Additionally it can only be measured in the source code itself from the front-end and is therefore a front-end side metric.
Number of Attributes and Methods simply counts the number of attributes and methods of a class. It is an object-oriented metric that can be applied to modular languages by considering the number of variables (globally visible in a module) and its number of functions and procedures.

Number of local Methods measures the number of methods locally declared in a class. Inherited methods are not considered. It is the size of the interface of a class and allows conclusions on its complexity.

McCabe Cyclomatic Complexity is a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.

A weighted sum of methods implemented within a class. It is parameterized by a way to compute the weight of each method. Possible weight metrics are:

- McCabe Cyclomatic Complexity,
- Lines of Code,
- 1 (unweighted WMC).

This variant of WMC uses McCabe Cyclomatic Complexity metric for calculating the weight for each method. Originally defined as an object-oriented metric, it can easily adapted to non-object-oriented systems computing the weighted sum of functions implemented within a module or file. Response for a class is a count of (public) methods in a class and methods directly called by these. RFC is only applicable to object-oriented systems.

Depth of Inheritance Tree (DIT) is the maximum length of a path from a class to a root class in the inheritance structure of a system. DIT measures how many super-classes can affect a class. DIT is only applicable to object-oriented systems.

Number Of Children (NOC) is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. NOC measures how many classes inherit directly methods or fields from a super-class. NOC is only applicable to object-oriented systems.

Coupling Between Objects (CBO) is the number of other classes that a class is coupled to. CBO is only applicable to object-oriented systems. The Data Abstraction Coupling (DAC) measures the coupling complexity caused by Abstract Data Types (ADTs). This metric is concerned with the coupling between classes representing a major aspect of the object oriented design, since the reuse degree, the maintenance and testing effort for a class are decisively influenced by the coupling level between classes. Basically same as DAC, but coupling limited to type references.

The Message Passing Coupling (MPC) measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes. It allows for conclusions on the message passing (method calls) between objects of the involved classes. This allows for conclusions on re-useability, maintenance and testing effort.
The Lack of Cohesion in Methods metric is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion. It represents the difference between the number of method pairs not having instance variables in common, and the number of method pairs having common instance variables.

The Improvement of LCOM (cf. Lack of Cohesion in Methods) metric is a measure for the number of connected components in a class. Components are methods of a class sharing (being connected by) instance variables of the class. The less separate components there are the higher is the cohesion of the methods in the class.

The Tight Class Cohesion metric measures the cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a low cohesion indicate errors in the design.

Lack Of Documentation shows how many comments are lacking in a class, considering one class comment and a comment per method as optimum. Structure and content of the comments are ignored.

2.5 Quality analysis tools
This section describes one of the quality measurement frameworks – VizzAnalyzer™, which is used during the software quality analysis.

2.5.1 VizzAnalyzer™
According to [20], VizzAnalyzer™ is a “framework designed to support maintenance and reengineering of software. It allows to integrate arbitrary reverse-engineering tools, i.e. tools for program analysis and/or visualization. Then the VizzAnalyzer™ can interactivelly and iteratively retrieve program information, focus, analyze, and visualize it.”

The software architecture of the framework consists of three main components, each implemented as an extension point where user defined algorithms can be added [21] (see figure 2.7):

- A low/binary-level analysis engine (LLA).
- A high-level analysis and metrics engine (HLA).
- A visualisation engine (VE).

According to [21], let us understand what each of the components does. “The LLA takes source and binary code, and produces model of the program under investigation. The program model is captured in form of one or more structural graphs. These graphs are configurable, according to the needs of the HLA.”

“The HLA works on structural graphs. It adds additional information captured in either further graphs or in annotations of existing ones. Special algorithms of the HLA compute annotations that are interpretable as distance metrics between nodes of the graphs. In order to draw a graph it needs to have at least one distance metric computed.”
“The VE computes a layout of a graph containing distance information and depicts the graph. It provides a user interface allowing for zooming, rotating and aggregating the depicted graphs.”

Figure 2.7: General architecture of the VizzAnalyzer™ framework

2.6 Existing research

Area of the software quality analysis is quite large and this field is popular at this time among researchers and developers. This leads to the presence of a large number of works and already conducted research in the field. Unfortunately, there are not too many works, which use MSR process. The most interesting and related research is presented below.

In [22] authors have described the influence of identifier names on code quality. They assume that poor quality identifier names are the cause of the low quality source code that translates into low quality software. To evaluate quality of source code cyclomatic complexity metric and maintainability index are used. Also a readability metric is applied to evaluate the readability of methods.

Authors employ several research methods which include data collection and statistical Analysis. Also, a tool for automatic extraction and analysis of identifiers from Java source code was developed. Using statistical analysis, they found that flawed identifiers in Java classes were associated with low quality source code. They showed that the association also presents at the higher level of granularity of Java methods and adopt techniques used in medicine to evaluate diagnostic tests in order to show, which identifier naming flaws could be used as a diagnostic of problematic Java source code.

In the above research authors employ only several software quality metrics. In that case general value of quality will not be a full-fledged. It will lead to distortion of results. In my work this shortcoming will be eliminated by use of 8 key performance indicators with 21 quality metrics.

In [23] D. Nemer analyzes the differences of two subsequent software versions in the source code and during execution by demonstrating the impact of
the source code changes on the software run. This impact helps developers understand what will be affected in the application during execution due to their changes in the source code. The approach, which allows predicting what will be impacted in the future, and the corresponding tool called IMPEX were created and presented in this paper.

[23] has the same shortcoming like previous one. To determine impact the author uses specific metrics but in that case user of this system will not know real state of the project after changes, because he will get incomplete info about software quality.

Analysis about structural software changes between software versions (during software’s evolution) is described in [24]. Authors observe changes made close to the software releases and employ a data collection research method, in particular observation and measurement, to measure code churn and structural source code changes.

During their research authors detected basic changes (e.g. field, property and method changes) and complex changes (e.g. beyond class and move changes) that indicate on a refactoring processes. Also authors found that near the release dates the number of structural source code changes significantly increases.

In this work there is no software quality analysis and calculation of correlation between structural source code changes and software quality (e.g. value of KPIs, number of bugs, etc.). However, it is listed in the future author’s goals as well as uses of a machine learning algorithms to predict software quality.

At the moment all of the above because of absence can be considered as shortcomings. In this work they will be eliminated by calculating correlation between software quality and software repository features and creating predictive model based on observed data about software repository features and software quality.

In [25] Bart Du Bois and Tom Mens describe the impact of refactoring on internal program quality. The authors’ goal is to develop tools and techniques for software developers to help improve program quality in the refactoring stage. In this paper authors propose a formalism to describe the impact of program structure refactoring. The main contribution of this work is to show the quality drift caused by the code refactoring. Finally K. S. Herzig describes a related topic in [26] but with a focus on the long-term impact of changes on the software quality.

Existing works in the area of software quality were analyzed and it was found that all of them describe software quality analysis without a connection to the software repository features. Thus, they lose important information that can be used for preventing bugs, decision-making and optimizing development processes.
3 Method

This chapter contains detailed description about research methods and ways of their usage in this study.

3.1 Scientific Approach

To answer the defined research questions, a quantitative strategy and hypothesis testing will be used. Hypothesis testing is the scientific method, which consists of three components:

- Collection of data through observation and experimentation,
- Formulation hypotheses,
- Testing of hypotheses.

This method uses statistics to determine the probability that a given hypothesis is true [27]. It refers to statistically analyzing sample data with aim of making assumptions about a population. Nowadays there are three testing approaches that are briefly described below [27].

1. P-values. P-values evaluate how well the sample data support the argument that the null hypothesis is true. It measures how compatible data are with the null hypothesis.

2. Estimation statistics. Estimation is the process, which allows making inferences about a population, based on information obtained from sample data [28]. An estimate of a population parameter may be expressed in two ways: point estimate and interval estimate.

3. Bayes factor. In statistics, the use of Bayes factors is a Bayesian alternative to classical hypothesis testing [29][30]. Bayesian model comparison is a method of model selection based on Bayes factors.

Let us examine the algorithm of the P-values approach [27].

1. Formulate hypothesis:
   - Null hypothesis, denoted by $H_0$, is the hypothesis that sample observations are the result of pure chance;
   - Alternative or experimental hypothesis, denoted by $H_1$ or $H_a$, is the hypothesis that sample observations are influenced by some non-random cause (observations are the result of a real effect).

2. Identify a test statistic using the sample data and assuming that the $H_0$ is true.

3. Calculate the $P$-value, which is the probability that a test statistic can be obtained at least as extreme as the observed one, assuming the truth of the $H_0$. The smaller the $P$-value, the stronger the evidence against the $H_0$.

   Compare the $P$-value to $\alpha$ (acceptable significance value). If $P \leq \alpha$ then the $H_0$ is rejected, the $H_1$ is valid, and the observed effect is statistically significant. Otherwise, $H_0$ is not ruled out.

   To have an ability to test the hypotheses the necessary amount of data must be collected. To do that a controlled experiment method will be used. The controlled experiment is a "highly focused way of collecting data and is especially
useful for determining patterns of cause and effect” [31]. Such type of experiments compares the results obtained from experimental and control groups. There are usually the same except for the one aspect in control group – the independent variable, whose effect is being tested [31].

An experimental and a control group are needed to conduct a controlled experiment. The experimental group is a “group of individuals that are exposed to the factor being examined” [31]. The control group is not exposed to the factor. All another external influences should be constant, i.e. every other factor or influence between the experimental group and the control group needs to stay the same. The only thing that could be different between the two groups is the factor being researched [31].

Important part of the experiments is variables. A variable is any factor that can be controlled, changed, or measured during the experiment. There are several types of variables in scientific experiments. The most commonly used ones are independent and dependent variables, usually plotted on a chart or graph [31]. In the science experiment independent variable is the only one variable that changes to test how it influences the dependent variable. The dependent variable is the variable that should be observed and measured. It is dependent because “it is the factor that is dependent on the state of the independent variable” [31].

In a controlled experiment everything except independent variable is held constant. A set of data is used as the control group, which is “commonly the normal or usual state, and one or more other groups are examined, where all conditions are identical to the control group except one variable” [31].

One of the advantages of a controlled experiment is that it is possible to eliminate much of the uncertainty in the results. A controlled experiment provides a high degree of confidence in the outcome, but in case if it is impossible to control each variable, confused outcome is inevitable [31].

Controlled experiments have both strengths and weaknesses. One of the strengths is that “results can establish causation, i.e. they can determine cause and effect between variables” [31]. On the other side, controlled experiments can be artificial, i.e. “they are done, for the most part, in a manufactured laboratory setting and therefore tend to eliminate many real-life effects. As a result, analysis of a controlled experiment must include judgments about how much the artificial setting has affected the results.” [31]

3.2 Method Description
In the section 3.1 two research methods: hypothesis testing and controlled experiment were defined to conduct the study and answer the research questions. Given that we have made a few assumptions at the beginning, most popular approach how to prove or disapprove them is the hypothesis testing. At the same time, one of the most suitable ways, how to collect the data for that is the controlled experiment. This section describes how the research methods are applied in this project.
According to defined research questions, the main purpose of the study is to determine whether the features of a software repository influence the software quality during software evolution. In other words, find the correlation between software repository features and software quality. And if such correlation exists, then determine which of the repository features have the greatest impact on the software quality.

To conduct controlled experiments the independent and dependent variables should be defined. In case when we want to measure software quality with dependency of software repository features, software quality is a dependent variable, and VCS features are independent variables. Features are the some interesting information that can be extracted and analyzed for the different purposes. In this study a few source code repository features will be used:

- Source code changes (added, changed, removed lines of code),
- Total amount of source code changes in one commit,
- File changes (added, changed, removed files),
- Total amount of changed files in one commit,
- Length of commit message,
- Meta data:
  - Date of commit,
  - Author of commit.

To achieve the objective of the research it is necessary to conduct the software quality measurement. It will be measured for each selected feature. For example, in case of using file changes, software quality should be measured for each commit in the project. For the software quality measurement 7 key performance indicators with 21 quality metrics will be used (see sections 1.1.1 and 2.4). Below is the list of most interesting KPI’s:

- Cloning,
- Cohesion,
- Complexity,
- Coupling,
- Inheritance hierarchy,
- Readability,
- Size.

To get summarized quality value, all KPI's will be aggregated into one average value of all key performance indicators. This value will be used as a dependent variable in the controlled experiments.

All the necessary data will be obtained using custom created tool (see chapter 4). The tool will perform three main functions:

- Measurement of all the quality metrics and KPIs. For that purpose VizzAnalyzer™ will be used (see section 2.5.1). It is a Java library, which allows to conduct software quality analysis.
- Extraction of the source code repository features. It will be conducted using custom created module in the tool (see chapter 4).
• Generation of the XLSX reports, which will contain all the information about quality analysis and source code repository features.

When all the necessary information will be obtained and aggregated into one document, actual research can be started. To build the correlation between software quality and source code repository features in this study, Microsoft Power BI will be used (see chapter 4).

Measurements and experiments will be conducted for a few Java, Maven based, projects. Java is one of the TOP 3 most popular programming languages nowadays and at the same time Maven is the leading build and dependency management system for Java projects.

3.3 Ethical Considerations

Firstly, all the operations with source code from open source projects should be conducted anonymously. Owners and developers of the projects will not be disclosed in order to avoid damage to their reputation in case of low quality of the source code of their projects.

All the data, results, methods and procedures should be honestly reported. Also data should not be fabricated, falsified or misrepresented. And last but not least, it is necessary to strive for objectivity, avoid bias in, data analysis, data interpretation, peer review, expert testimony, and other aspects of research where objectivity is expected or required.
4 Implementation

This chapter contains technical and implementation details about the developed software to conduct the analysis. It covers use cases and functional/non-functional requirements to the software; architecture and system components description; overview of the used tools and programming languages; proposed approach for the data analysis and visualization; and user guide for usage and future development.

4.1 General overview and requirements

As was described previously, to conduct the study a lot of data should be collected about software quality and source code repository features. Given that obtain such amount of information manually is a hard and very time consuming task, it was decided to implement the tool, which can do all this job automatically and in as simple as possible way, so then future researchers can easily reuse or even extend functionality of this tool for different purposes.

There are several key features, which are implemented in this tool:

1. Clone Git repository from remote URL to a local machine.
2. Extract list of all commits in a specified branch.
3. Automatically build the project with dependencies using Maven.
4. Run software quality analysis for specific commit.
5. Extract repository features for specific commit.
6. Aggregate obtained information and generate XLSX report.

Tool is available in two different options: as an executable JAR file, so then researchers without programming knowledge can easily use it through command line, and as a Java library, which can be imported into existing Java project and usable. To extend the existing possibilities of the tool, developers can fork Git repository on GitHub [32] or submit pull requests to the existing one.

4.1.1 Use cases

For the end user part of the key features, described above, are hidden. Below there is a list of actions (use cases), which are available for the end-user. Use case diagram is presented on the figure 4.1.

1. Enter input data for quality analysis: remote URL, name and owner name of the Git repository.
2. Specify one of the repository branch, from which commits will be analyzed.
3. Specify number of commits, which should be analyzed.
4. Specify offset of commits. If necessary, for example, analysis could start from 10th commit from the top.
5. Get generated XLSX report with analysis output.
4.1.2 Functional and non-functional requirements

Below are presented functional (table 4.1) and non-functional (table 4.2) requirements to the software.

Table 4.1 – Functional requirements

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1</td>
<td>Tool should clone Git repository from remote URL to a local machine.</td>
</tr>
<tr>
<td>FR2</td>
<td>Repository should be stored in a temp folder during the analysis.</td>
</tr>
<tr>
<td>FR3</td>
<td>Repository should be removed after the analysis.</td>
</tr>
<tr>
<td>FR4</td>
<td>Tool should extract list of all commits in a specified branch.</td>
</tr>
<tr>
<td>FR5</td>
<td>Tool should automatically build the project with dependencies using Maven.</td>
</tr>
<tr>
<td>FR6</td>
<td>Tool should run software quality analysis for specific commit.</td>
</tr>
<tr>
<td>FR7</td>
<td>Tool should extract repository features for specific commit.</td>
</tr>
<tr>
<td>FR8</td>
<td>Tool should aggregate obtained information and generate XLSX report.</td>
</tr>
<tr>
<td>FR9</td>
<td>Software should be distributed as an executable JAR file.</td>
</tr>
<tr>
<td>FR10</td>
<td>Software should also be available as a Java library.</td>
</tr>
</tbody>
</table>

Table 4.2 – Non-functional requirements

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR1</td>
<td>It should be possible to easily start using tool for people who don't have any programming knowledge.</td>
</tr>
<tr>
<td>NR2</td>
<td>Tool should be available in two versions: executable JAR file, which could be used from CLI by different researchers; and as a Java library,</td>
</tr>
</tbody>
</table>
which could be imported into another project.

NR3 Tool must be designed in such way that it is possible to easily expand the functionality (e.g., add connector to another VCS or replace VizzAnalyzer™ by some different framework).

NR4 Quality analysis algorithm should be completed in a reasonable time.

NR5 Source code must be well commented.

4.2 Software Architecture

This section presents general overview of the software architecture of the developed tool for quality analysis. It includes descriptions of the main components, high-level diagram and class diagram to get more detailed and technical understanding of the system.

4.2.1 Components

The software consists of four main components (services), which cover all the functionality, presented above. The services are: Git Service, Maven Service, Analysis Service and Report Service. The high level overview of the developed software architecture and its components could be presented as on the figure below:

1) **Git Service.** It handles basic Git operations (e.g. clone repository, checkout specific branch/commit, etc.), which are necessary to manipulate Git repositories in order to get the information about source code repository features and conduct the analysis in general.
Also, Git Service works together with connectors to web-based Git hosting services like GitHub, GitLab and Bitbucket. Such connectors are needed to obtain detailed information about each commit in the repository:

- How many lines of code were added/changed/removed,
- How many files were added/changed or removed,
- Who did the changes,
- When did the changes,
- How long is the commit message,
- How many commits another developers have left to this commit in the GitHub/GitLab/Bitbucket,
- Total count of all the changed files and lines of code.

The implementation is based on the Eclipse JGit. It is a free library for Java, which provide basic functionality to manipulate Git repositories. This study and software covers only GitHub connector. Another connectors could be easily implemented with purpose of the software extension.

2) **Maven Service** is responsible for build flow of the source project, which should be analyzed. To conduct good quality analysis, each source project should be builded with dependencies beforehand. In this case values of quality metrics and KPIs will be more accurate.

This service does Maven build automatically and directly after the Git Service has checkout necessary commit. Implementation is based on the free Java library from Apache – Maven Invoker. It provides Maven functionality through Java. Detailed information about Apache Maven could be found in the section 4.1.2.

3) **Analysis Service.** It is a main service, which conduct software quality analysis and calculate all the quality metrics and KPI’s (see sections 1.1.1 and 2.4). It is based on the VizzAnalyzer Java library, which was described in the section 2.5.1.

4) **Report Service.** When analysis is completed, it is time to summarize all the results and present them to the researcher. To make it as easy as possible for them it was decided to generate XLSX document at the end of analysis and present this document to end-users. Then it will be much easier to analyze and work with output data.

XLSX document has defined structure. In general it contains two main sections: results of software quality analysis and results of source code repository features extraction. Below is a list of all the data in the output XLSX report:

- Software quality analysis:
  - Cohesion,
  - Size,
  - Complexity,
  - Hierarchy,
  - Coupling,
  - Cloning,
  - Readability,
- Total quality.
- Repository features extraction:
  - Commit hash,
  - Date of the commit,
  - Commit author’s email,
  - Commit author’s name,
  - Length of the commit message,
  - Amount of comments to the commit on GitHub,
  - Amount of additions,
  - Amount of deletions,
  - Total amount of changes,
  - Added files,
  - Changed files,
  - Deleted files,
  - Total amount of changed files.

To have more detailed technical overview of the system, below there is a class diagram of the developed software:

![Class diagram](image)

Figure 4.3: Class diagram
4.3 Programming Language and Tools
This section describes main tools and programming languages, which were used to develop the software for quality analysis.

4.3.1 Java
One of the most popular programming languages in 2017, according to GitHub [33], is Java (see figure below). Java is also very popular at the Universities. Students use it quite often for educational and thesis projects. This tool also was developed using Java.

![Number of pull requests, created on GitHub during 2017](image)

Figure 4.4: Number of pull requests, created on GitHub during 2017

According to the next reasons, it was decided to implement the software using the Java programming language:

1. Java has a lot of freely-available third-party libraries, which can significantly extend base language functionality, encapsulate a lot of complex tasks (e.g. creating web-applications based on Java EE with comparison to Spring Boot, etc.) and help developers a lot to do trivial things faster and keep focus only on the most important tasks.

2. Third-party library for software quality measurement – VizzAnalyzzer™ has been written on Java. It means that it could be imported and usable only in the Java based projects.

3. Focus of the software quality analysis in this thesis project is the analysis of Java based projects.
4.3.2 Apache Maven
To automate builds and manage dependencies during the development it was decided to use Apache Maven. According to Apache, Maven is a “software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.” [34].

Maven has a lot of advantages with comparison to another build and dependency management systems. Some of them [35][36] are listed below:

- Maven makes the build process easy,
- Provides a uniform build system,
- Has pre-defined targets to perform a few certain tasks (e.g. compilation, packaging, etc.),
- Allow downloading dependencies (external JAR files) over the network,
- Has a lifecycle with several pre-defined phases, such as compile, package, install, deploy, etc.

Like Java, Maven is also very popular and easy to use. Given that VizzAnalyzzer™ was written on Java and could be accessible through Softwerk AB Maven repository, it was decided to use Maven as a build tool and dependency management system for this thesis project. In that case it is much easier to conduct the development.

4.3.3 Git
To ensure the safety of the project source code, all the files and changes, it was decided to use a version control system. The most popular and easy to use VCS nowadays is Git. To get more details about VCSs and Git see sections 2.2 and 2.2.2. In addition to Git, source code of the project is stored in the GitHub [32] (see section 2.2.2). It prevents information loss and allows constant access to the code.

4.4 Data analysis and visualization
There are a lot of different visualization options, which could be used. In this thesis project it was decided to not implement custom visualization tool and use one of the existing ones. Nowadays there are a few most popular visualization and data analysis tools, such as:

- Qlick products: QlickView, QlickSence, QlickSence Cloud,
- Tableu Software,
- Microsoft Power BI,
- SAS Visual Analytics,
- Microsoft Excel,
- D3.js.

and even more powerful:
- RapidMiner,
• R (programming language).

To visualize output data and conduct the analysis, it was decided to use Microsoft Power BI. It is fully cover all the needs for this research and at the same time easy to start and doesn’t require any specific knowledge.

Microsoft Power BI is a “business analytics service provided by Microsoft. It provides interactive visualizations with self-service business intelligence capabilities, where end users can create reports and dashboards by themselves, without having to depend on information technology staff or database administrators.” [37]

Power BI provides “cloud-based BI services, known as "Power BI Services", along with a desktop based interface, called "Power BI Desktop". It offers data warehouse capabilities including data preparation, data discovery and interactive dashboards.” [37]

4.5 User guide for usage and future development

This section contains a short manual of how to use the quality analysis tool inside another Java project. First, you need to have JAR file – quality-analyzer-1.0.jar. Then it could be imported into the project, e.g. for Maven based projects it should be in the resources folder:

```xml
<dependency>
    <groupId>se.lnu.qualityanalyzer</groupId>
    <artifactId>quality-analyzer</artifactId>
    <version>1.0</version>
    <systemPath>${project.basedir}/src/main/resources/quality-analyzer-1.0.jar</systemPath>
</dependency>
```

To conduct the analysis input data should be prepared. There are 4 mandatory parameters:

1. Name of the source code repository,
2. Repository remote URL in GitHub, GitLab or Bitbucket (currently tool supports only GitHub),
3. Repository owner (username in Git hosting service),
4. Name of the branch, from which commits will be analyzed,
5. Type of the Git hosting service: GitHub, GitLab or Bitbucket (currently tool supports only GitHub).

and two optional:

1. Number of commits, which should be analyzed,
2. Offset of commits, if analysis should start not from the top commit.

Below there is a sample piece of code of input data:

```java
String repositoryName = "commons-beanutils";
String repositoryUrl = "https://github.com/apache/commons-beanutils.git";
```
To run the analysis you should create specific type of input data, create object of the QualityAnalyzerImpl class and run the analysis. Result will be the list of output data. Each of the list elements is a data about one commit in the specified branch, which include software quality KPI’s and repository features. Sample code is listed below:

```java
CommitAnalysisInput input = new CommitAnalysisInput(
    VCSType.GIT_HUB,
    repositoryName, repositoryUrl, repositoryOwner, branchName,
    numberOfCommits, offsetOfCommits);
QualityAnalyzer analyzer = new QualityAnalyzerImpl();
List<CommitAnalysisOutput> output = analyzer.analyzeCommits(input);
```

To generate XLSX report you should use XLSXCommitReportService class. This service will automatically convert list of output data into the XLSX file and save it on your local machine. Sample code is listed below. Full sample of the source code could be found in the Appendix A.

```java
ReportService reportService = new XLSXCommitReportService();
reportService.create(input, output);
```
5 Evaluation
This chapter describes application of the developed tool to conduct set of the experiments main goal of which is to understand, how does the software repository features influence the software quality during software evolution and shows correlation between software repository features and software quality. Also, figure out, which repository features have the greatest impact on the software quality.

5.1 Experiments
To answer the research questions it is necessary to have a certain amount of data, which could be analyzed. To get those data it was decided to conduct several experiments. In case of this thesis project, experiment is a measurement of the software quality and extraction of the source code repository features with the subsequent construction of a correlation between those features and overall quality of the software. Then, when all the data are obtained it is possible to visualize and present correlation on a scatter plots.

5.1.1 Measurement approach
To measure quality of the source code and collect all the necessary source code repository features let us use developed tool. Calculation should be conducted for each project separately. It means that in general it will be 3 different analyses. For each of the projects around 100-150 commits should be analyzed. It will give us an opportunity to get the data for a certain period of time and from the different developers. At the end of each measurement XLSX report will be generated. Structure of the report could be found in the section 4.2.1.

In this thesis we want to analyze repository features influence in general without focusing on the specific projects. It gives us an ability to neutralize the impact of negative factors in particular project and produce the results, which represent different projects at the same time, including the impact of developer experience and skills. To do that on the final stage results from all experiments will be aggregated by average values, visualized and then could be analyzed for correlation between software repository features and software quality.

Given that calculation of the quality KPIs takes quite a long time, it is difficult to analyze a large number of commits on this stage. Chapter 6 describes possible options how to reduce calculation time and improve overall performance of analysis. At the same time, chosen number of commits still allows us to understand how software repository features influence software quality.
5.1.2 Source projects for measurements

To conduct the measurements let us use three different Java, Maven based projects: Tillit, Apache Common Beanutils and JUnit4. Below there is a short description about each of them.

1) **Tillit** – is the web application for storing and viewing manuals for any home appliance. It was implemented by 7 students on the 4DV611 - Agile Product Development course. Tillit has server and client parts, which are separated. Interaction between two parts is based on HTTP requests. Beck-end is a RESTful web services based on Java and was implemented with using of Spring Boot, MySQL and Liquibase. Front-end is a mobile-friendly web application with responsible user interface written with use of JQuery, HTML5 and CSS3.

2) **Apache Common BeanUtils.** “The Java language provides Reflection and Introspection APIs (java.lang.reflect and java.beans packages in the JDK Javadocs). However, these APIs can be quite complex to understand and utilize. [38]” According to [38], Apache Commons BeanUtils “provides an easy-to-use and flexible wrappers around reflection and introspection.” It has 22 active contributors and 87 forks on GitHub.

3) **JUnit4.** Authors of the JUnit in [39] describe their product as a “simple framework to write repeatable tests”. They also provide us more detailed description of the framework: “JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks, which is collectively known as xUnit that originated with SUnit.”

“A research survey performed in 2013 across 10,000 Java projects hosted on GitHub found that JUnit, (in a tie with slf4j-api), was the most commonly included external library. Each library was used by 30.7% of projects.” JUnit4 has almost 7000 stars on GitHub, 2605 forks and 139 contributors.

Chosen projects are implemented by different groups of developers with different skills and experience and due that during the experiments will give us various results. The Tillit is created by students and mainly represents skills of junior software developers. JUnit is the big and very popular library used by many developers in the world. Apache Common BeanUtils is smaller than JUnit4, but still quite popular library. All together this 3 projects represent different domains, different developers and different levels of usage. It makes them a good choice to analyze impact of software repository features on software quality.

5.2 Data analysis and visualization

On the next step it is necessary to calculate quality changes in %. This way allows us to operate with relative quality value instead of absolute and, given that, all the obtained data for three projects could be summarized and used at the same time. According to section 4.4, it was decided to use Microsoft Power BI for results visualization and data analysis.
Let us analyze influence of the repository features on the software quality: Total changed files in commit, Added files, Changed files, Deleted files, Total amount of changes in commit, Additions, Deletions, Length of the commit message.

The figure below displays scatter plot with total files and quality changes in % by commit.

![Figure 5.1: Correlation between total changed files and quality changes in %](image)

Here range of changes for quality is from -10% (decrease quality) to 32% (increase quality). Let us examine more detailed view by filtering values on axes: Total files < 40 and Changes of quality, % < 5.

![Figure 5.2: Correlation between total changed files and quality changes in %](image)
As can be seen from the plot, during the software evolution quality of the source code was changing within 4%. Red line on the plots represent trend of the correlation. On the figure above there is almost no any correlation between amount of changed files and quality of the software. Specific numeric value of the correlation could be found at the end of this section.

Let's have a deep look and build scatter plots for added, changed and deleted files separately. To get only valuable values, filters for axes are applied from the beginning.

![Figure 5.3: Correlation between a – added files, b – changed files and quality changes in %](image)

![Figure 5.4: Correlation between deleted files and quality changes in %](image)

As we can see, correlation between deleted files and changes of the quality has a positive trend. It means that, when developer removes some source code files, quality of the software increases. It could be interpreted in the following way –
each of the source code files contains quality issues. As much files project contains, as much quality issues there are. Therefore, remove of the files decreases amount of issues and increases overall software quality.

Now let us visualize amount of changes in files (added, changed, deleted lines of code) and how they influence quality of the software.

![Figure 5.5: Correlation between total changes and quality changes in %](image)

Here correlation is almost absent. More detailed plot on the figure below shows us that code changes influences the quality, but this influence is too small.

![Figure 5.6: Correlation between total changes and quality changes in %](image)
Also, below there are detailed plots for each type of the changes (additions, deletions):

![Figure 5.7: Correlation between a – additions, b – deletions and quality changes in %](image)

Both of the plots above represent the same situation as almost all another plots – correlation exists, but it is too small to talk about real influence.

Last repository feature – length of the commit message. Interesting that as long the message as better the quality of the code. The reason for such behavior could be that people, who usually spend time to write descriptive commit message, also take care about code quality and produce less quality issues or even fix existing ones.

![Figure 5.8: Correlation between message length and quality changes in %](image)
Now let us calculate correlations between repository features and total quality value for each of the three source projects. Results of the calculations (for the first 4 repository features) are presented in the table below:

**Table 5.1 – Correlation between repository features and total quality value**

<table>
<thead>
<tr>
<th>Project</th>
<th>Total files</th>
<th>Added files</th>
<th>Changed files</th>
<th>Deleted files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tillit</td>
<td>-0.062</td>
<td>-0.094</td>
<td>-0.002</td>
<td>0.031</td>
</tr>
<tr>
<td>Comm. BeanUtils</td>
<td>-0.030</td>
<td>-0.132</td>
<td>-0.029</td>
<td>0.046</td>
</tr>
<tr>
<td>JUnit4</td>
<td>-0.035</td>
<td>0.088</td>
<td>-0.105</td>
<td>0.000</td>
</tr>
<tr>
<td>Average</td>
<td>-0.042</td>
<td>-0.046</td>
<td>-0.045</td>
<td>0.025</td>
</tr>
</tbody>
</table>

Results for the rest of the repository features are presented in the table 5.2:

**Table 5.2 – Correlation between repository features and total quality value**

<table>
<thead>
<tr>
<th>Project</th>
<th>Total changes</th>
<th>Additions</th>
<th>Deletions</th>
<th>Message length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tillit</td>
<td>-0.031</td>
<td>-0.030</td>
<td>-0.023</td>
<td>0.010</td>
</tr>
<tr>
<td>Comm. BeanUtils</td>
<td>0.030</td>
<td>-0.068</td>
<td>0.116</td>
<td>0.084</td>
</tr>
<tr>
<td>JUnit4</td>
<td>-0.068</td>
<td>-0.058</td>
<td>-0.111</td>
<td>-0.034</td>
</tr>
<tr>
<td>Average</td>
<td>-0.023</td>
<td>-0.052</td>
<td>-0.006</td>
<td>0.020</td>
</tr>
</tbody>
</table>

Generally, correlation between features and software quality is from -0.1 to 0.1. That is too small to say that repository features somehow influence the quality of the software. Also, average values of the correlation are almost the same for all of the features. It means, that none of these features has more influence than another on the software quality.

### 5.3 Findings and discussion

Based on the conducted experiments and results of the data analysis, there are several major and most important findings in this research:

1. During the software evolution quality of the source code changes from -10% (decrease quality) to 30% (increase quality) on the long-term period and within 4% on the short-term one.
2. There is almost no any correlation between quality of the software and amount of added, changed and deleted files.
3. Correlation between deleted files and changes of the quality has a positive trend. It means that, when developer removes some source code files, quality of the software increases. It could be interpreted in the following way – each of the source code files contains quality issues. As much files project contains, as much quality issues there are. Therefore, remove of the files decreases amount of issues and increases overall software quality.
4. Code changes (added and removed lines of code) influences the quality, but this influence is too small and correlation is almost absent.
5. As long the commit message as better the quality of the code. The reason for such behavior could be that people, who usually spend time to write
descriptive commit message, also take care about code quality and produce less quality issues or even fix existing ones.

Also, all the research questions raised earlier can be considered as answered. Correlation between features and software quality is from -0.1 to 0.1. That is too small to say that repository features somehow influence the quality of the software. Average values of the correlation are almost the same for all of the features. It means, that none of these features has more influence than another on the software quality. Analysis tool was implemented and used to conduct all the quality KPIs and metrics measurement, and extraction of the software code repository features. It fully proves that it is possible to automate process of the multi-version software quality analysis through MSR.

Comparing this thesis to the existing works in the area of software quality analysis, we could say that authors of those works didn't use MSR approach. All works doesn't have a connection to the software repository features and, therefore, all the previously conducted analysis describe software quality only from the one, non-MSR, prospective, which could be interpreted as a weakness or incompleteness of results, because they lose important, hidden information, which could be extracted from the software repositories and used in the future for preventing bugs, decision-making and optimizing development processes.

In additional, even ignoring absence of the MSR in the existing works, they also have a narrower coverage of the quality analysis process. For example, in [22] and [23] authors employ only several software quality metrics. In that case general value of quality will not be a full-fledged. It will lead to distortion of results. Our work eliminates this shortcoming by use of 7 key performance indicators with 21 quality metrics. In [24] there is no any software quality analysis at all and also calculation of correlation between structural source code changes and software quality is absente. [25] and [26] describe impact of the structure refactoring on the internal program quality, which is quite similar to what we have researched in this work, but the difference is that they have focused only on the refactoring process, while this thesis has covered all the cases, when source code could be changed, including refactoring. Given that, contribution of this work will allow future researchers to conduct more detailed and complex research and analysis.
6 Conclusions and Future Work

During this research several significant results were obtained. General procedure of the software quality analysis through MSR during software evolution was defined, described and applied in practice. A few important answers were obtained in the analysis phase. It was found that correlation between software quality and software repository features exist, but it is to small to make a real influence. Also, it was proved that there is no most influential repository feature, which has bigger relationship with software quality compare to another ones.

One of the main values of this work is also an ability for another future researchers to reuse developed tool to conduct their own more detailed, complex research and analysis. Software is designed in such way that it will be very easy to start using it for people who don’t have any programming knowledge at all, even given that this tool doesn’t have graphical user interface and could be accessible only via command-line interface.

Developed tool could be easily extended. Software architecture was designed in such way that all of the components are very flexible and could be replaced, e.g. it could be another quality measurement tool instead of VizzAnalyzer™ or connector to another VCS service, such as Bitbucker or GitLab. It gives us an opportunity to significantly expand the scope of a research. Different quality measurement tools allow supporting more programming languages and a wide range of VCS connectors expands number of available projects for analysis. At the same time flexibility of components can give us more reliable results, e.g. when using two different quality measurement libraries to analyze the same project.

To make the experience with the tool even better, a few improvements could be performed. Currently measurement of the software quality takes quite a long time. In case, when it is necessary to analyze whole projects with thousand of commits, measurement will be run for each of them and it will take a lot of time. Possible solution for this issue is to parallelize all the calculations and run the analysis in a several threads, e.g. calculate software quality for several commits at the same time and extract software repository features in parallel with quality measurement. That could increase performance and reduce overall runtime.

Another improvement, which could be done during the short amount of time, is support of the C# and PHP programming languages. VizzAnalyzer™ already provide an ability to analyze projects, written on those languages. Such improvement will give us an opportunity to start conducting not only multi-version, but also multi-language analyses.

Also, as a part of the future work, a few more research based on big projects with large number of commits (more than 1000) could be conducted. Then any threats to validity can be dispersed, we can deeply analyze the results, and find out new insides, which currently are invisible.

All of the above brings tons of the new possibilities to the future researchers and allows them to conduct new more complex, deeper analysis and discover new findings.
References


Appendix A: Source code sample of how to use the quality analysis tool

```java
public static void main(String[] args) {
    String repositoryName = "commons-beanutils";
    String repositoryUrl = "https://github.com/apache/commons-beanutils.git";
    String repositoryOwner = "apache";
    String branchName = "trunk";
    int numberOfCommits = 10;
    int offsetOfCommits = 0;

    CommitAnalysisInput input = new CommitAnalysisInput(
        VCSType.GIT_HUB,
        repositoryName,
        repositoryUrl,
        repositoryOwner,
        branchName,
        numberOfCommits,
        offsetOfCommits);

    QualityAnalyzer analyzer = new QualityAnalyzerImpl();
    List<CommitAnalysisOutput> output = analyzer.analyzeCommits(input);

    ReportService reportService = new XLSXCommitReportService();
    reportService.create(input, output);
}
```