Supporting Software Maintenance Process by Detecting Software Co-Changing using Mining Software Repositories

By:

Ali Aljilani Khamis Ben Abdabdullah

Supervisor:

Dr. Abdelsalam Maatuk

This Thesis was submitted in Partial Fulfillment of the Requirements for Master's Degree of Science in Software Engineering

University of Benghazi

Faculty of Information Technology

Department of Software Engineering

Mars 2022

University of Benghazi
Faculty of Information Technology

Department of Software Engineering

Supporting Software Maintenance Process by Detecting Software Co-Changing Using Mining Software Repositories

By

Ali Aljilani Khamis Ben Abdabdullah

This Thesis was Successfully Defended and Approved on

Supervisor

Signature**: ……..…….**

Dr ……...……. …..……. (Internal **examiner)**

Signature: …….…….

Dr……...…….…… (External **examiner)**

Signature: ……...…….

Dean of Faculty **Director of Graduate studies and training**

Acknowledgements

It was a long journey, and no matter what the destination is, I enjoyed it so much. I walked through the days and nights, I faced Hills and valleys. While I am finally here, I would like to raise my hat and appreciate the effort of those who pulled me out of the valleys and pushed me up to the top of hills. Those who lit my path in the dark of night and shed my way in the heat of the day. The staff members of the University of Benghazi who led me throughout my Master's study, my supervisors Professor **Abdelsalam M. Maatuk** and Dr **Osama Bin Omran**, and of course, my family who suffered during all of those years and supported me with everything. Finally, I want to thank my mentor **Haj Waleed Aldubia** , the one who guided me with his wisdom and provided me with everything I needed since I was in primary school. I owe him all my achievements; without him, I would not be here submitting my master thesis.

Table of Content

List of Figures

List of Tables

Abbreviations

ICSE	**International Conference of Software Engineering**
MSR	**Mining Software Repositories**
MAMP	**Machintosh, Apache, MySQL, PHP**
TID	**Transaction Identifier**
GUI	**Graphical User Interface**
IID	**Item Identifier**
CCP	**Changer Propagation Path**
CVS	**Concurrent Versioning System**
ROSE	**Reengineering of Software Evolution**
IDE	**Integrated Development Environment**
DR	**Development Replay**
CDG	**Change Dependency Graph**
JSON	**JavaScript Object Notation**
VCS	**Versioning Control System**
DVCS	**Distributed Versioning Control System**
ECLAT	**Equivalence Class Clustering and Bottom-up Lattence Traversal**
SQL	**Structured Query Language**
CP	**Change Prospect**
BLOB	**Binary Large Object**
WALead	**Wide Assisting and Leading**
DBMS	**Database Management System**
PHP	**Hypertext Preprocessor**
HTML	**Hypertext Markup Language**

Supporting Software Maintenance Process by Detecting Software Co-Changing using Mining Software Repositories

By: Ali Aljilani Khamis Ben Abdabdullah

Supervisor: Dr. Abdelsalam Maatuk

Abstract

Software maintenance is considered the costliest process in the software system development life cycle. The changes made in this process on a specific software entity may trigger co-changes in other software entities. Detecting these co-changes manually increases the time and the cost of the maintenance process, while ignoring those co-changes may lead to software defects or poor software performance. Mining the historical data stored on software repositories may help in detecting software entities' co-changes. In this research, we propose the Change Propagation Path (CPP) approach. The CPP approach is a co-change detection approach that depends on mining software repositories. The CPP approach consists of three main phases. In the first phase, the commit data stored in the Git repository are gathered. In the second phase, the data gathered are prepared to be analyzed. The features are extracted, the misleading commits are removed, and the file names are coded. Then, the files that are tagged as deleted are ignored. Finally, the data are reduced. The output of this phase is a transactional database containing a set of coded file names lists. The final phase includes four main steps. The first step is generating all the possible patterns from the file names lists. The second step is creating rules from the patterns that describe the relationship between files. In the third step, the rules with the same antecedent are aggregated. In the fourth step, the rules are chained according to the software editing scenario. The output of the approach was tested manually to evaluate the output. A tool (Wide Assisting and Leading) was built upon the CPP concept and tested to prove the feasibility of the approach. Testing the CPP approach proved that mining software repositories may reduce the time of the maintenance process by 50%.

Chapter 1

Introduction

A software system can be defined as several separate programs along with its related configuration files. It may also include the documentation that describes its design, underlying databases, and other related files. In other words, the software system is a set of entities that depend on each other and evolve together. The source code elements, databases, and files are considered software entities [9] [10].

Software entities are being updated continually due to a new feature requested or to fix a reported bug. The changes made during the updating process may trigger other changes. Therefore, one change may lead to a complementary change (co-change) or a change propagation through the whole software system [10] [9].

Co-change is the change required by another change to complete the maintenance process. For example, adding a new data field to an existing system requires multiple changes in the software system. First, the data field should be added to the database. Then, the code responsible for adding, reading, and editing that data field must be changed too. Finally, the front end of the system must be changed by adding data input to receive the new value and label that describes the purpose of that input. All these mentioned changes are the result of one single change. Hence, the co-change is the effect of the coupling among software entities [21].

Coupled pairs in software engineering are the software entities that have a direct or a hidden relationship. For example, a global variable that is used within an object or an object that uses another object. This type of coupling is referred to as explicit coupling or direct coupling. The other type of coupling occurs when software entities are frequently changed together and there is no direct relationship among them. This hidden relationship can be referred to as logical coupling or evolutionary coupling [12][14]. Hence, evolutionary coupled entities are the entities that frequently change together [13].

Predicting co-changes makes developers aware of entities that need to be changed along with the entities that they are currently working on. This is to avoid defects and maintain system integrity [5]. Many difficulties to detect bugs are induced by developers who did not notice the hidden relationships among the software entities, which lead to a change propagation failure. Other defects come as a result of the ignorance surgeries, which are

modifying the source code by developers who do not have enough knowledge about its purpose and its structure. Predicting co-changes can also reduce the time consumed during the maintenance process by guiding the developers through the related changes [9][11].

In some cases, co-changes are easily recognized by the dependency browsers provided within the development environments. These browsers analyze the structure of the software system to detect the related entities (directly coupled) and consequently, detect the affected ones during the updating process [11] [14]. For example, when Class X uses an instance of Class Y as shown in Figure 1.1. However, in other cases, especially when there is no direct relationship between entities (logically coupled), co-changes are hard to be detected using structural analysis. For example, when entity A writes data in a file, and entity B is desired to read this data, so that any change in entity A that may affect the data written must be reflected in entity B. The entities A and B are not structurally related but there is a hidden relationship between them. Figure 1.2 visualizes this type of relationship.

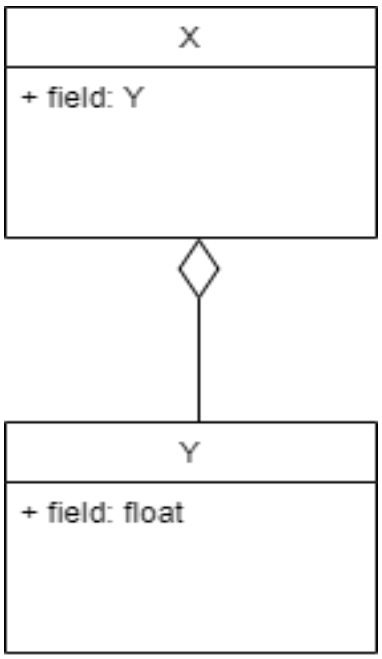

Figure 1.1 direct relationship between two classes

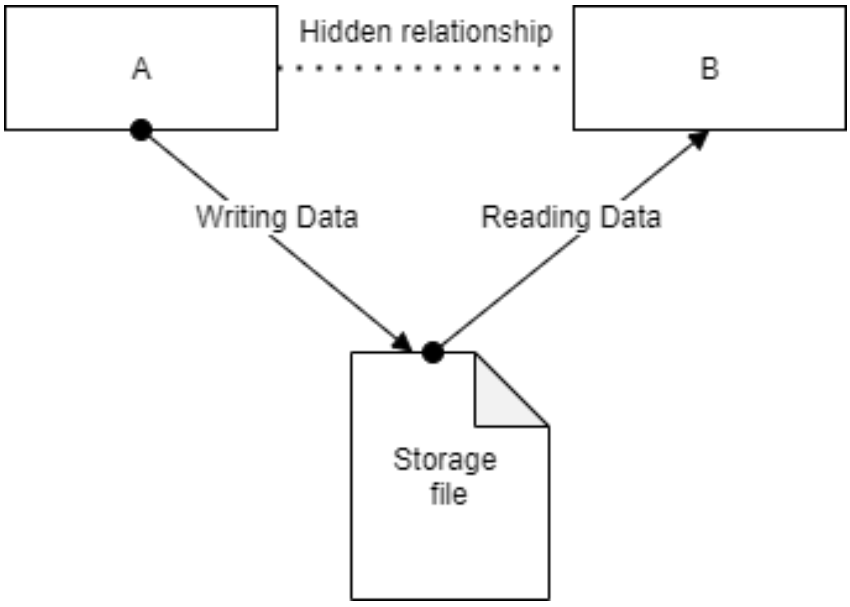

Figure 1.2 Hidden relationship among software entities

A tremendous amount of data is produced during the software system development process, describing the changes made in this process. It represents which part of the system has changed, who made the change when this change was made and other related data describing each detail of the development process. These data provide a beneficial information source, which is helpful for many software engineering aspects, especially detecting co-changes. All of the mentioned data in addition to the software system itself and all its previous versions are stored in software repositories, which are the infrastructures that support software development process activities [1].

1.1 Problem Statement

Most difficulties to detect software bugs are induced by developers who sometimes failed to detect related entities and propagate co-changes correctly. The explicitly coupled entities are usually detected manually by reading the source code to detect the related entities or using the Dependency Browsers, which are tools that are usually installed within the Integrated Development Environments (IDEs). These tools analyze the structure of the software system to detect the explicitly coupled software entities. Hidden relationships among entities are undetectable by manual revising or Dependency Browsers. These relationships can be detected by analyzing the development history of the software system to recognize frequently changed together entities [13].

Many approaches have been proposed to solve this problem by analyzing the data available in software repositories to guide the developers through the change propagation process. However, the developers' community has not yet adopted these approaches widely because of the low accuracy and the high misleading recommendations provided by these approaches [1] [11] [14]. In this research, we introduce a Change Propagation Path (CPP) approach, which is a Mining Software Repositories (MSR) approach that adopts the frequent patterns analysis techniques. The CPP approach provides a suggestion about the complementary changes depending on the data extracted from the software repository.

1.2 Aim of the Research

Since the last decade, until the recent date, researchers tried to find an optimal solution to predict co-changing software elements. All of these attempts are not adopted yet by the developer's community because of the low accuracy and the high rate of misleading recommendations. Hence, this research aims to propose an approach that increases the accuracy of detecting co-changed software entities by using the historical data stored within software repositories. The approach is designed to assist the software maintenance process by reducing the time and cost it takes. This research also aims to prove the feasibility of the proposed approach by conducting an experimental study, in which we will apply the approach to a maintenance task and measure its effect on the maintenance process. The following objectives will guide this research to fulfil its aims

1. To introduce a sufficient background about software co-changes and MSR. This includes the analysis of the previous work and identifies the problem related to the co-change prediction area,
2. To extract and prepare the historical software development data,
3. To apply an adapted frequent pattern analysis algorithm to produce a single entity antecedent rule, and process these rules to form the change propagation path,
4. To evaluate the proposed approach and discuss the obtained results.

1.3 Research Questions

This research tries to answer the following questions:

RQ1: To what extent the cost and time are affected positively by applying CPP during the maintenance process?

RQ2: What is the optimal software repository data extracting tool?

RQ3: What are the features of the data extracted from the software repositories that will produce knowledge?

RQ4: What factors are vital to selecting a data mining algorithm for producing required knowledge for the CPP approach?

1.4 The Proposed Method

In this research, we employed the quantitive methods through a deductive approach, on data extracted from the Git software repositories. The proposed approach consists of three main phases. The data extraction phase is where the data are gathered. The data preparing phase where the data is cleaned, transformed, and reduced. The analytical phase is where the data mining techniques are applied to produce knowledge from the preprocessed data. The output of these three phases comes as recommendations to guide the developers through the maintenance process.

1.5 Dissertation Structure

This dissertation starts with an introduction and a brief description of the context of the problem under the study and outlines our aims, objectives, and research questions. Chapter 2 provides a sufficient background about the topics, tools, and applications mentioned in this research. Chapter 3 previews the previous works related to our topic, and the contribution of this research. Chapter 4 introduces the research methodology that will guide this research to achieve its aim. Chapter 5 contains the implementation of the proposed solution and describes the experiment that we are conducted to prove the validity of the proposed approach. Chapter 6 represents testing the feasibility of the approach. Chapter 7 presents the conclusion of the research along with the recommendations and future work.

Chapter 2

Background

Software maintenance is the costliest process among software system life cycle processes. It costs about half of the total software system development budget [10]. The edits made during this process may require a complementary change (co-changes), which are the changes made to other software entities according to a previous change. Ignoring those co-changes may cause defects or software poor performance. Co-change is the result of software entities coupling that may be explicit and easily detected, or implicit and difficult to be noticed manually. Co-changes can be detected by revising the historical data of the software system development. A vast amount of data is produced during the software development process. This data describes each detail in the software system history, and is stored in the software repository along with the software itself and its previous versions. The data stored in software repositories are a valuable source of knowledge that serves many aspects of software engineering. Software repositories store a huge amount of data in unstructured form, gaining knowledge from big unstructured data requires applying data mining techniques to produce the desired knowledge [1].

This chapter gives a sufficient background about the topics of this research. In the beginning, this chapter introduces the software maintenance process, then it discusses the co-changes, which is the side effect of the changes made during the software maintenance process, and why it is important to predict those co-changes. After that, this chapter talks about software repositories and their categories and how knowledge is extracted from software repositories using data mining. Finally, the chapter describes the tools and programs used to accomplish this research aims.

2.1 Software Maintenance

Software maintenance is the process of updating the software system after being delivered. This update may be required as a result of a software defect occurring after system delivery, adapting the system to a new environment, or a new feature requested by the customer. After the software system is delivered, development team members usually break up, opening the way for new members who have no experience of the software system to do the maintenance tasks [10]. The new team members might spend a long time to be familiar with the system, and be able to propagate changes correctly. The

maintenance contract is usually separated from the development contract, leading to assigning the development process to one company, and the maintenance process to another company. As a result of this poor practice, more time might be consumed to understand the system by the new company team members. The maintenance process is considered a less-skilled process than the development process, so it is usually assigned to junior developers who do not have enough experience, which induces more defects in the system. Due to incomplete changes, more time is consumed and higher costs to maintain. Consequentially, maintaining a software system costs two times as much higher than the developing process [10].

2.2 Co-change

Changing a software system entity may lead to changing another entity, or a change propagation through the whole system. This change is also referred to as co-change. Failing to propagate changes correctly is the main reason for software defects and poor performance so that predicting these changes can reduce the time and cost spent on software maintenance. In some cases, when software entities are structurally related, co-changes are easily detected. In other cases, when there is no direct relationship among software entities, co-changes require more effort to be detected by revising the software development history and relating the frequently changing together software entities [1][9][22].

2.3 Software Repositories

According to Hassan et al [12], software repositories have three main categories depending on the data that is stored in them: historical repositories, code repositories, and deployment logs. These repositories provide the infrastructure that supports the software development process, forms a collaborative environment where development teams can host their projects, keeps the track of those projects, and works remotely in a collaborative way. The following subsections describes each type of software repositories:

2.3.1 Historical Repositories

The purpose of the historical repositories is to be used as an archive for software systems and the data, which illustrate the software development process. This category of repositories has different types according to the data stored in them.

2.3.1.1 Source Control Repositories

Source control repositories or version control systems track the project's development history by recording each change made in the software along with a meta-data that describes each change. For example, which part of the software was changed, who made the change, and when the change was made. Source control repositories provide a short message sent by the developer to describe the purpose of the change. They also provide the ability of parallel development by branching features. Some of these repositories work in centralized style, where the repository is hosted on a single server, i.e., Concurrent Versioning System (CVS) and Subversion, while other repositories work in distributed style by mirroring the whole repository among clients like Git and Mercurial [17][12].

2.3.1.2 Bug Repositories

Bug tracking is the process of tracking and monitoring the bugs and issues that occur during the development process. Issue tracking systems (also known as bug repositories) are responsible for storing developers' bug reports and the features requested by users. It categorizes, describes, and tracks the problem to enable the developers to suggest enhancements of the reported bugs. The Jira and Bugzilla are examples of this type [18] [17].

2.3.1.3 Communications Archives

Communications archives record all the discussions and communications among the development team members about the development process. Communications archives contain Emails, instant messages, and other types of communications [12].

2.3.2 Code Repositories

Code repositories are storage spaces where developers can store and share their project releases. These repositories are usually integrated with other software development support systems to create a collaborative development environment such as sourceforge.com, GitHub, and Google code [1].

2.3.2.1 Git Repository

Git is a distributed version control system that allows developers to work remotely. Also, keeps track of all the changes made during the development process, along with the related data, such as who made these changes, when these changes were made and a message that describes why these changes were made. Figure 2.1 shows a sample of Laravel framework data stored on Git repository.

```
Date : 2011-06-08 23:45:08-05:00
 Message initial commit of laravel!
 Commit Id a188d62105532fcf2a2839309fb71b862d904612
 Files
File Name : .gitignore
complexity : None
Change Type : ModificationType.ADD
Removed : 0
Added : 0
diffrences : {'added': [], 'deleted': []}
Number of lines of code : None
**************************************************
File Name : application.php
complexity : 29
Change Type : ModificationType.ADD
Removed : 0
Added : 98
diffrences : {'added': [(1, '<?php'), (2, ''), (3, 'return array('), (4, ''), (5, '\t/*')
Number of lines of code : 32
```

Figure 2.1: A sample of Laravel framework development data stored in the Git repository

In 2002, the Linux kernel project started to use a DVCS called BitKeeper as a free-of-charge product, to keep the track of changes they made [12]. After three years, the free-of-charge deal was cancelled due to some issues between the Linux community and the company producing BitKeeper DVCS. This forced Linus Torvalds, the inventor of the Linux operating system, to create a new tool named Git. This new tool is focused on

fixing the issues they faced with BitKeeper. The new tool aimed to work faster with a simpler design and support for parallel development. Also, fully distributed and able to handle large projects [12].

Git works in a simple style as described in Figure 2.2. After the repository is initiated, the first step is making changes to the projects file, the changes are snapshotted on the staging area, and then the changes are committed to the Git repository. The commits made on the Git repository can be pushed to the remote server later. Git Commit contains all the data describing the changes made, besides the affected files. Each commit has a unique identifier that allows the developer to retrieve the data within the commit, and also to revert the changes made to a certain point of time [12].

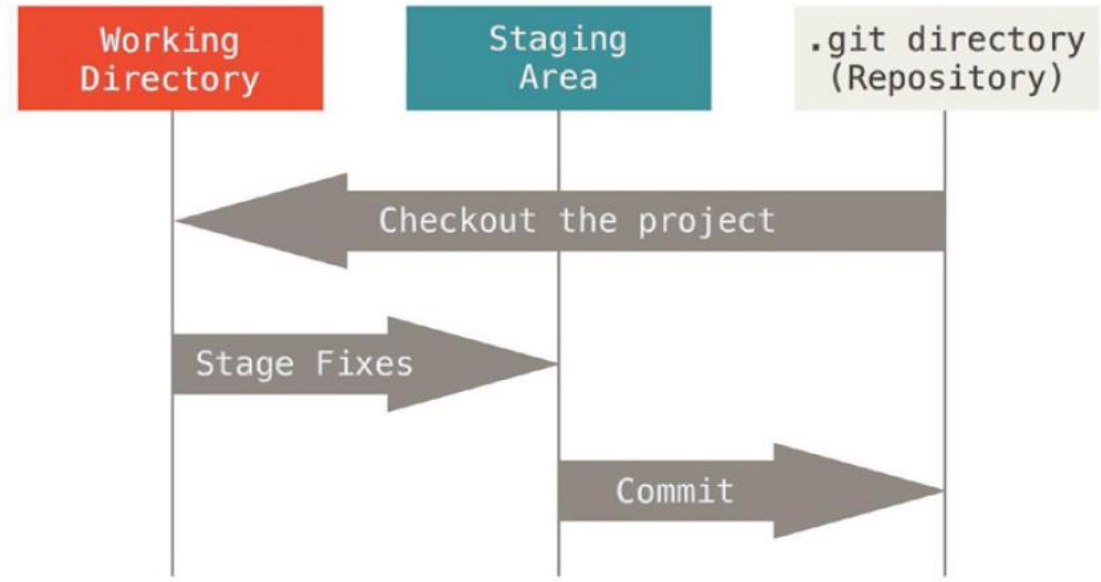

Figure 2.2: The way that Git stores and retrieves changes data [12]

2.3.3 Deployment Logs

Deployment logs repository contains data about the software execution and usages such as error messages and software performance. The data stored in logs is used to diagnose failures and poor performance and helps to propose solutions [19] [12].

2.4 Mining Software Repositories

Since the last decade, software repositories have taken a vital role in the software development process. The data stored in these repositories started to attract the attention

of the scientific community. Analyzing this data can produce valuable information, which can be acted upon. This process is known as Mining Software Repositories (MSR) [37]. Due to the importance of the MSR field, the first workshop about MSR was held in 2004 by the International Conference of Software Engineering (ICSE), and after four years of working in this field, the first MSR conference was held [12]. MSR is helpful in many aspects of software engineering. It can help the development team to understand software systems. Predicting and identifying bugs can be more effective by analyzing the previous versions of the software systems stored within the repositories. Revising the historical data of a software system conveys vital information about the pairs of entities that change together and how and why the change was made. This can guide developers while propagating changes in future versions of the systems [12].

2.5 Frequent Pattern Analysis

Frequent pattern analysis is the analytical process of detecting the frequently occurring data sets. This process is applied to a transactional database, which is a set of transactions, each transaction has a unique identifier and contains a set of items [28]. A frequent pattern is the item set that satisfies the minimum support threshold assigned by the data analyst. Those patterns are considered interesting patterns. The support means that the percentage of transactions that contains a particular data set in a given transaction set [29]. Let T be a set of transactions and X is a set of items, the support (SUPP) of X is calculated as shown in the following equation.

$$\text{SUPP(X)} = \frac{|\{X\} \in T|}{|T|}$$

Interesting patterns are the raw material to create association rules, which are the rules that describe the relationships among items. This type of rule is usually used to predict consumer behaviour, in the manner of consumers who bought this item also bought that other item. This can be useful in making offers, promotions, and ordering items on market shelves. Each rule has a left side (antecedent) and a right side (consequent). The rule is considered interesting if it satisfies a minimum confidence measure [28]. Let X and Y be disjoint item sets, where T is the set of transactions so that the confidence (CONF) of the rule $X \rightarrow Y$ is calculated by the following equation

$$\text{CONF}(X \rightarrow Y) = \frac{SUPP(X \cup Y)}{SUPP(X)}$$

In the same way, the frequent pattern analysis can be used to predict the co-changing software entities, developers who changed this entity also changed that other entity, which reveals the hidden relationships among software entities.

In 1966, Petr Hájek and chytil [30] introduced the General Unary Hypothesis Automation (GUHA) method. This method is aimed to analyze the properties of a set of objects to convey if a combination of properties is the cause of another combination of properties. For example, a combination of symptoms is an indicator of diseases. GUHA was the first attempt to analyze the frequent patterns [30].

In 1994, Agrawal and Skrikant [31] have proposed the Apriori algorithm that follows the candidate itemset generation approach, by applying a breadth-first search to generate all the possible itemsets within a transaction, where k-frequent itemsets are used to find the k+1 item set. Generating all the candidate itemsets makes the Apriori algorithm unable to handle large transactions or big databases.

Later in 2000, Han et al [32], have proposed another method to generate frequent patterns, called the FP-Growth algorithm, to generate frequent itemsets without candidate generation. By using a prefix-tree structure called FP-tree (Frequent-Pattern tree), FP-growth solved several issues in the Apriori algorithm, such as the repeated scanning of database FP-growth only scans the database twice the time consumed generating the candidate itemsets also reduced in FP-growth. However, the FP-growth algorithm suffers from memory consumption when applied to large data sets.

Another algorithm was introduced in 1997, which is called Equivalence Class Clustering and bottom-up Lattice (ECLAT) was proposed by MJ Zaki, et al [33]. ECLAT is a scalable algorithm that uses the depth-first search approach, which consumes less memory than the Apriori algorithm, also the ECLAT does not involve multiple database scans, which makes it work faster than the other approaches[33]. Candidate Generation and FP-growth approaches use item-id data sets, where each transaction is a set of items with a transaction id as shown in Figure 2.3. The ECLAT algorithm uses a different data format, where each item is associated with a set of (TIDs) transaction ids. Figure 2.4 describes the TID data format, which is helpful in the manner of scalability, but sometimes TID gets quite long and expensive to compute. This problem was solved by using the Diffset technique [29].

TID	Items
T1	I1, I2, I3, I4 ……In

Figure 2.3: items IDs data format

Item	TIDs
I1	T1, T2, T3, T4……Tn

Figure 2.4: Transactions IDs data format

2.6 Tools and Applications used in the Proposed Solution

In this section, we describe a set of tools and applications that were used to accomplish our study. First, we introduce Komodo Edit, which is a text editor that was used to write the front end of the suggestion tool. After that, we give a brief description of MAMP, the local web services environment that was used to host the suggestion tool on the local server. Then, we describe PyCharm, which is an integrated development environment that was used to write the back end of the suggestion tool. After that, we introduce Ali Research Tool (ART), a web-based tool, which is designed by the researcher to summarize the information gathered from books and research papers. We also mention the Git repository, the source of the data that was used to prove the feasibility of our approach. Finally, we describe MySQL relational database management system, and its graphical user interfaces PhpMyAdmin, which was used to store the output of the processed data.

2.6.1 Komodo Edit

Komodo edit as shown in Figure 2.5, is a text editor developed by Active State [53]. Komodo Edit provides the ability of managing source code, by colouring different parts of the source code and giving the code a proper layout. It also auto-corrects syntax errors and auto-completes the code therefore, the code writing using Komodo is a faster and more accurate process. We used Komodo Edit to write the Php source code of the WAlead tool [53].

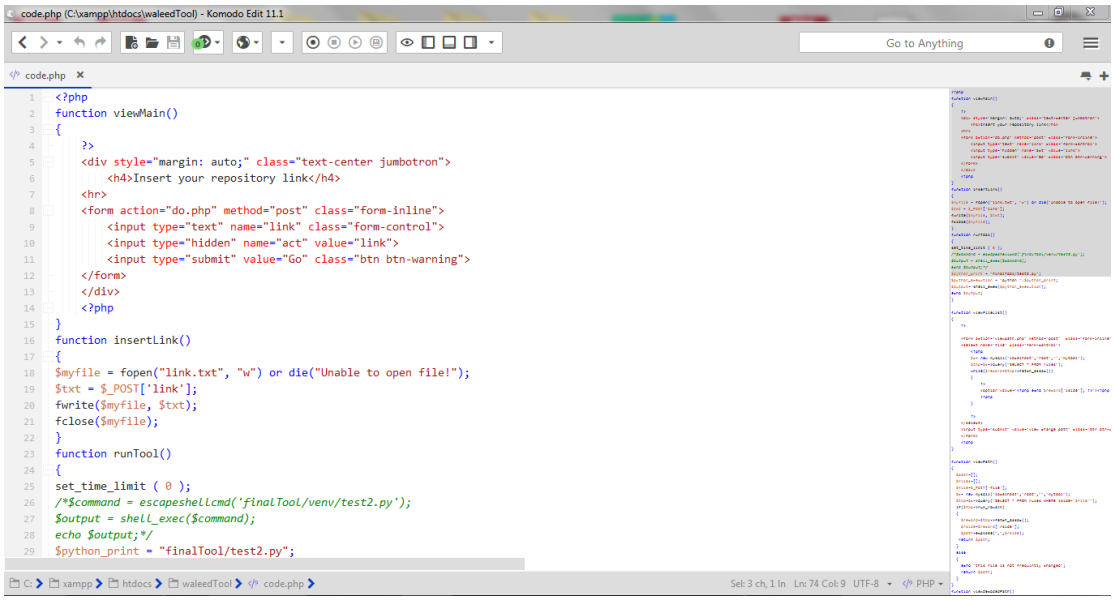

Figure 2.5 Komodo edit to write PHP, code

2.6.2 MAMP

The MAMP is a local web services environment as described in Figure 2.6. It contains all the necessary tools and apps to test web applications on local machines. MAMP contains apache and Nginx webservers, MySQL database management system, and supports most back-end web development languages [54]. We used MAMP to test the WAlead tool on a local machine and we managed to use it as a server on a local network during the experiment.

Figure 2.6: MAMP local web services environment

2.6.3 PyCharm

PyCharm is an (Integrated Development Environment) IDE Designed specifically for Python language as shown in Figure 2.7. It provides all the required tools to write and run python code [55]. We used PyCharm to write python code in data extraction, preprocessing, and analyzing phases. Also, we used it to write the python part of the WAlead tool.

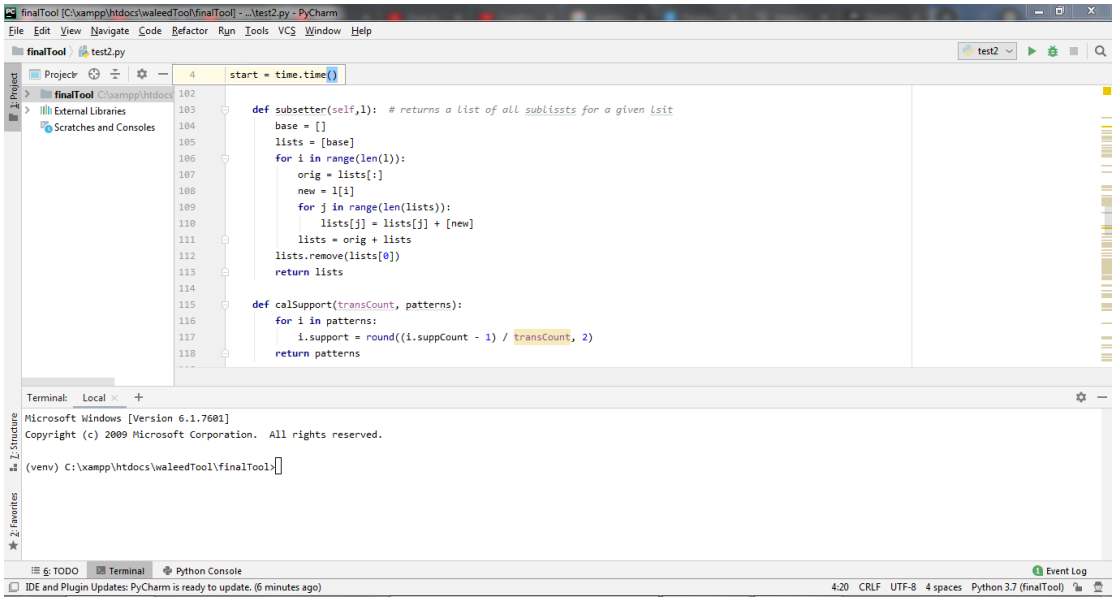

Figure 2.7: The PyCharm IDE

2.6.4 Git Repository

Git repository is one of the most popular software repositories, as it is considered a software changes tracker. Git repository provides the ability to store the software itself along with all the previous versions and the metadata that describes the development process [56]. We used Git to keep track of the changes of the WAlead tool. Also, Git is the data source for our research.

2.6.5 MySQL and PhpMyAdmin

MySQL is a database management system that adopts the concept of the relational database. MySQL databases is a reliable storage system that is suitable for small and medium projects, and it is also compatible with PHP and python the languages used in

this research's practical part [57]. MySQL is usually accessed and manipulated through the command line, which is a complex and time-consuming process. Hence, we used PhpMyAdmin, the graphical user interface of MySQL to create and manage the databases. Figure 2.8 describes the PhpMyAdmin user interface.

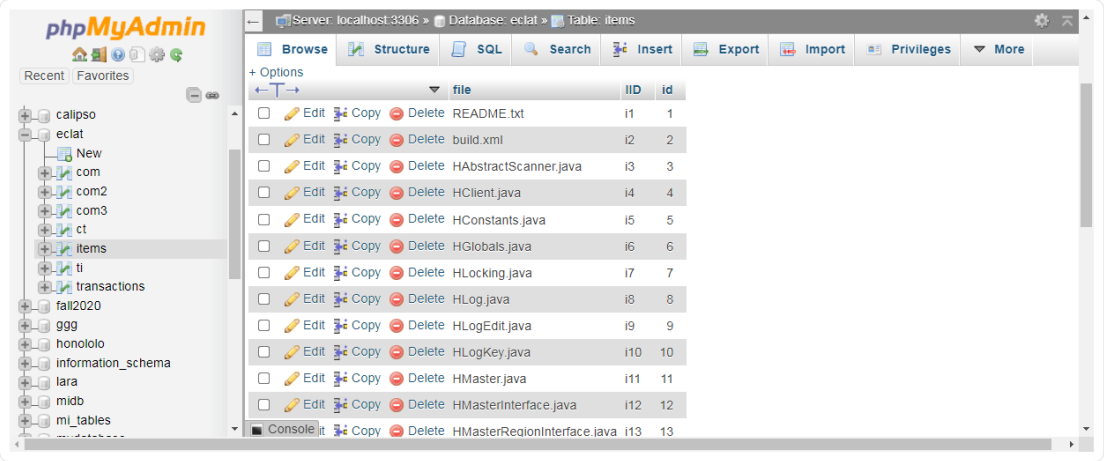

Figure 2.8: PhpMyAdmin the GUI of MySQL

Chapter 3

The Literature Review

As a time-consuming process, searching for hiddenly related software entities during the maintenance process has gained the attention of many researchers since the last decade. The aim was to find an optimal approach for automating this process and providing accurate suggestions for developers to assist the maintenance process.

Thomas Zimmerman [13] applied the Apriori algorithm on historical data extracted from the Concurrent Versioning System (CVS). The rules produced by the algorithm are used to build a Reengineering of Software Evolution (ROSE) recommendation tool to guide developers while propagating changes. The ROSE was designed to be used as a plugin for Eclipse IDE, which can only be applied to a specific type of software project. The Apriori algorithm that was used to build ROSE has some drawbacks, where it consumes a long time scanning the database (N times) and spends time while creating candidate itemsets.

Thomas Zimmerman et al [21] has investigated the co-change among lines of code. Using the annotation graph to visualize how lines of code co-change over time. An annotation graph is a multipartite graph, where every part represents a version of a file. The nodes in the graph represent a line of code, and the line connecting two nodes means that a line of code is produced by editing another line of code. The approach concluded that searching for a co-changing line of code is a quite expensive and infeasible method to be applied in supporting the development process.

An approach for extracting data of over 40 years of software development and applying the Development Replay (DR) approach to these data has been proposed in [9]. The empirical results convey that the historical data of software development are extremely useful in predicting complementary changes. This can assist developers in propagating changes in the future.

Ramadani [14] proposed a recommendation system for co-changed by applying the FP-Growth algorithm on data extracted from Git versioning repositories. However, this

approach was only applicable to detect co-changes on coarse granular software entities(files).

Rolfsnes et al[4] have used a frequent pattern analysis on the data of 15 open-source software systems and introduced the concept of hyper-rules, which is the result of aggregating multi-applicable rules. The results showed that the hyper-rules can improve the accuracy of the suggestions by 13% to 90% compared with previous works.

Rolfsnes et al [27] introduced what so-called TARMAQ algorithm for mining evolutionary coupling. The algorithm focuses on the drawbacks of using off-shelf mining algorithms and worked file-level granularity. The TARMAQ algorithm has achieved a higher accuracy rather than the ROSE tool [13].

Islam [5] introduced the concept of the transitive evolutionary coupling, which is a relationship among software entities that never changed together in the past and are likely to change in the future. The traditional association rules cannot detect that kind of relationship. Therefore, a set of transitive association rules have been proposed. Compared to the TARMAQ tool that depends on the regular association rules, the transitive association rules achieved 13.96% recall and a 5.56% precision higher.

Ajienka et al [3] have investigated the hidden relationships among software classes according to the semantic coupling of its identifiers. However, this approach was applicable only on OO-designed systems and on the class-level entities. The solution concludes that there is no correlation between semantic coupling and change coupling although 70% of semantic dependencies are linked to change coupling but not vice versa.

Wiese, et al [11] have proposed a prediction model for each pair of software entities, based on relevant association rules. Those rules are produced from the contextual information extracted from issues tracking systems, developers' communications, and commits metadata. Later, Wiese, et al [20] have compared suggestions based on contextual information with suggestions based on association analysis and concluded that the contextual information provided fewer false recommendations.

Tosun and Romero [7] extended the work of Wiese et al [11] by building a prediction model to predict the co-changing files using the contextual records on software repositories. This approach achieved a 20% to 45% less accuracy than the previous work. Similar to [14] this approach applies to coarse granular entities.

Vidács, L., and Pinzger [8] investigated the co-evolution patterns between production code entities and their test code entities.

Stana and Șora [16] analyzed the relationships among logical dependencies and the structural dependencies on data extracted from 27 open-source software projects written in Java and C#. The work concluded that including structural dependencies along with logical dependencies improves the applications based on dependency models and co-change detection is one of them.

Wang et al [6] conducted an empirical study on bug fixes including multi-entities to discover the frequently fixed together entities and based on syntactic dependencies among changed entities. The approach suggested creating a Change Dependency Graph (CDG), which can be used to guide developers through the entities that are meant to be fixed.

Jiang, et al [2] proposed the CMSuggester approach, which is aimed to predict co-changing software entity pairs during maintenance tasks that require multiple changes. The most majority of the proposed approaches depends on the historical data of the software development, where the frequently changing together entities in the past is likely will change in the future. The CMSuggester approach provides recommendations based on analyzing the structure of the software code, where the methods that access the same data field are clustered, and the produced recommendations are based on method clusters. This approach has achieved 70% of suggestions accuracy. However, this approach can only suggest co-changing methods.

Beyre and Noak [26]introduced a method that clusteres software artefacts into subsystems using a co-change graph, which is a model that represents software artefacts as vertices and the co-changes among these entities as edges between vertices.

Kouroshfar [34] investigated the impact of software entities co-change on the software quality. They applied a subsystem decomposition model on four different open-source projects. The results showed that the co-changes among software entities in the same subsystem can improve the bugs prediction process.

Kagdi et al [35] have used the log-entries data in the Subversion repository to investigate the sequence of files changing the results. A set of tools have been built to discover the correct sequence of file changing, to help developers in predicting future changes, and analyze the evolution process of software systems.

Martinez and Monperrus [44] have designed a tool called Coming, which is a tool that extracts the commits data from the Git repository. The data extracted are revised to convey the change patterns of the fine-grained software entities (Classes, Methods, etc.). The result of this process is stored in JSON format. However, this tool does not provide any recommendations, since it is only applicable as a plugin in a larger mining approach.

Alali et al [46] introduced two new ranking patterns measures, i.e., pattern age and coupled files distance. Those new measures are used to reduce the false positives in co-changes recommendations. To extract the patterns from sub-version VCS, the srcMiner tool has been designed and built upon the vertical data format pattern generation algorithm ECLAT. The tool was applied on eleven different projects using file-level granularity and concluded that about 75% of co-changes are localized.

Agrawal et al [47] introduced a tool called Ruffle that was used to produce change recommendations using software revision history by calculating the changing proximity for each pair of classes. The Ruffle tool was built using Java and Python programming languages. Each software entity was stored along with the revision id that includes it, then an SQL query was applied to generate changing pairs. To evaluate the Ruffle tool performance, five different project histories have been used. However, the tool has achieved accuracy between 0.7 and 0.8.

The ability of the commit data to provide predictions for co-changes decreases by time [49] [50] [51]. To avoid this problem, Agrawal et al [48] proposed Change Prospect (CP) to measure the feasibility of a commit to increase the accuracy of predicting the co-changing pair of classes.

In conclusion, the previous work investigated the co-change occurrence or the occurrence of the factors responsible for making software entities evolve together by using different methods and approaches. Table 3.1 provides a summary of this related research work. However, it seems that none of these approaches was widely adopted by developers yet. This is because of the lack of accuracy or during the high rate of false recommendations. Some of these works focused on one level of granularity so that it cannot be generalized on the other levels of granularity. Other works have proposed expensive to apply approaches, therefore, increase the cost of the maintenance process. Hence, a usable automated co-change detecting approach requires to be accurate enough, stable and cost effective in manner of time and computation effort. While the software development is a

continues process, the co-change detection approach output must be scalable to cover the changes without reanalyzing the old data.

In this research, we introduce the CPP approach, which tries to avoid the weak points in the previous works, and employs some of the proposed techniques to produce more reliable recommendations. Therefore, this will serve the main aim of this research, which is decreasing the cost of the maintenance process

Table 3.1: a summary of the software complementary change detection approaches

Approach	Tool	Data format	Data source	Mining algorithm	Granularity level	Main Findings
Thomas Zimmerman [13]	ROSE tool	Commits	CVS	Apriori	Code elements level	ROSE tool employees' unfeasible algorithm that is expensive to apply and provides unscalable output.
Thomas Zimmerman [21]	Annotation graph	Lines of code	CVS	-	Lines of code	Detecting co-changes among lines of code is an expensive process.
Ahmad Hassan et al [9]	Development replay	Commits	CVS	-	Files	Proved that historical data is a significant source for detecting co-changes
Ramadani, J [14]	Recommendation system	Commits	Git repository	FP-Growth	Files	This approach is only applicable on file level only
Rolfsnes, T et al[4]	Hyper rules	-	16 open source projects	Association rules	Files	Aggregating applicable rules can increase the accuracy of the co-change detection process
Rolfsnes et al [27]	TARMAQ	Commits	-	TARMAQ	Files	TARMAQ is a mining algorithm that was exclusively designed to detect co-changes
Islam, M.A[5]	Transitive rules	Commits	-	Association analysis	Files	Spotted the light on the relationship among software entities that have never changed together in the history
Wiese, I.S et al [11]	Prediction model	Contextual data/Commits metadata	Communication archive, issue tracking systems	Association analysis	-	Using the contextual data along with the traditional methods may increase the accuracy of the co-change detection process.
Tosun, A. and Romero, B [7]	Prediction model	Contextual data/Commits metadata	Communication archive, issue tracking systems	Association analysis	Files	Using the contextual data only may decrease the accuracy of the predicting process by 20% to 45% less than the work in 7
Wang, Y et al [6]	CDR (Change	Contextual data	Bug tracking systems	-	Code entities	Using bug fixing data can help in detecting co-changes

	propagation graph)					
Jiang, Zet al[2]	CMSuggester	Source code	CVS	clustering	Code entities	Co-changing methods that access the same data field are likely to co-change
Kouroshfar [34]	Subsystem decomposition model	Source code	Four different open-source projects	-	Code entities	Artifacts in the same subsystem are more likely to co-change
Kagdi et al [35]	Sequence file change correcting tools	Historical data	Subversion repository	analyze the evolution process of software systems	Code entities	The correct sequence of software entities changing may affect the accuracy of the co-change detection process
Martinez and Monperrus[44]	Coming tool	Commits	Git repository	-	Software releases	Provided a plugin that can be used in larger mining software repositories projects
[47] Agrawal	Ruffle tool	Source code	Five different projects	SQL queries	classes	Investigated the change proximity for each pair of software entities

Chapter 4

The Proposed Method

This chapter describes the design of the CPP approach and its phases. The CPP is an approach for detecting co-changes among software entities. The CPP approach employs frequent patterns analysis techniques to search in the historical data of software development for frequently changing together software entities. The frequent patterns generated are used to create association rules, which predict the co-changes for a set of software entities based on a change on one single software entity. The created rules are aggregated to form a larger rule based on the same antecedent. The larger rules create the Change Propagation Path depending on the software editing scenario, which leads the developers through the related changes. Using the CPP approach may decrease the time consumed while searching for related changes manually during the maintenance process. It also may eliminate the cost of hiring highly paid senior developers.

The aim of this research is achieved by employing quantitative methods, through a deductive approach [58] [61]. The aim is reached by applying a data mining framework, on the data generated during the development process and stored in software repositories. Figure 4.1 describes the three phases of the proposed approach. Phase I is concerned with gathering data from the Git software repository. The data collected in Phase I are prepared and cleaned in Phase II. Finally, in Phase III, the required knowledge is produced using data mining techniques.

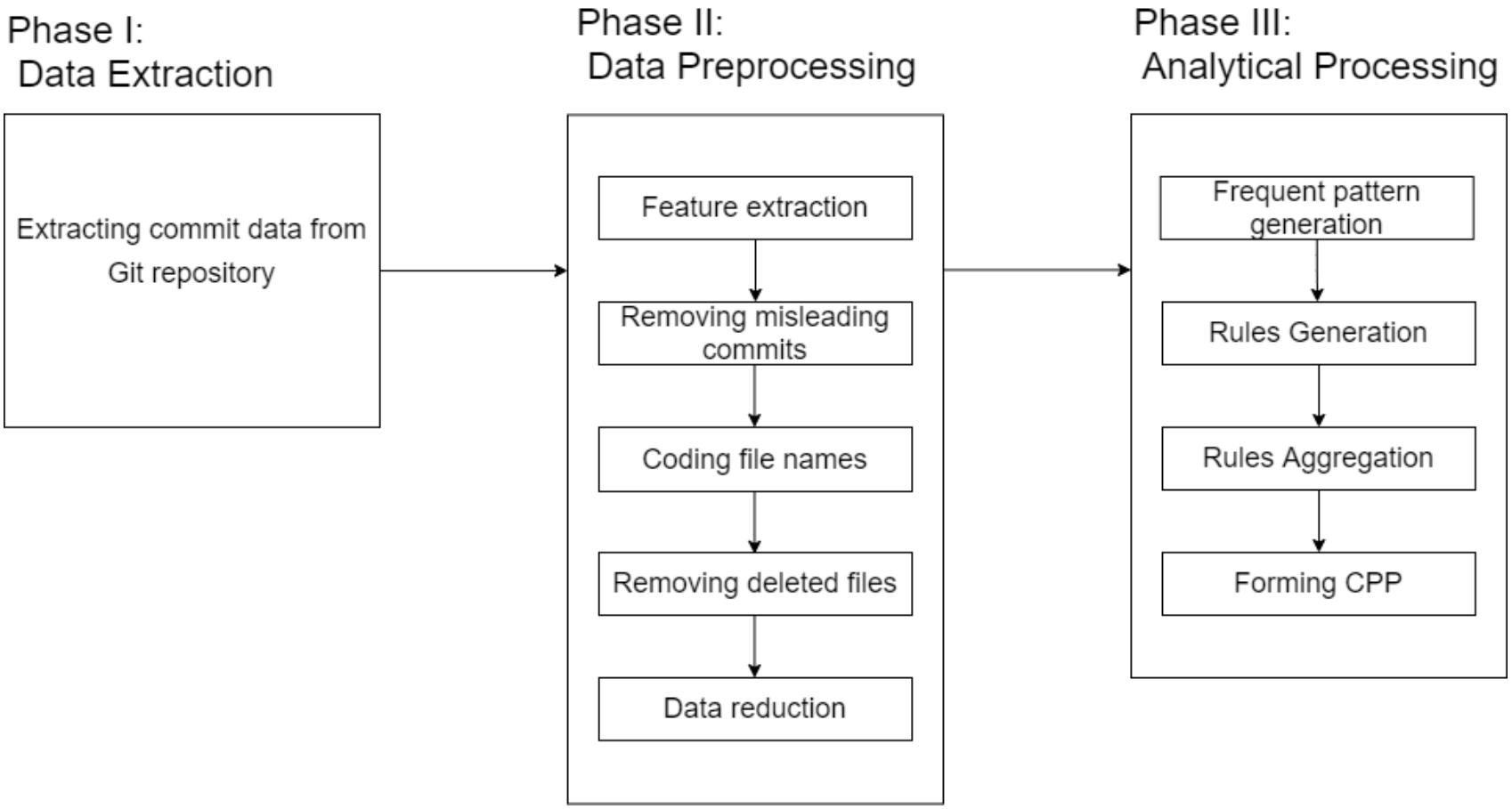

Figure 4.1 The CPP Approach Framework

The following subsections describe the three phases of the CPP approach. The process starts with extracting data from the software repository, cleaning and preparing the extracted data., and finally, analyzing the data and producing the required knowledge in the form of a change path.

4.1 Phase I: Data Extraction

In this phase, the data stored within the Git software repository are extracted. This data should be covering a long period of development time to enable the mining method to convey relationships among software entities. The commits stored within the Git repository that is stored on a local machine, or the commits pushed to GitHub are pulled using one of the available commit extraction tools. We conducted a comparative study among data extracting tools and frameworks. To decide which extraction tool is suitable for our purpose.

The Git repository stores the software system versions and records the changes made on its entities. Also, it records the metadata describing those changes in the following three different objects as shown in Figure 4.2.

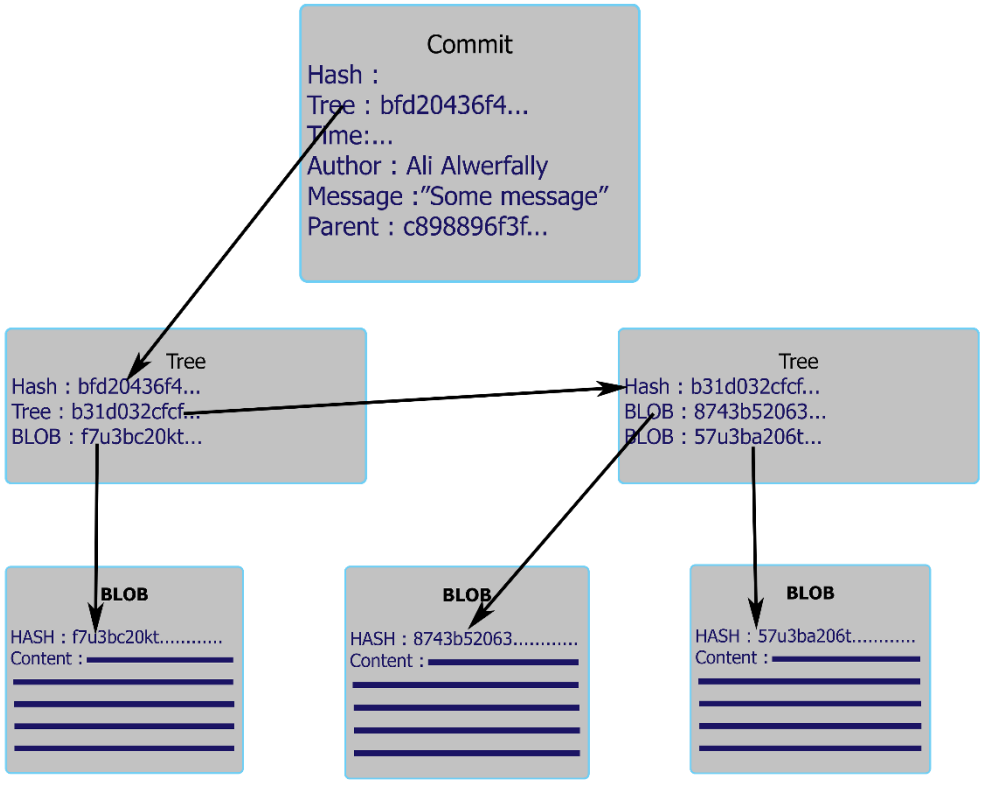

Figure 4.2: The method used by Git to capture files changes

1. **BLOB (Binary Large Object):** stores the content of each file as a string without the file metadata (filename, creation date, ... etc.). Each BLOB is identified with a SHA1-hash referring to its content.
2. **Tree Object:** represents a directory that refers to BLOBs and other trees (directories), trees are identified by SHA1-hash that are produced according to the tree content.
3. **Commit Object**: represents the status of the system at a point in time (snapshot of the system). The commit points to the main tree hash and contains the metadata about the latest changes (i.e., who made the latest changes, when the latest changes were made, … etc.). Each commit has a parent commit describing the previous statutes of the system.

When a developer makes a change to a file, the hash of the related BLOB will be changed according to the new content of the file. This change will be reflected on the tree hash containing that BLOB. The new tree will contain the edited BLOB and the references to the untouched BLOBs. This process produces a new commit with a new hash and new metadata [12].

Extracting data from the GIT objects is a quite complex process that requires extra programming effort and consumes more time. To accomplish this task, we examined five different data extraction tools that were designed to deal with software repositories shown in Table 4.1. These data extraction tools are described as follows:

4.1.1 Tidyextractors

The Tidyextractors is a python framework introduced by Becker et al [40]. This framework was built on the tidy data conceptual framework [43]. The Tidyextractors aimed to extract data from local Git repositories, Twitter user data, and email data with minimal effort and in a "tidy data format", which is the cleaned reshaped data that is ready for analysis. We followed the installation instructions through the pip package manager and by cloning the Tidyextractors repository to our local machine, and we tried to install the framework on different operating systems. Unfortunately, the installation process did not complete successfully for an unknown reason.

4.1.2 GHTorrent

The GHTorrent provides a scalable mirror of GitHub repositories in the form of MongoDB incremental data dumps. This mirror is distributed in a peer-to-peer BitTorrent network. The latest data reflected in the GHTorrent mirror was on 30-6-2019, which is outdated data and cannot be used to predict co-changing software entities [41].

4.1.3 CVSAnalY

The CVSAnaly is a data extraction tool that aimed to extract relevant data from software repositories. This tool was used by several researchers to collect data from software repositories [11][20]. This tool depends on the 2. x python version, which is replaced lately with the 3. x version [42].

4.1.4 GitPython

The GitPython is a python library that was created to deal with Git repositories. It simplifies the access of Git objects by reflecting the content of these objects into databases to be ready for use [36].

4.1.5 PyDriller

In our research, we used the PyDriller framework, which is a python framework Built upon the GitPython [36] framework to make the commit data extraction simpler. The PyDriller achieved 50% less LOC than GitPython to produce the same results. The result of this phase is a set of commits containing all the data related to the software system development process [24].

Table *4*.1: data extracting tools comparison

Tool	Extracting method	Advantages	Drawbacks	Environment
Tidyextractors	General data extracting framework	● Provides cleaned formatted data. ● Provides data with minimum effort. ● Extract data from multiple sources.	Suffers from bugs and errors	Python code
GHTorrent	Repository mirroring	● Provides a database of Git objects ready to use. ● Data stored in a peer-to-peer BitTorrent network which provides fast access and scalability	The database is out of date	Online service
CVSAnaly	Extracting tool	A reliable tool that was used in previous works successfully	Works on an old version of python	Stand-alone tool
GitPython	Extracting framework	● Reliable ● Fast relatively to other methods	Requires extra effort to produce outputs	Python code
PyDriller	Extracting framework	● Reliable ● Fewer lines of code to produce results	Slower respectively to other methods	Python code

4.2 Phase II: Data Preprocessing

The raw data extracted from software repositories requires several steps to be suitable for the mining process. Relevant data must be extracted, then cleaned by removing noise. After that, the relevant data should be transformed in a shape suited to the used algorithm. The data preprocessing phase includes the following steps [28].

4.2.1 Step 1: Feature Extraction

Git commit is an action made by the developer. To preserve the changes made on the system as a snapshot in the repository. Git Commits contain several attributes describing the event when the commit was made, i.e., the commit date, the author, the affected files ...etc. Some specific attributes form the features that will convey relationships among software entities. In this step, the relevant attributes, which are the set of affected files in each commit are extracted and then inserted into a relational database to be cleaned and preprocessed.

4.2.2 Step 2: Removing Misleading Commits

Some of the extracted commits are considered noise or misleading commits. In this step, the extracted commits from Phase I are revised to remove the commits considered as misleading data. Commits with one file edited do not represent any relationship among files. Also, commits with no affected files do not provide any knowledge and will affect the accuracy of calculating support and confidence. The other type of misleading commits is the commits with an extra-long affected files list. These commits come as a result of a software developer's bad practices. When a developer makes changes in a software system for a long period without committing the changes, the commit will contain an extra-long affected file list. The files in this list are might not be related therefore will produce misleading knowledge. We assigned the average number of files that the developer change for each software project to be a threshold of the commits that were considered useful. Commits with affected files numbers equal to or less than the threshold, and more than one file are considered as useful commits, and other commits will be removed.

4.2.3 Step 3: Coding File Names

File names are usually long strings that require a large portion of the main memory and so that require extra processing effort during the mining process. In this step, file names are replaced with integers. The original file names and their integer codes are stored in a relational database table to be retrieved later after the mining process.

4.2.4 Step 4: Removing Deleted Files

Commit data records each detail in the development process. Deleting files is one of the main operations made during the development process. The recorded deleted files in the commit data may lead to suggesting none existing files, which are false recommendations. In this step, the files tagged as deleted are removed to enhance the accuracy of the produced recommendations.

4.2.5 Step 5: Data Reduction

Old commits are less valuable than new commits for the knowledge-producing process [48]. Also, the vast amount of commits in the large projects are expensive to analyze and will increase the time of the co-change detection. In this step, the total amount of the commits is reduced to remove the valueless old commits and to reduce the time consumed during the analyzing phase.

4.3 Phase III: Analytical Processing

The final phase of the data mining framework is gaining valuable knowledge out of the cleaned data. After extracting and preprocessing the data, the altered frequent patterns analysis algorithm is applied to the preprocessed data. The applied algorithm produced rules with a single software entity on the antecedent side. After that, the rules with the same antecedent are aggregated to create larger consequent side rules, containing all the entities that frequently change with the entity in the antecedent. Later, these rules are chained to create the co-change path, which will guide the change propagation process. The following steps clarify each part of Phase III.

4.3.1 Step 1: Frequent Patterns Algorithm Applying

The final output of the preprocessing phase is a transactional database that contains sets of codes that represents the names of affected files in each commit. In this step, a frequent pattern algorithm is applied to generate patterns from each set of codes. The generated patterns are stored in a relational database along with its support count, which is the frequency number of that pattern in the whole transactional database.

4.3.2 Step 2: Rules Generation

After generating all the possible patterns from the transactional database, the generated patterns are used to create the rules that represent the relationships among software entities. The generating rules consist of the following sub-steps:

4.3.2.1 Substep 2.1: Evaluating the Patterns

In this step, the patterns are evaluated according to their support count threshold. Patterns with support counts less than the threshold specified are ignored.

4.3.2.2 Substep 2.2: Creating Antecedent and Consequent Lists

The frequent patterns generated and evaluated as an interesting pattern contain a set of items. The number of items per pattern ranges from one item to N number of items. The patterns with one item that satisfy the minimum evaluation criteria threshold are selected and inserted into the antecedents list. Patterns with more than one item and have a support count equal to or higher than the threshold are inserted in the consequent list.

4.3.2.3 Substep 2.3: Forming the Rules

In this step, the rules are created from the antecedent and consequent lists. A rule is a statement that describes the relationship between two disjoint sets of software entities. The rules produced in this step contain one item on the left side and one or more items on the right side. Forming a rule starts with a loop through the antecedent list, and another loop starts on the consequent list, to select patterns that contain the current antecedent and form a rule. The rules will be stored in permanent storage to be accessed in the following steps.

4.3.3 Step 3: Rules Aggregation

Not all of the generated rules are applicable [28]. The generated rules must be evaluated to avoid misleading recommendations. There are several criteria to evaluate rules (Support, confidence ...etc.). In this process, an evaluation criterion is chosen to pick the rules that may form valuable recommendations. The rules that satisfy the minimum evaluation criteria threshold and have the same antecedent are aggregated into a larger rule.

4.3.4 Step 4: Forming Change Propagation Path

The consequence of the aggregated rules is a set of software entities, some of which have their own rules. Therefore, a rule may trigger other rules and so on. In this process, we will create a path of changes depending on a starting point the software developer will choose during the maintenance process. The algorithm will chain all the affected rules and merge them as a long path of suggestions. The developer will change another entity and move to the entities affected and so on till the path ends.

4.5 Summary

In this chapter, we introduced the CPP approach and described each of its phases. The next chapter will describe the prototype implementation of the CPP approach.

Chapter 5

The Prototype Implementation

All the proposed tools in the literature are plugins or stand-alone desktop tools, which make them only available on one device at a time. However, being a plugin within an IDE makes it aimed at a narrow range of programming languages. In this chapter, we introduce the design of the Wide Assisting and Leading (WALead) tool as an implementation of the CPP approach. The WAlead is a web-based tool that can be accessed online from everywhere. The WAlead tool does not require previous installation or any other requirements. It only requires a stable internet connection and a machine with a web browser. After building the tool, we tested the correctness of its output. The output of the tool is unique (one item antecedent rules) it cannot be compared with other tools and approaches. Hence, we tested it manually by inserting a small dataset into the tool and testing the same dataset manually and comparing the output.

The following subsections describe the implementation of each phase in the CPP approach. Each phase is applied to the sample data extracted from the Git software repository, to examine the actual effect of each process on actual data.

5.1 Selecting the Environment

After revising several maintenance recommendation tools, we found that the tools were designed to work as plugins within another software, making it applicable to a specific type of software or a few programming languages. The other type of recommendation tool works as a stand-alone desktop application that requires pre-installation on the machine and requires locally stored data to work with. To overcome this insufficiency, we designed the WAlead tool to be hosted on a server and accessed online. The tool can receive a compressed file containing a Git repository or extract the data directly from Github. This feature will enable the developers' team to work remotely and share suggestions about software changes with no need to install applications or programs on their machines.

5.2 Choosing the Programming Languages

In the data extracting phase, we used PyDriller as a data-gathering tool, which is a sturdy data extraction python framework. Using PyDriller forced us to use python as a data processing language and python provides wide support for data mining tasks in the form of libraries and frameworks. The output of the data processing phase is stored in a relational database, where we chose the MySQL DBMS to be the data storage. The online tool requires a web-developing language. Therefore, we used PHP to be the back end of our tool along with python language. The PHP script receives the user requests, processes the request and provides outputs from the MySQL database as HTML layout. Figure 5.1 shows the main form of the WALead tool.

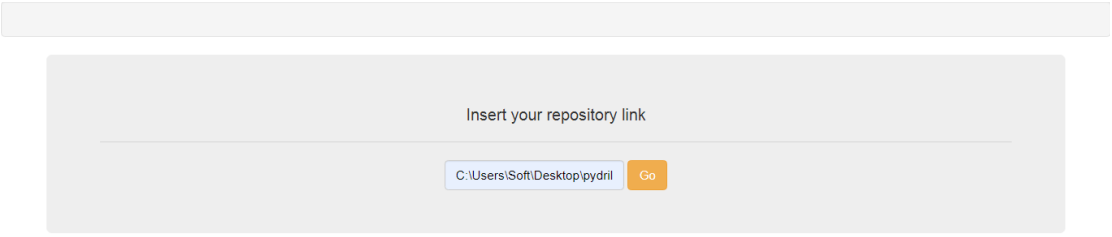

Figure 5.1 the main form of WALead tool

5.3 Phase I: Data Extraction

By using the Pydriller framework we extracted the commits data of five open-source projects with different sizes of development history, different purposes and, different programming languages. Table 5.1 describes the extracted projects and the differences among them.

Table 5.1: The projects extracted from Git repositories

Project name	**Launching date**	**Number of commits**	**Number of files**	**Programming language**	**project purpose**
Laravel	2011	6441	673	Php, Blade, Shell	Web application framework
Hbase	2007	18258	8920	Java, Ruby, Perl, Shell, Python, Thrift	Distributed datastore
Pydriller	2018	630	134	Python	Data mining framework
Cassandra	2009	25908	6157	Java, Python, HTML, Shell, GAP, Lex	Scalable row-store
React	2013	13776	3853	JavaScript, HTML, CSS, C++, TypeScript, CoffeeScript	User interface JavaScript library

5.4 Phase II: Data Preprocessing

The raw data collected in the data extraction phase in Figure 5.2 is not suitable for any type of data mining, therefore this data requires multiple preparation processes.

```
Commit date: 2020-11-30 14:35:15+01:00

Commit author email: spadini.davide@gmail.com

Commit author name: Davide Spadini

Commit message: (removing git obj from GitRepo)

Commit hash: #92511c8ee1e9356b8e487435b48f144a566e3acf

The affected files in this commit
**********************************
File name: git_repository.py
Change type: ModificationType.MODIFY
Change complexity: 43
File name: test_git_repository.py
Change type: ModificationType.MODIFY
Change complexity: 48
Commit date: 2020-11-30 14:51:27+01:00
```

Figure 5.2: Raw data extracted from Git Repository

The pre-processing phase consists of five different activities. These activities will guarantee the quality of the data passed to the analytical phase. The following steps describe the preprocessing activities:

5.4.1 Step 1: Feature Extraction

To detect hiddenly related software entities, we applied a frequent pattern analysis algorithm on the data extracted from the git repository. Frequent pattern analysis requires a Transactional database [29]. A transactional database is a set of transactions that are collected over a period. Each transaction contains a set of items that occur together. This set of items determines the relationships among items. The raw commit data collected from the git repository contains a similar format shown in Figure 5.3. The entire history of the software development process is stored as a set of commits, and each commit

contains a set of filenames that changed together. Hence, we created a transactional database from the extracted commits Figure 5.3.

```
['repository_mining.py', 'test_between_dates.py', 'test_repository_mining.py']
['git_repository.py', 'repository_mining.py', 'test_commit_filters.py']
['commit.py', 'git_repository.py', 'test_commit.py']
['commit.rst', 'configuration.rst', 'commit.py']
['tutorial.rst', 'commit.py', 'test_memory_consumption.py']
['requirements.txt', 'setup.py', 'test-requirements.txt']
['commit.py', 'git_repository.py', 'test_memory_consumption.py']
['commit.py', 'test-repos.zip', 'test_commit.py']
['git_repository.py', 'repository_mining.py', 'test_git_repository.py']
['commit.py', 'git_repository.py', 'repository_mining.py', 'test_git_repository.py']
['gitrepository.rst', 'git_repository.py', 'test-repos.zip', 'test_git_repository.py']
['gitrepository.rst', 'git_repository.py', 'test_git_repository.py']
['git_repository.py', 'test-repos.zip', 'test_git_repository.py']
['test-repos.zip', 'test_git_repository.py', 'test_memory_consumption.py']
['git_repository.py', 'test-repos.zip', 'test_git_repository.py']
['test_commit.py', 'test_developer.py', 'test_ranges.py']
['reference.rst', 'commit.py', 'git_repository.py']
['commit.py', 'git_repository.py', 'test_memory_consumption.py']
['commit.py', 'git_repository.py', 'test_git_repository.py']
['requirements.txt', 'setup.py', 'test_memory_consumption.py']
['repository_mining.py', 'requirements.txt', 'test_memory_consumption.py']
```

Figure 5.3: Transactional database represents all commits in the git repository

5.4.2 Step 2: Removing Misleading Commits

In this step, the commits that may affect the final result are eliminated. The commit with one file does not represent a relationship between files and will affect that certain file's support. Also, a commit with no files changed will produce empty items in the database and will also increase the total number of commits and induce misleading support for all files. As a bad practice, developers may make changes for a long period without committing them to the repository, after making a commit all the files changed in that long time will be added in one commit. Some of those files are unrelated to each other and the long transaction extracted from that commit requires a long time to produce subsets. In [13], the ROSE tool ignores commits with more than 30 files. In this step, we take the average number of files that the developer's team changes in all commits and make it a threshold for the considered commits. After applying this step to the five projects data the results shown in Table 5.2, only 11.83% to 47.65% are considered useful commits.

Table 5.2: useful commits in each project

project	All commits	Used commits	percentage
Laravel	6441	762	11.83%
Pydriller	630	205	32.50%
HBase	18258	8700	47.65%
React	13776	4452	32%
Cassandra	25908	7081	27%

5.4.3 Step 3: Coding Files Names

File names require a relatively large space of storage to save, also take a massive portion of the main memory while processed, leading to a delay in the mining process, especially when those names are long.

To solve this problem, we created a table containing file names and a serial number for each file name as shown in Figure 5.4. This table will be used to code and decode the file name during processing and storing. Numbers take less memory than file names and it is faster to process. In addition, in matters of scalability, adding new files to this table is easier than other coding methods.

```
[51, 52]
[29, 65, 51]
[51, 52, 84]
[10, 86]
[10, 89]
[79, 110]
[65, 120, 131]
[51, 96]
[10, 60]
[83, 51, 117]
[51, 110, 117]
[51, 117]
[83, 117, 129]
[21, 83]
```

Figure 5.4: A sample of transitions after coding

5.4.4 Step 4: Removing Deleted Files

During the software system lifetime, new files are added and other files are deleted. As we mentioned in Chapter 1, we attempt to create a co-change path that will guide the change propagation process. This process will be stopped if a deleted file appears in that path, preventing the rest of the files to be changed. In addition, if a commit contains only removed files, it will affect the pattern support calculation.

The examination of the five projects extracted data proved that 28% to 73% of the files mentioned in the developing history are deleted and must be ignored in the mining process Table 5.3

Table 5.3: The deleted files in each project

project	deleted files	active files	Total
Pydriller	52	82	134
Hbase	3525	5395	8920
Laravel	489	184	673
Cassandra	1744	4413	6157
React	2045	1808	3853

5.4.5 Step 5: Data Reduction

After the previous data preparation process, the amount of data is significantly reduced. However, for large projects, it is still large and expensive to generate patterns out of it. Furthermore, the older commits are less valuable for producing frequent patterns [48]. Therefore, a reduction process must be applied to reduce the time consumed during the mining process.

Reducing the data requires a unit to be used as a breaking point to divide data. We examined different units for data reduction. The first unit we examined was the Git tag (Git release). The Git tag is used to specify a point of time when an important event occurs in the project's developing history. After examining the releases in each project, we found that the number of commits in each release ranges between 0 and 40 commits as shown in Figure 5.5. The huge variation in the number of commits makes the release unusable as a data reduction unit.

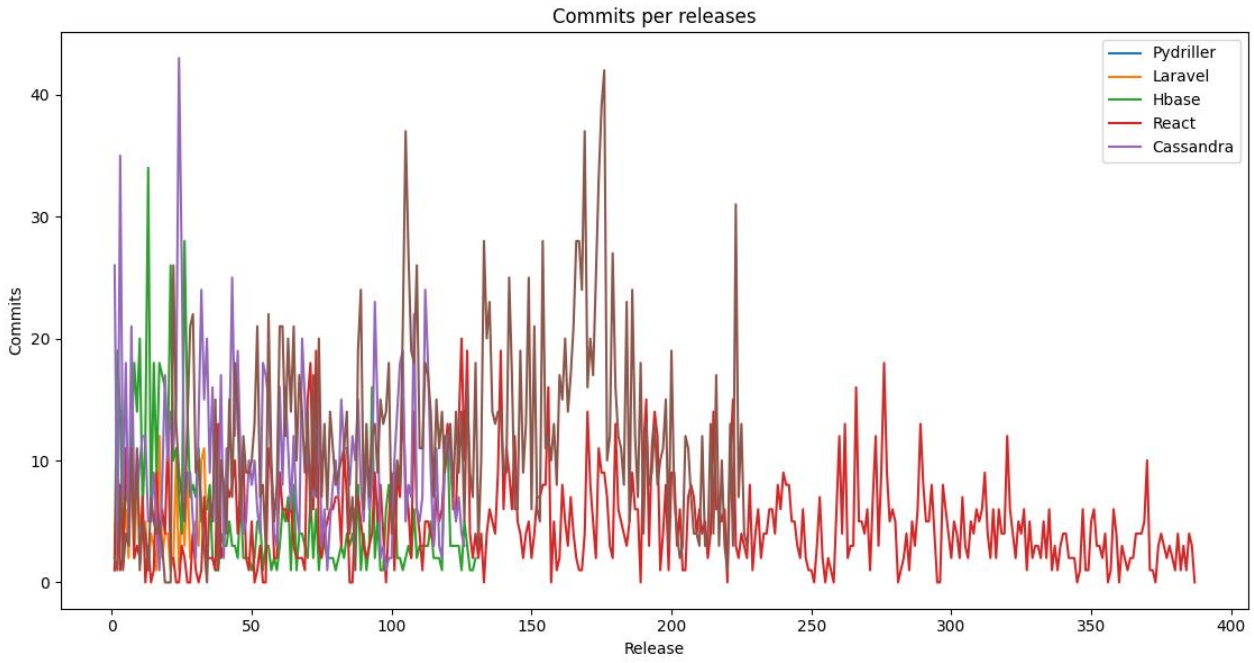

Figure 5.5: A plot describes the variation of commit number between releases

The second data reduction unit we examined is the year of development. The data in Table 5.4 presents each year of the developing history of the software system. In each year, there is a different number of commits, a different number of releases, and the behaviour of developers changes from time to time in each project. Hence the year cannot be used as a unit to divide the data. Also, the average changing files in each commit is not the same for each time in the same project, therefore the threshold of the considered number of files in each commit should be dynamic according to the developer's behaviour at each time.

In this step, we considered the number of commits as a dividing unit for data reduction. The number of considered commits is 1000 commits since it requires an acceptable time to generate frequent patterns.

Table 5.4: The number of commits releases and average files number changing in commits for each year

project	year	Commits	average files in a commit	releases in each year
HBASE	2007	235	9	2
HBASE	2008	559	7	0
HBASE	2009	664	6	0
HBASE	2010	769	8	55
HBASE	2011	1015	6	32

HBASE	2012	792	16	72
HBASE	2013	1059	10	97
HBASE	2014	924	12	65
HBASE	2015	898	9	44
HBASE	2016	947	9	37
HBASE	2017	1027	20	21
HBASE	2018	819	9	43
HBASE	2019	657	9	63
HBASE	2020	637	10	37
Pydriller	2018	117	4	17
Pydriller	2019	62	4	9
Pydriller	2020	95	5	10
Laravel	2012	445	6	31
Laravel	2013	43	15	12
Laravel	2014	128	6	4
Laravel	2015	54	8	11
Laravel	2016	57	3	13
Laravel	2017	19	3	12
Laravel	2018	18	3	10
Laravel	2019	30	3	16
Laravel	2020	15	4	43
react	2013	507	6	10
react	2014	507	6	12
react	2015	713	8	16
react	2016	800	6	18
react	2017	826	12	26
react	2018	594	8	27
react	2019	1359	8	20
react	2020	779	10	8
Cassandra	2009	666	8	1
Cassandra	2010	1134	7	36
Cassandra	2011	1545	6	41
Cassandra	2012	1012	9	36
Cassandra	2013	1020	7	49
Cassandra	2014	1164	7	44
Cassandra	2015	1257	8	54
Cassandra	2016	918	9	50
Cassandra	2017	487	7	28
Cassandra	2018	331	10	12
Cassandra	2019	190	12	15
Cassandra	2020	396	7	57

5.5 Phase III: Analytical Processing

This research aims to support the maintenance process by decreasing the cost and the time consumed while applying the required changes to the software system. Hence, the speed of the frequent pattern algorithm is a vital factor to achieve our goal. The algorithm used

should be fast and the output is suitable to generate single item antecedent rules. Moreover, scalability is an important feature to ensure continued maintenance support for the software system. The analytical phase in the CPP approach consists of the following four steps:

5.5.1 Step1: Applying Frequent Patterns Generation Algorithm

According to the comparative studies [38] [39] [45] on frequent pattern algorithms, the ECLAT algorithm achieved higher speed respectively to FP-growth and Apriori algorithm. Also, it is suitable for large databases, since there is no database scan in the ECLAT algorithm. We applied the ECLAT algorithm on the pre-processed data to generate frequent itemsets, each item set is presented with its frequency in the transactional database shown in Figure 5.6.

```
51=>>47
52=>>2
51,52=>>2
29=>>38
65=>>2
29,65=>>1
29,51=>>10
51,65=>>1
29,51,65=>>1
84=>>1
51,84=>>1
52,84=>>1
51,52,84=>>1
10=>>15
86=>>3
10,86=>>1
```

Figure 5.6: The result of applying ECLAT on our data

5.5.2 Step 2: Rules Generation

While applying changes to a software file, the software developer performs those changes on one file per time. Hence, we need a rule that describes the effect of changing one file on the other files in the software system. The off-shelf data mining algorithms produce rules that contain multiple items on the left side and another set of items on the right side.

This form of rules is not suitable for describing the effect of changing one file, because software developers cannot change multiple files at the same time.

In this process, we will produce rules with a single item in the antecedent (left-side) and a set of items in the consequence (right-side), by performing the following sub-steps:

5.5.2.1 Sub-step 1: Selecting the Interesting Frequent Patterns

In this step, we select the patterns that have support equal to or higher than the support threshold specified. Support threshold is usually set manually by revising the data characteristics [48]. After revising the data of five different projects. We discovered that each data sample has its characteristics, and applying a support threshold according to one project data on other projects' data is not feasible. In [13], the Support Count was used to measure the interestingness of the generated patterns. The Support Count of a pattern is the number of the transactions (commits) that contain that pattern. The Support count is easier for developers to understand rather than support. On the other hand, it applies to different projects data. To determine the Support Count threshold, we grouped the patterns with the same support to select the largest group. We found that the largest group has a support count of 2. Hence, we considered each pattern with a support count of 2 an interesting pattern.

5.5.2.2 Sup-step 2: Creating Antecedents and Consequents Lists

After selecting the interesting patterns, we extract the patterns that contain one item that satisfies the minimum support count threshold and insert it into a list along with their support count. This list will form the antecedents of the rules. The other patterns that contain more than one item and satisfy the support count threshold, are inserted in the candidate consequent list.

5.5.2.3 Sup-step 3: Forming Rules

The final step in this process is creating rules. We scan the antecedents list and search for each item we reach in the consequent list. If the item in the antecedents list appears in any patterns in the consequent, we form a rule out of the two patterns and calculate the confidence of the rule using the support of the two patterns.

5.5.3 Step 3: Rules Aggregation

"The confidence of an association rule is a percentage value that shows how frequently the rule head occurs among all the groups containing the rule body. The confidence value indicates how reliable this rule is." IBM [62].

To select the valuable rules, we assigned 50% as a confidence threshold to filter the generated rules. This means that the rules are 50% accurate. Setting a high confidence threshold will extremely reduce the number of recommendations, also setting a low confidence threshold will produce misleading recommendations.

After assigning the confidence threshold, we looped through the rules table, and combined the rules with the same antecedent, and satisfied the minimum confidence threshold to create a large rule.

5.5.4 Step 4: Change Propagation Path Creation

The final step in the CPP approach is to create the path that the developer will follow during the maintenance process. In this step, the developer will select a starting point, a software entity from the antecedents list. A list of the software entities affected by the starting point will be shown. The developer will examine, which file deserves to be changed. After that, the developer can choose one of the affected entities to continue the path. This process will be repeated till the path ends.

5.6 Summary

This chapter described the practical effort to implement the three phases of the CPP approach. Firstly, the data extracting, then preprocessing the data after that, analyzing the preprocessed data.

Chapter 6

Evaluation of The CPP Approach

This research aims to develop an approach to support the software development process by reducing the time and the cost of the maintenance process. To accomplish this aim, we proposed the CPP approach, which is a path created using the historical data of the software development process recorded on the Git repository. The path is the result of chaining multiple aggregated rules that describe the effect of changing a particular software entity. To prove the feasibility of the approach, we put it under test. We employed this approach in developing a software maintenance recommendation tool called WAlead tool. This tool can guide developers through the related changes after editing a software entity in a software system. To examine the feasibility of the WALead tool, we put it under three different tests. First, we compared the actual output of the tool with the theoretical description of the algorithm. Then, we tested the performance of the tool by measuring the time required for the tool to produce recommendations. Finally, we examined the effect of using the WALead tool during the maintenance process. This chapte describes the three experiments and the results of each test.

6.1 The WALead Tool Using Scenario

Using the WAlead tool is quite simple, the user requests the main form of the tool, and inserts the repository link as shown in Figures 6.1 and 6.2. The tool receives and processes the request. After that, the tool provides a list of files so the developer chose a starting point along with the time consumed and the number of the usable commits as shown in Figure 6.3. Then, the tool will show a list of affected files in Figure 6.4. The developer will check the files and change the files that should be changed. The next step is choosing one of the changed files and this process will be repeated till the end of the path as shown in Figure 6.5.

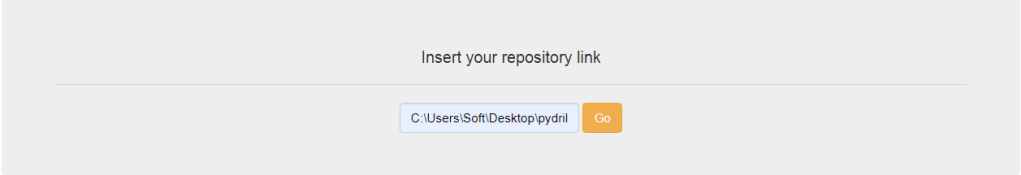

Figure 6.1 shows using a locally stored repository in the WAlead tool.

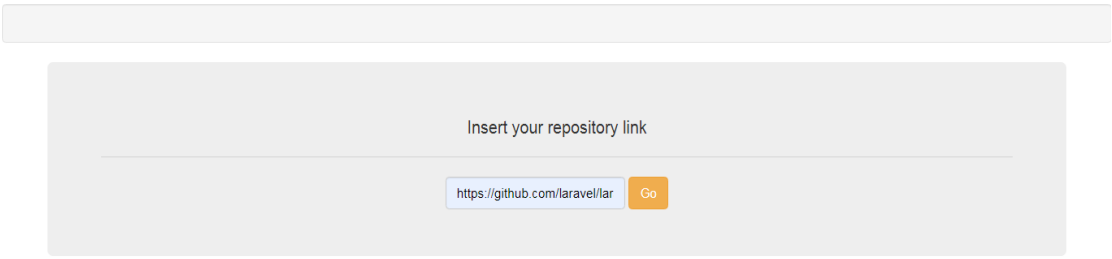

Figure 6.2 shows extracting data directly from GitHub using the WAlead tool.

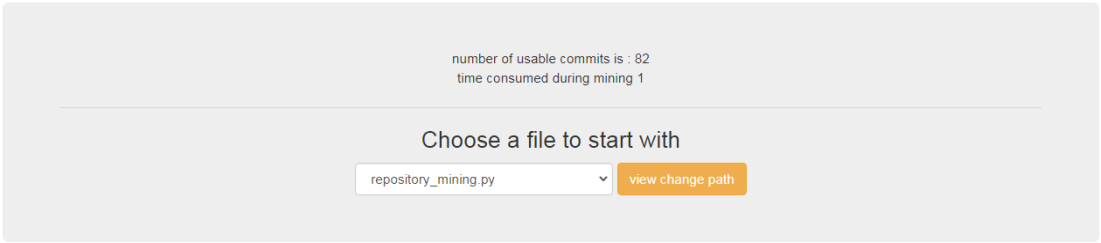

Figure 6.3 choosing the starting point of the path

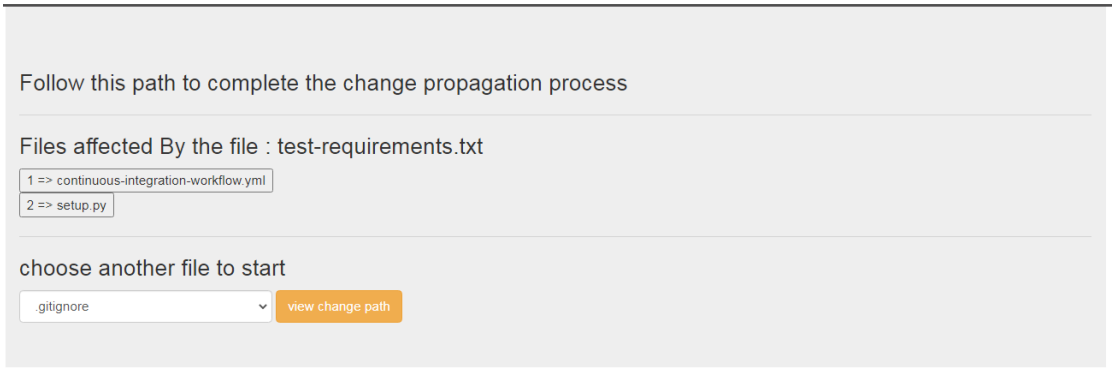

Figure 6.4 the list of the affected files by the changes made in the starting point

Figure 6.5 The end of the path where no more files will be changed

6.2 Experiment I: CPP Approach Validation

Before employing the WALead tool in supporting the software maintenance process. We have to validate the output of the tool. To perform this task, we applied the three phases of the CPP approach manually on a small dummy data set. Then, we used the same data set and inserted it into the WALead tool. After that, we compared the output of the manually generated recommendations and the onces produced by the WALead tool. The results revealed that the output of the WALead tool is identical to the output of the manually performed CPP approach phases.

6.3 Experiment II: Testing the Effect of CPP on Maintenance Process

WAlead tool is a recommendation tool that provides change suggestions built on the CPP approach and using the historical data of the development process. To prove the feasibility of the WAlead tool we put it under test by using it to support the maintenance process of a simple attending registration system. We invited six junior PHP developers to add a feature to an existing system built using PHP language.

We prepared a simulation for the World Wide Web environment. First, we prepared a server to client network containing seven computers. After that, we installed the WAlead tool on one machine to be the service provider. Then, we installed the Komodo text editor and MAMP server on the other six machines. Later, we installed a copy of the system that will be edited during the experiment in the MAMP server.

We made the experiment in 24-5-2021 at the Higher Institute of Engineering Professions Almajory in Lab-2 the experiment started at 10:15 am. We split the developers into two groups. The first group was allowed to use the WAlead tool. The second group was asked to figure the related changes on their own. We gave the developers a simple task, which is adding a feature to an existing system. The experiment ended at 11:40, and we recorded the time consumed by each developer. Table 6.1 shows the results of the conducted experiment.

Table 6.1 The Time Consumed By Each Developer During The Experiment

Developer name	Tool	Start time	Ending time	Time consumed
Rela	Yes	10:15	10:52	00:30
Aya	Yes	10:15	10:40	00:25
Asma	Yes	10:15	10:30	0:15

Monia	No	10:15	11:13	0:58
Aisha	No	10:15	10:40	0:25
Amal	No	10:15	11:10	00:55

6.4 Experiment III: Testing the WAlead Tool Performance

This research aims to find an approach to reduce the time and cost of the maintenance process. The performance of the CPP approach can be measured by the time used during the recommendation production process. During this experiment, we tested the performance of the WAlead tool. The experiment was conducted on the data of the five projects we used previously in this research. The time consumed during the whole three phases depends on three main factors: number of the used commits, the number of files in each commit, the number of characters in the names of the files. The results are shown in Table 6.2. After that, we tested the tool on fresh copy extracted directly from GitHub and found that the efficiency of the tool in this scenario depends on the quality of the network.

Table 6.2: testing the tool on locally stored repositories

Project	**Number of usable commits**	**average files in a commit**	**Time consumed**
PyDriller	82	4	1 minute
Laravel	229	6	7 minutes
React	1000	8	24 minutes
Hbase	1000	10	1 hour
Cassandra	1000	8	34 minutes

6.5 Results Discussion

After implementing and testing the CPP approach. We review and discuss the results of each phase and try to answer the research questions. We show the results of each phase starting with the data gathering phase and ending with the experiment conducting.

6.5.1 Data Extraction Phase Results

In this phase, we made a comparative study among five different software repository data extracting tools in Table 4.1 and concluded that PyDriller is the most suitable tool for this task. Using the PyDriller tool, we extracted the data of five different open source projects

Table 5.1. The data extracted in this phase is raw commits data containing a massive amount of information, some of this information is useful for our purpose and other information are useful for other tasks. Hence, the data extracted requires several preprocessing stages to produce knowledge.

6.5.2 Data Preprocessing Phase Results

This phase prepares the extracted data to be analyzed by a data mining technique. First, the features were extracted from the raw data. to make the data suitable for frequent pattern analysis in the analytical phase. Most of the data attached to each commit such as date, committer, message ...etc. was abandoned. The names of the files edited in each commit were kept to create a transactional database. After that, the extra-long commits were removed along with the commits with one file edited. Then the files that were tagged as deleted from each commit were discarded. Later the number of transactions was reduced to minimize the time consumed by the analytical phase. The final step is coding the names of the files to optimize the mining process performance.

During the preprocessing of the data extracted from the five different projects we concluded the following:

After removing the extra-long and short commits. We discovered that the useful commits are between 11.83% to 47.65% of the total number of commits. Table 6.3 presents the number of extracted and discarded commits. Figures 6.6, 6.7, 6.8, 6.9, 6.10 show pie charts for the percentage of the commits used in each project. Hence, we conclude that the number of usable commits differs from one software project to another.

Table 6.3: the number of used commits for each project

project	**all commits**	**used commits**	**percentage**
laravel	6441	762	11.83%
pydriller	630	205	32.50%
hbase	18258	8700	47.65%
react	13776	4452	32%
Cassandra	25908	7081	27%

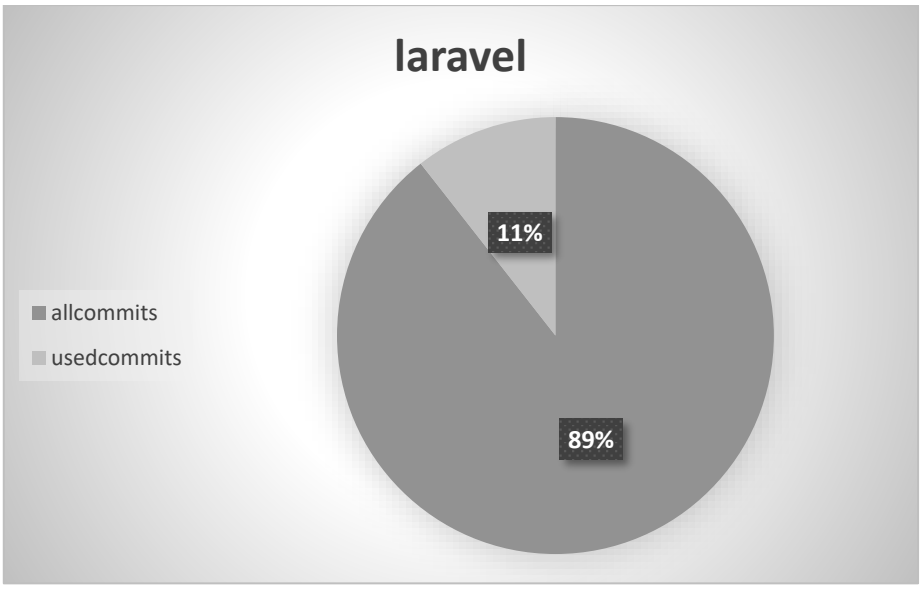

Figure 6.6: shows the percentage of the usable commits extracted from the project Laravel

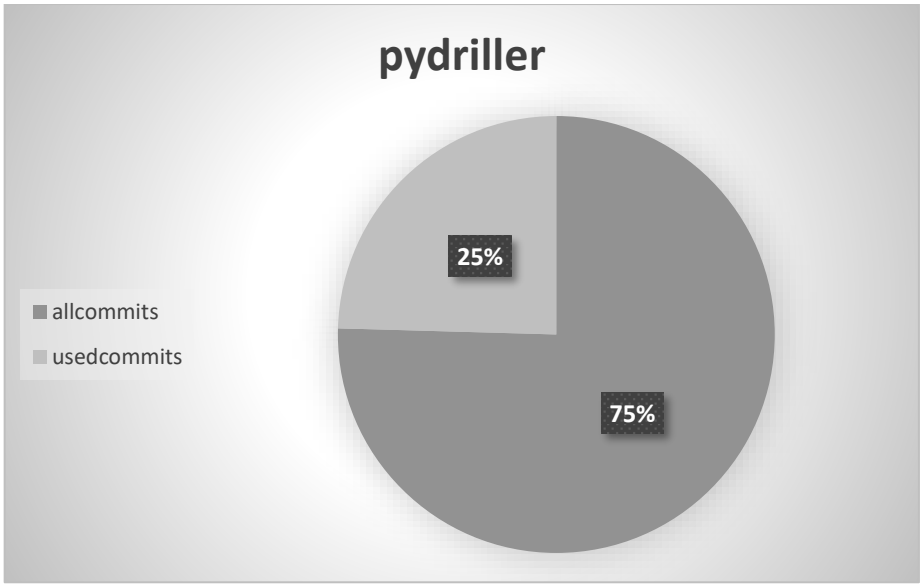

Figure 6.7: shows the percentage of the usable commits extracted from the project PyDriller

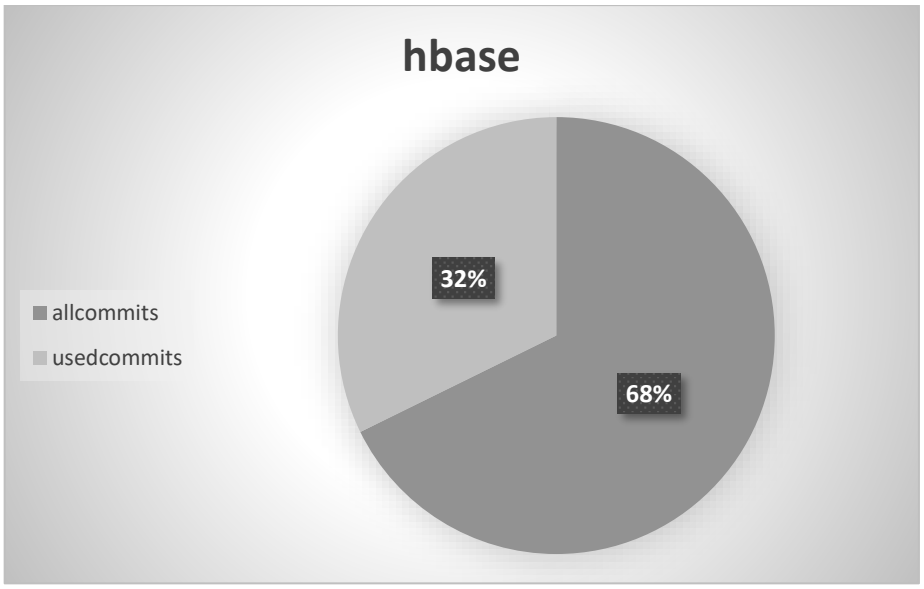

Figure 6.8: shows the percentage of the usable 5commits extracted from the project Hbase

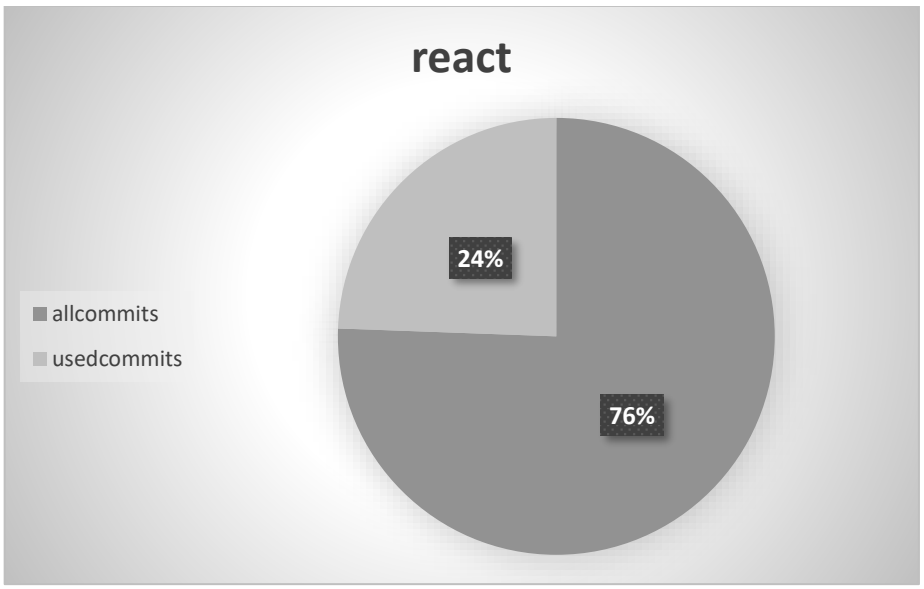

Figure 6.9: shows the percentage of the usable commits extracted from the project React

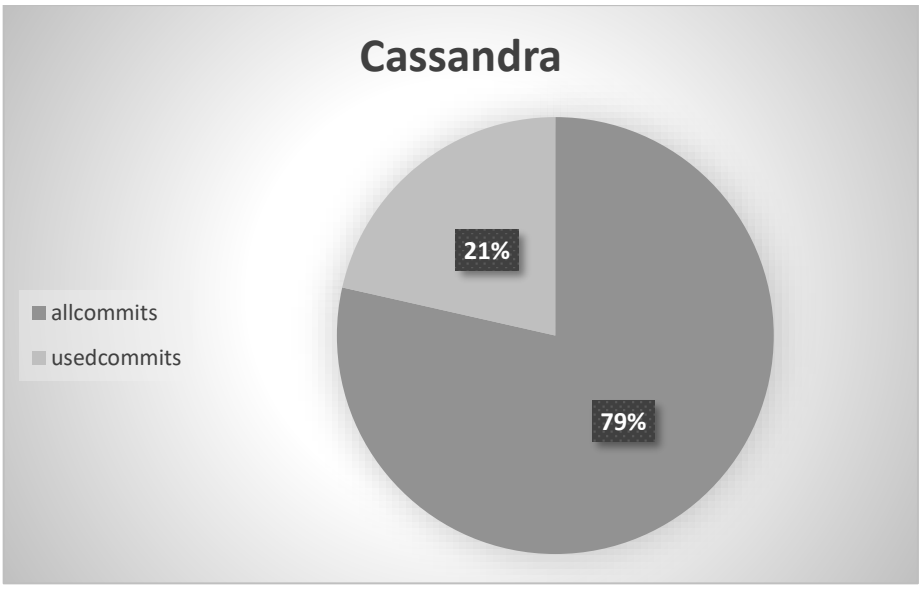

Figure 6.10: shows the percentage of the usable commits extracted from the project Laravel

Discarding the deleted files from the extracted commits revealed that 28% to 73% of the files recorded within the commits metadata are deleted during the development process. Table 6.4 presents the number of files of each project and the deleted files. Figures 6.11, 6.12, 6.13, 6.14 show pie charts for the deleted files of each project.

Table 6.4 shows the number of files for each project and deleted files

Project	**Deleted Files**	**Active Files**	**Total**
Pydriller	52	82	134
Hbase	3525	5395	8920
Laravel	489	184	673
Cassandra	1744	4413	6157
React	2045	1808	3853

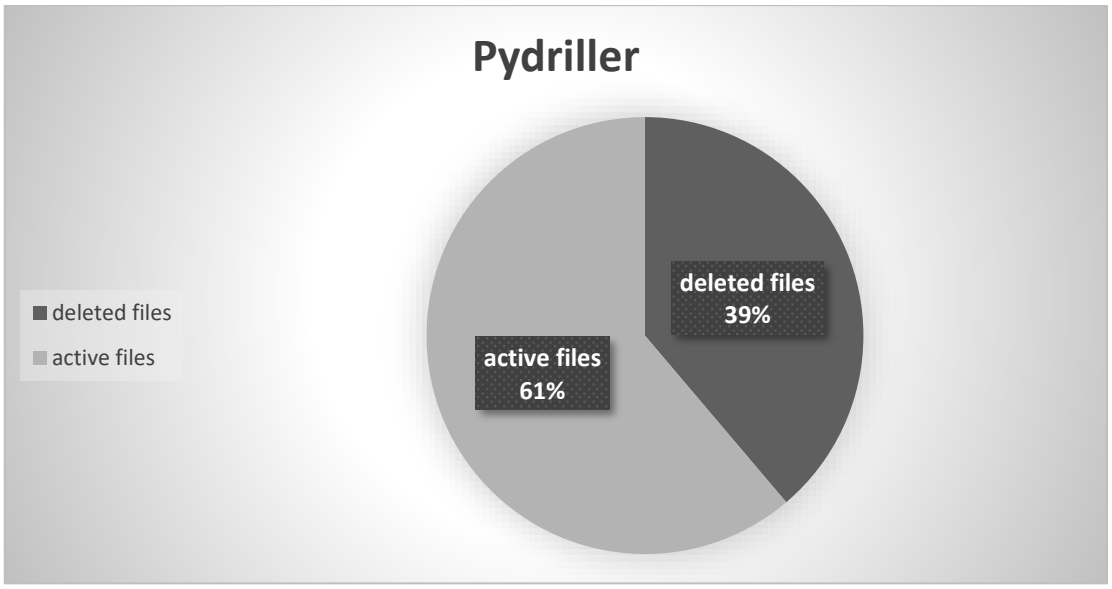

Figure 6.11: shows the percentage of the deleted files to the total files of the PyDriller project

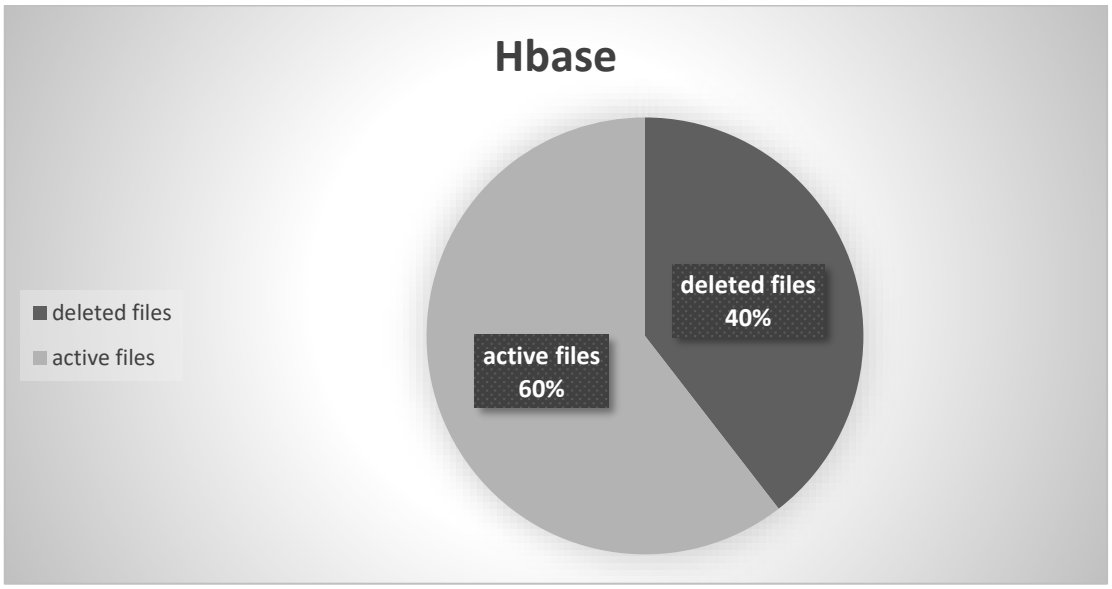

Figure 6.12: shows the percentage of the deleted files to the total files of the Hbase project

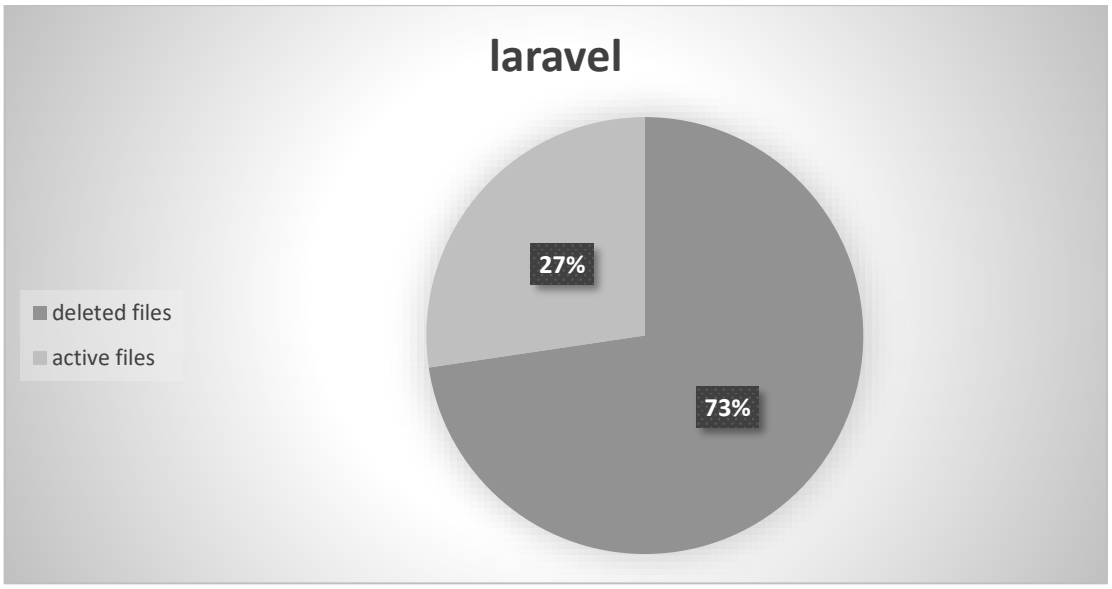

Figure 6.13: shows the percentage of the deleted files to the total files of the Laravel project

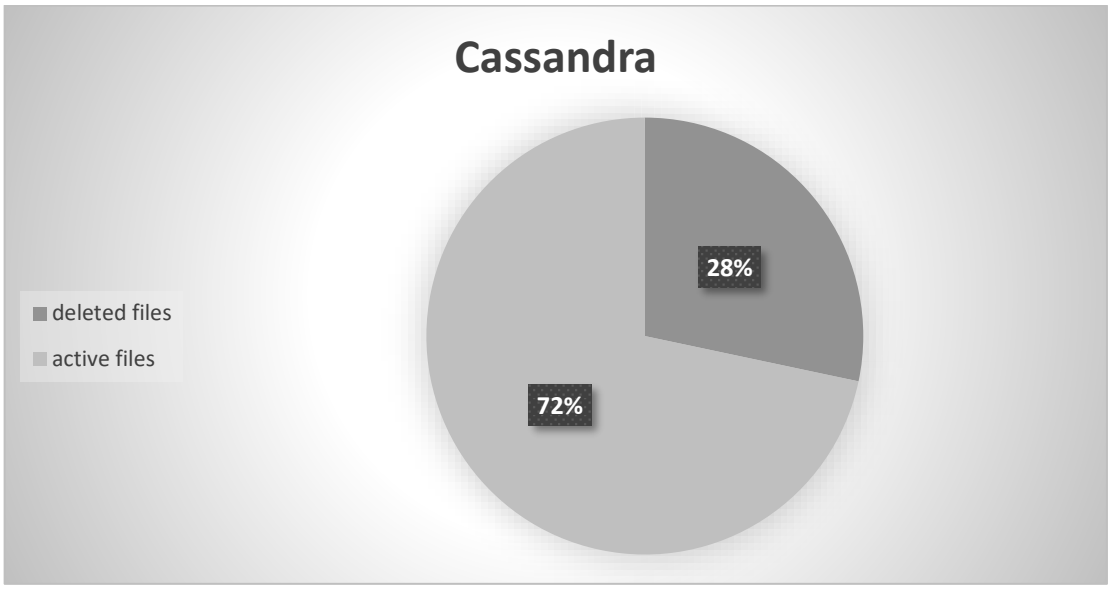

Figure 6.14: shows the percentage of the deleted files to the total files of the Cassandra project

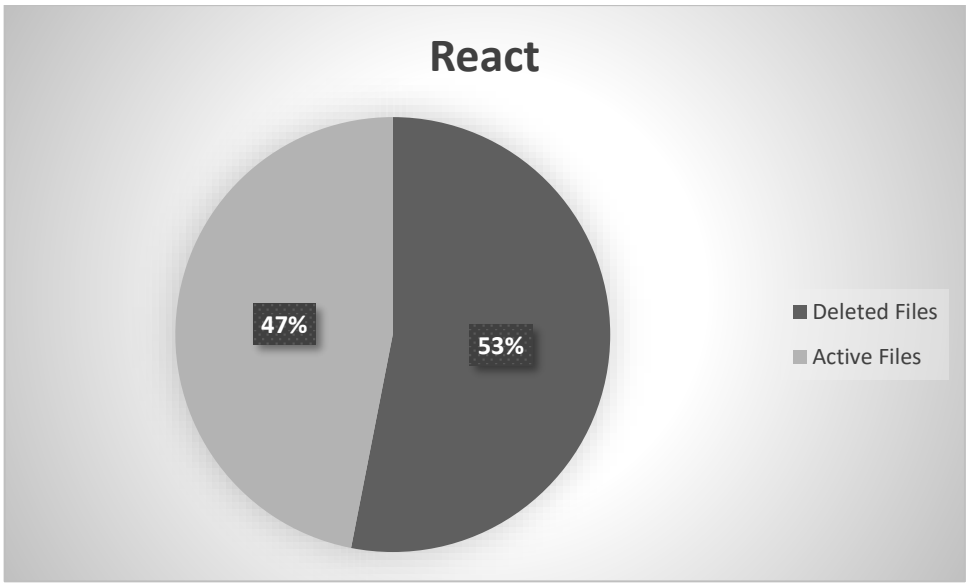

Figure 6.15: shows the percentage of the deleted files to the total files of the React project

The next preprocessing step is data reduction. In this step, we concluded that old commits are not useful and the huge amount of data requires a long time to be analyzed. While we aim to reduce the time consumed and provide proper accuracy. We reduced the number of commits analyzed to the last 1000 commits. Before that, we tested several units to use it for data reduction. First, we tested the releases. Release or tag is a point of the project history when a major event occurs. We found that the number of commits in each release differs from project to project and from release to release. Figure 5.5 presents a chart of each release's commit number in our five projects, which varies from 1 commit per release to 40 commits per release. After revising the number of commits in each release we concluded that the release is not suitable for reducing the number of commits

The next reduction unit we tested was the year of development. We revised the number of commits in each year of software development. We found that the number of commits in each year is extremely varying and cannot be used as a reduction unit. Table 5.4 shows the number of commits, the number of releases and the average number of files changed in each year of development.

Finally, we used the number of commits as a data reduction unit and we set the latest 1000 commits hence 1000 commits requires a maximum of 1 hour to be analyzed and contain enough commits to produce knowledge.

The final step of the preprocessing phase is coding. File names preserve a huge portion of the main memory during the processing. To reduce the memory usage, we gave every

file a code this code maximumly contains 5 digits. Coding file names reduced the time consumed during analyzing from 15 hours to 1 hour in worst cases.

In a conclusion, the relevant data that is useful in co-change prediction is the list of changed file names in each commit discarding the deleted files, the latest 1000 commits and the commits that contain changed files less than or equal to the average number of files changed in each commits and, greater than one file.

6.5.3 Applying Frequent Patterns Algorithm Results

In this phase, we revised previous comparative studies [38][39][45] and concluded that the ECLAT algorithm is the optimal algorithm for our task. ECLAT scans the database 1 time, the output of ECLAT is scalable and, the variation we made of ECLAT provides a single item antecedent rule, which is the main component in our approach (Change Propagation Path).

6.6 Comparing WALead tool with the existing proposed tools

In this research we proposed the CPP approach to be a usable approach in detecting co-changing software entities. We built WALead tool to prove the feasibility of the CPP approach, while designing this tool we tried to avoid the shortcomings of the existing tools. In table 6.5 a comparison preview among the proposed tools and WALead tool

Table 6.5: A comparison among the proposed tools and WALead tool

Tool	**Accuracy**	**Scalable**	**Availability**	**Cost effectiveness**	**Granularity level**	**Data source**	**Algorithm**
ROSE [13]	Depends on the support threshold	No	Eclips IDE Plugin	Consumes long time to produce patterns	Code level elements	CVS	Apriori
Ramadhani [14]	Depends on the support threshold	No	Standalone tool	Scans the database twice to produce patterns	File level	Git	FP-growth
Hyper-rules [4]	13% to 90% higher than previous work	No	Standalone tool	Yes	File level	Git	Hyper-rules
TARMAQ [27]	-	No	Standalone tool	Yes	File level	Git	TARMAQ
Coming [44]	-	No	Plugin within other tools	-	Code level elements	Git	Frequent patterns analysis
Ruffle [47]	70% to 80%	No	Standalone tool	-	Code level elements	-	-
WALead	Depends on the support threshold	Yes	Web based system	Yes	File level	Git	ECLAT

Chapter 7

Conclusion and Recommendations

7.1 Conclusion

The main aim of this research is to find a solution to support software maintenance processes by reducing the time and cost of this process. To reach the aim of this research we employed MSR to detect co-changes among software entities, which will reduce the time consumed while searching for related changes and eliminate the cost of hiring highly paid senior developers to guide the development team through the change propagation process. We conducted deductive research on quantities data to test the effect of detecting co-changes using our proposed CPP approach.

The CPP approach consists of three main phases. In Phase I, the commits' data stored within the Git repository is extracted. We conducted a comparative study among five different data extraction tools, from which we selected the appropoarte tool to do this task. The study revealed that PyDriller is the most compatible tool for our task. Phase II is the data preparation phase, in which we eliminated the noise, transformed the data and reduced the data amount. The output of Phase II is a transactional database that contains lists of coded file names. Phase III is the core of the CPP approach, where the preprocessed data is transformed into recommendations that guide developers to propagate changes correctly. In Phase III, the data within the transactional database are transformed into patterns using the ECLAT algorithm. After that, the patterns are evaluated to select the interesting patterns that may form knowledge. Then, interesting patterns are used to create rules that describe the relationships among files. The rules produced in this stage contains one item on the antecedent side. The interesting rules with the same antecedent are aggregated to create larger rules. Finally, the aggregated rules are used according to the editing scenario to create the change propagation path.

To prove the feasibility of the CPP approach we built a recommendations tool, which is called the WALead tool. The tool is a web-based tool that uses the data stored in the Git repository and provides recommendations to the software developers during the maintenance process. These recommendations are produced according to the software editing scenario. After building the WALead tool, we conducted three different

experiments to prove the feasibility of the CPP approach. First, we tested the output of the WALead tool and compared it with the expected output using dummy data. After that, we tested the effect of using the WALead tool during the maintenance process. We managed to reduce the time consumed by 50% and eliminated the cost of hiring highly paid a senior developers to guide the development process. Finally, we tested the performance of the WALead tool by recording the time required to produce recommendations for five different software projects. The results revealed that the time consumed during the production of the recommendation is affected by three factors the number of commits extracted from the software repository, the average number of files in each commit and the size of the string the presents the file names.

We attemped to answer the following questions during the conduction of this research.

RQ1: To what extent the time and cost can be reduced by detecting co-changes during the maintenance processes?

The results of the experiment show that our approach reduced the time of the software maintenance process by 50%. In addition, the cost was reduced by eliminating the role of the guiding senior developer.

RQ2: What is the optimal software repository data extracting tool?

Extracting data from software repositories is a complex process and is out of the scope of this research. Hence, we conducted a comparative study among six different extracting tools. We concluded that the PyDriller is the most suitable tool for our purpose.

RQ3: What are the features of the data extracted from the software repositories that will produce knowledge?

Data is the base that our research is built on. Selecting the right pieces of data guarantees more accurate results. After reviewing the data of five different projects. We concluded that the relevant data (the features) are the list of the edited entities in each commit regardless of the entities that are tagged as deleted in the development history. On the other hand, a huge portion of the commits is considered as noise. Extralong commits with one edit entity and old commits. We set 1000 commits as a limit for the extracted commits because the old commits are not valuable and more than 1000 commits require a long time to be processed.

RQ4: What factors are vital to selecting a data mining algorithm for producing required knowledge for the CPP approach?

This research aims to propose a solution to reduce the time of co-changed software entities. Therefore, the speed of the algorithm is a vital factor to select the data mining algorithm. Software development processes are usually continuous processes. Changes on the software system are continually made to add a new feature or to fix defects. Hence, the output of the mining algorithm must be scalable to support continual development. After revising several mining algorithms and according to previous studies, we selected the ECLAT algorithm. The ECLAT algorithm scans the database on time, therefore, it is considered faster relatively compared to the other mining algorithms. The ECLAT algorithm produces a scalable output, which can be used incrementally to support continual development. The output of the ECLAT algorithm is suitable to produce single item antecedent association rules which is the main component to create the Change Propagation Path.

7.2 Recommendations

Given the results discussed in Chapter 6, and to obtain accurate results from the commit metadata, we recommend developers avoid editing and committing unrelated files. Moreover, to avoid editing files for a long time without committing. Finally, we recommend developers avoid committing after editing one file on its own. Those practices that we recommend to avoid, forms valueless commits that will produce misleading change suggestions.

7.3 Future Work

In this research, we tested the feasibility of our approach on file-level software entities. As future work, we attend to add code parsers to test the validity of the CPP approach on the source code entity level. While we aim to reduce the time of the maintenance process, we attend to design a software repository data extracting tool based on a compiled language to avoid the latency induced by interpreted language such as Python.

Appendices

The data mining portion of the WALead tool written in Python programming language :

```
from pydriller import RepositoryMining
import time
import mysql.connector
start = time.time()
# -------------Real Shit goes here :)--------


class RawData:
    link = ""
    avgFilesPerCommit = 0
    commits = []
    filesStatue = {}
    codedFiles = {}
    codedCommits = []
    start = 0
    end = 0

    def __init__(self, link, start, end):
        self.link = link
        self.start = start
        self.end = end
        self.extractCommits()
        self.removeUnwantedFiles()
        self.codeFiles()
        self.codeCommits()
        self.commits[start:end]
        self.insertCodedFiles()

    def extractCommits(self):
        sum = 0
        for commit in
RepositoryMining(self.link).traverse_commits():
            itemsInCommit = []
            for modification in commit.modifications:
                itemsInCommit.append(modification.filename)

self.filesStatue[modification.filename]=str(modification.change_
type)
            if len(itemsInCommit)>1:
                sum+=len(itemsInCommit)
                self.commits.append(itemsInCommit)
        self.avgFilesPerCommit=round(sum/len(self.commits))
        self.commits.reverse()

    def removeUnwantedFiles(self):
        newCommits=[]
```

```
        for commits in self.commits:
            itemList=[]
            for item in commits:
               if
self.filesStatue[item]!="ModificationType.DELETE":
                   itemList.append(item)
            if len(itemList)>2 and
len(itemList)<=self.avgFilesPerCommit:
                newCommits.append(itemList)
        self.commits=newCommits
        self.commits=self.commits[self.start:self.end]
    def codeFiles(self):
        i = 0
        for x in self.filesStatue:
            i += 1
            self.codedFiles[x] = i

    def codeCommits(self):
        for i in self.commits:
            itemlist=[]
            for j in i:
                itemlist.append(self.codedFiles[j])
            self.codedCommits.append(itemlist)


    def insertFile(self,file, code):
        mydb = mysql.connector.connect(
            host="localhost",
            user="root",
            password="",
            database="mytool"
        )
        mycursor = mydb.cursor()
        sql = "INSERT INTO files (file_name, file_code) VALUES
(%s, %s)"
        val = (file, code)
        mycursor.execute(sql, val)
        mydb.commit()
    def insertCodedFiles(self):
        for i in self.codedFiles:
            self.insertFile(i,self.codedFiles[i])

class pattern: # frequent pattern list content
    def __init__(self, files, suppCount):
        self.files = files
        self.suppCount = suppCount
    suppCount = 0
    support=0
    files = []
```

```
    def patternSupport(self, commitsSize):
        self.support=self.suppCount/commitsSize
class patterns:
    patts=[]
    averageSupport=0
    commits=[]
    def __init__(self,commits):
        self.commits=commits
        self.patternGenerator()
        self.removeUglyOne()
        self.addSupport()
        self.calculateAvgSupport()

    def subsetter(self,l):  # returns a list of all sublissts
for a given lsit
        base = []
        lists = [base]
        for i in range(len(l)):
            orig = lists[:]
            new = l[i]
            for j in range(len(lists)):
                lists[j] = lists[j] + [new]
            lists = orig + lists
        lists.remove(lists[0])
        return lists

    def calSupport(transCount, patterns):
        for i in patterns:
            i.support = round((i.suppCount - 1) / transCount, 2)
        return patterns

    def removeUglyOne(self):
        for i in self.patts:
            i.suppCount -=1

    def patternGenerator(self):
        t=pattern([],0)
        self.patts.append(t)
        for i in self.commits:
            newList=self.subsetter(i)
            for j in newList:
                c=0
                for k in self.patts:
                    c+=1
                    if set(k.files)==set(j):
                        k.suppCount +=1
                        break
                    elif c == len(self.patts):
                        t=pattern(j,1)
```

```
                        self.patts.append(t)
        self.patts.remove(self.patts[0])
    def addSupport(self):
        for i in self.patts:
            i.support=i.suppCount/len(self.commits)

    def calculateAvgSupport(self):
        self.averageSupport =2
class rule:
    def __init__(self,left , right, confidence, support):
        self.left=left
        self.right=right
        self.confidence=confidence
        self.support=support
class rules:
    patterns=[]
    minconf=1
    minsup=0
    rules=[]
    lsides=[]
    rsides=[]

    def __init__(self,patterns,items,minsup):
        self.items=items
        self.minsup=minsup
        self.patterns=patterns
        self.createRules()

    def createRules(self):
        for i in self.patterns:
                if len(i.files) <= 1:
                    if i.suppCount >=2:
                        self.lsides.append(i)
                else:
                    if i.suppCount >= 2:
                     self.rsides.append(i)
        for i in self.lsides:
            for j in self.rsides:
                if i.files[0] in j.files :
                    conf= j.support/i.support
                    r = rule(i.files[0],j.files,conf,j.support)
                    if r.confidence>=0.5:
                        self.rules.append(r)




class AgRules:
    rules=[]
```

```
    agrules={}
    codedItems={}

    def __init__(self,rules):
        self.rules=rules
        self.agit()
        self.insertRules()

    def agit(self):
        for i in self.rules:
            if i.left in self.agrules.keys():
                self.agrules[i.left] += i.right
            else:
                self.agrules[i.left] = i.right

    def insertRule(self,lside,rside):
        res = []
        for i in rside:
            if i not in res and i != lside:
                res.append(i)
        mydb = mysql.connector.connect(
            host="localhost",
            user="root",
            password="",
            database="mytool"
        )

        mycursor = mydb.cursor()
        rr=[str(int) for int in res]
        r= ','.join(rr)
        sql = "INSERT INTO rules (lside, rside) VALUES (%s, %s)"
        val = (lside, r)
        mycursor.execute(sql, val)

        mydb.commit()
    def insertRules(self):
        for i in self.agrules:
            self.insertRule(i,self.agrules[i])
```

The implementation of the web services portion of WALead tool written in Php programming language and HTML:

```
<?php
function viewMain()
{
    ?>
    <form action="<?php echo $_SERVER['self']; ?>"
method="post">
        <input type="text" name="link">
```

```
        <input type="hidden" name="act" value="link">
        <input type="submit" value="Go">
    </form>
    <?php
}
function insertLink()
{
$myfile = fopen("link.txt", "w") or die("Unable to open file!");
$txt = $_POST['link'];
fwrite($myfile, $txt);
fclose($myfile);
}
function runTool()
{
$command = escapeshellcmd('python
C:/Users/Soft/research/venv/extracting.py');
$output = shell_exec($command);
echo $output;
}
?>
<!DOCTYPE html>
<html>
    <?php
    include('code.php');
    $act=$_POST['act'];
    switch($act)
    {
        case "":
            viewMain();
            break;
        case "link":
            insertLink();
            runTool();
            break;
        default:
            echo"<h1>something went wrong</h1>";
    }
    ?>
</html>
```

References

[1] D. Güemes-Peña, C. López-Nozal, R. Marticorena-Sánchez and J. Maudes-Raedo, "Emerging topics in mining software repositories", *Progress in Artificial Intelligence*, vol. 7, no. 3, pp. 237-247, 2018. Available: 10.1007/s13748-018-0147-7 [Accessed 13 September 2021].

[2] Z. Jiang, Y. Wang, H. Zhong and N. Meng, "Automatic method change suggestion to complement multi-entity edits", *Journal of Systems and Software*, vol. 159, p. 110441, 2020. Available: 10.1016/j.jss.2019.110441 [Accessed 13 September 2021].

[3] N. Ajienka, A. Capiluppi and S. Counsell, "An empirical study on the interplay between semantic coupling and co-change of software classes", *Empirical Software Engineering*, vol. 23, no. 3, pp. 1791-1825, 2017. Available: 10.1007/s10664-017-9569-2 [Accessed 13 September 2021].

[4] T. Rolfsnes, L. Moonen, S. Alesio, R. Behjati and D. Binkley, "Aggregating Association Rules to Improve Change Recommendation", *Empirical Software Engineering*, vol. 23, no. 2, pp. 987-1035, 2017. Available: 10.1007/s10664-017-9560-y [Accessed 14 September 2021].

[5] M. Islam, M. Islam, M. Mondal, B. Roy, C. Roy and K. Schneider, "[Research Paper] Detecting Evolutionary Coupling Using Transitive Association Rules", *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018. Available: 10.1109/scam.2018.00020 [Accessed 14 September 2021].

[6] Y. Wang, N. Meng and H. Zhong, "An Empirical Study of Multi-entity Changes in Real Bug Fixes", *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018. Available: 10.1109/icsme.2018.00038 [Accessed 14 September 2021].

[7] A. TOSUN and B. Romero, "Predicting Co-Changed Files: An External, Conceptual Replication", *Celal Bayar Üniversitesi Fen Bilimleri Dergisi*, pp. 161-169, 2019. Available: 10.18466/cbayarfbe.489291 [Accessed 14 September 2021].

[8] L. Vidacs and M. Pinzger, "Co-evolution analysis of production and test code by learning association rules of changes", *2018 IEEE Workshop on Machine*

Learning Techniques for Software Quality Evaluation (MaLTeSQuE), 2018. Available: 10.1109/maltesque.2018.8368456 [Accessed 14 September 2021].

[9] A. Hassan and R. Holt, "Replaying development history to assess the effectiveness of change propagation tools", *Empirical Software Engineering*, vol. 11, no. 3, pp. 335-367, 2006. Available: 10.1007/s10664-006-9006-4 [Accessed 14 September 2021].

[10] I. Sommerville, *Software engineering*. Boston, Massachusetts: Addison-Wesley, 2011

[11] I. Wiese et al., "Using contextual information to predict co-changes", *Journal of Systems and Software*, vol. 128, pp. 220-235, 2017. Available: 10.1016/j.jss.2016.07.016 [Accessed 14 September 2021].

[12] S. Chacon and B. Straub, *Pro Git*, 2nd ed. New York: Springer Natuer..

[13] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl, "Mining version histories to guide software changes", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429-445, 2005. Available: 10.1109/tse.2005.72 [Accessed 14 September 2021].

[14] J. Ramadani, "Mining software repositories for coupled changes", *Elib.uni-stuttgart.de*, 2021. [Online]. Available: https://elib.uni-stuttgart.de/handle/11682/9264. [Accessed: 14- Sep- 2021].

[15] "What is bug tracking? | IBM", *Ibm.com*, 2021. [Online]. Available: https://www.ibm.com/topics/bug-tracking. [Accessed: 13- Sep- 2021].

[16] A. Hindle and D. German, "SCQL", *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05*, 2005. Available: 10.1145/1083142.1083161 [Accessed 14 September 2021].

[17] A. Hassan, "The road ahead for Mining Software Repositories", *2008 Frontiers of Software Maintenance*, 2008. Available: 10.1109/fosm.2008.4659248 [Accessed 13 September 2021].

[18] J. Anvik, L. Hiew and G. Murphy, "Coping with an open bug repository", *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse '05*, 2005. Available: 10.1145/1117696.1117704 [Accessed 13 September 2021].

[19] D. Yuan, J. Zheng, S. Park, Y. Zhou and S. Savage, "Improving Software Diagnosability via Log Enhancement", *ACM Transactions on Computer*

Systems, vol. 30, no. 1, pp. 1-28, 2012. Available: 10.1145/2110356.2110360 [Accessed 13 September 2021].

[20] I. Wiese et al., "Pieces of contextual information suitable for predicting co-changes? An empirical study", *Software Quality Journal*, vol. 27, no. 4, pp. 1481-1503, 2019. Available: 10.1007/s11219-019-09456-3.

[21] T. Zimmermann, S. Kim, A. Zeller and E. Whitehead, "Mining version archives for co-changed lines", *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 2006. Available: 10.1145/1137983.1138001 [Accessed 14 September 2021].

[22] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl, "Mining version histories to guide software changes", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429-445, 2005. Available: 10.1109/tse.2005.72 [Accessed 14 September 2021].

[23] D. Spadini, M. Aniche and A. Bacchelli, "PyDriller: Python framework for mining software repositories", *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018. Available: 10.1145/3236024.3264598 [Accessed 13 September 2021].

[24] Jelber Sayyad Shirabad, T. Lethbridge and S. Matwin, "Mining the maintenance history of a legacy software system", *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*. Available: 10.1109/icsm.2003.1235410 [Accessed 14 September 2021].

[25] Bohner, "Impact analysis in the software change process: a year 2000 perspective", *Proceedings of International Conference on Software Maintenance ICSM-96*, 1996. Available: 10.1109/icsm.1996.564987 [Accessed 14 September 2021].

[26] D, Beyer. And A, Noack., 2005. *Mining co-change clusters from version repositories* (No. REP_WORK).

[27] T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen and D. Binkley, "Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis", *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016. Available: 10.1109/saner.2016.101 [Accessed 14 September 2021].

[28] C. Aggarwal, *Data mining*, 1st ed. New York: Springer.

[29] J. Han, M. Kamber and J. Pei, *Data mining*, 2nd ed. Amsterdam: Elsevier, Morgan Kaufmann, 2012.

[30] P. Hájek, I. Havel and M. Chytil, "The GUHA method of automatic hypotheses determination", *Computing*, vol. 1, no. 4, pp. 293-308, 1966. Available: 10.1007/bf02345483 [Accessed 13 September 2021].

[31] Agrawal, R. and Srikant, R., 1994, September. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (Vol. 1215, pp. 487-499).

[32] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation", *ACM SIGMOD Record*, vol. 29, no. 2, pp. 1-12, 2000. Available: 10.1145/335191.335372 [Accessed 13 September 2021].

[33] M. Zaki, S. Parthasarathy, M. Ogihara and W. Li, *Data Mining and Knowledge Discovery*, vol. 1, no. 4, pp. 343-373, 1997. Available: 10.1023/a:1009773317876 [Accessed 13 September 2021].

[34] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality", *2013 35th International Conference on Software Engineering (ICSE)*, 2013. Available: 10.1109/icse.2013.6606741 [Accessed 14 September 2021].

[35] H. Kagdi, S. Yusuf and J. Maletic, "Mining sequences of changed-files from version histories", *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 2006. Available: 10.1145/1137983.1137996 [Accessed 14 September 2021].

[36] "GitPython Documentation — GitPython 3.1.23 documentation", *Gitpython.readthedocs.io*, 2021. [Online]. Available: https://gitpython.readthedocs.io/en/stable/. [Accessed: 13- Sep- 2021].

[37] H. Gall, M. Lanza and T. Zimmermann, "4th International Workshop on Mining Software Repositories (MSR 2007)", *29th International Conference on Software Engineering (ICSE'07 Companion)*, 2007. Available: 10.1109/icsecompanion.2007.8 [Accessed 13 September 2021].

[38] C. Chee, J. Jaafar, I. Aziz, M. Hasan and W. Yeoh, "Algorithms for frequent itemset mining: a literature review", *Artificial Intelligence Review*, vol. 52, no. 4, pp. 2603-2621, 2018. Available: 10.1007/s10462-018-9629-z [Accessed 13 September 2021].

[39] H. Khanali and B. Vaziri, "A Survey on Improved Algorithms for Mining Association Rules", *International Journal of Computer Applications*, vol. 165, no. 9, pp. 6-11, 2017. Available: 10.5120/ijca2017913985.

[40] "BaseExtractor tidyextractors 0.2.1 documentation ", Tidyextractors.readthedocs.io, 2021. [Online]. Available: https://tidyextractors.readthedocs.io/en/latest/base.html. [Accessed: 13- Sep- 2021].

[41] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose", *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012. Available: 10.1109/msr.2012.6224294 [Accessed 13 September 2021].

[42] "CVSAnalY by MetricsGrimoire", *Metricsgrimoire.github.io*, 2021. [Online]. Available: https://metricsgrimoire.github.io/CVSAnalY/. [Accessed: 13- Sep- 2021].

[43] H. Wickham, "Tidy Data", *Journal of Statistical Software*, vol. 59, no. 10, 2014. Available: 10.18637/jss.v059.i10 [Accessed 13 September 2021].

[44] M. Martinez and M. Monperrus, "Coming: A Tool for Mining Change Pattern Instances from Git Commits", *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019. Available: 10.1109/icse-companion.2019.00043 [Accessed 14 September 2021].

[45] J. Heaton, "Comparing dataset characteristics that favor the Apriori, Eclat or FP-Growth frequent itemset mining algorithms", *SoutheastCon 2016*, 2016. Available: 10.1109/secon.2016.7506659 [Accessed 13 September 2021].

[46] A. Alali, B. Bartman, C. Newman and J. Maletic, "A preliminary investigation of using age and distance measures in the detection of evolutionary couplings", *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013. Available: 10.1109/msr.2013.6624024 [Accessed 14 September 2021].

[47] A. Agrawal and R. Singh, "Ruffle: Extracting co-change information from Software Project Repositories", *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2018. Available: 10.1109/icssit.2018.8748406 [Accessed 14 September 2021].

[48] A. Agrawal and R. Singh, "Identification of Co-change Patterns in Software Evolution", *2020 8th International Conference on Reliability, Infocom*

Technologies and Optimization (Trends and Future Directions) (ICRITO), 2020. Available: 10.1109/icrito48877.2020.9197979 [Accessed 14 September 2021].

[49] S. Zhang, X. Wu, C. Zhang and J. Lu, "Computing the minimum-support for mining frequent patterns", *Knowledge and Information Systems*, vol. 15, no. 2, pp. 233-257, 2007. Available: 10.1007/s10115-007-0081-7 [Accessed 14 September 2021].

[50] T. Mens and S. Demeyer, "Future trends in software evolution metrics", *Proceedings of the 4th international workshop on Principles of software evolution - IWPSE '01*, 2002. Available: 10.1145/602461.602476 [Accessed 14 September 2021].

[51] L. Moonen, S. Alesio, T. Rolfsnes and D. Binkley, "Exploring the Effects of History Length and Age on Mining Software Change Impact", *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016. Available: 10.1109/scam.2016.9 [Accessed 14 September 2021].

[52] D. Zhou et al., "Understanding Evolutionary Coupling by Fine-Grained Co-Change Relationship Analysis", *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019. Available: 10.1109/icpc.2019.00046 [Accessed 14 September 2021].

[53] "Komodo Edit - ActiveState", *ActiveState*, 2021. [Online]. Available: https://www.activestate.com/products/komodo-edit/. [Accessed: 14- Sep- 2021].

[54] M. GmbH, "MAMP & MAMP PRO - your local web development solution for PHP and WordPress development", *MAMP & MAMP PRO - Your local web development solution*, 2021. [Online]. Available: https://www.mamp.info/en/windows/. [Accessed: 14- Sep- 2021].

[55] "Download PyCharm: Python IDE for Professional Developers by JetBrains", *JetBrains*, 2021. [Online]. Available: https://www.jetbrains.com/pycharm/download/#section=windows. [Accessed: 14- Sep- 2021].

[56] "Git", *Git-scm.com*, 2021. [Online]. Available: https://git-scm.com/. [Accessed: 14- Sep- 2021].

[57] "MySQL", *Mysql.com*, 2021. [Online]. Available: https://www.mysql.com/. [Accessed: 14- Sep- 2021].

[58] C. W. Dawson, Projects in Computing and Information Systems A Student ' s Guide. Pearson Prentice Hall, 2009.

[59] *Sadc.int*, 2021. [Online]. Available: https://www.sadc.int/files/3713/7821/2867/Dissertation_PDF.pdf. [Accessed: 14- Sep- 2021].

[60] L. Blaxter, *How to research*. Maidenhead: Open University Press, 2011.

[61] A. Tolmie, D. Muijs and E. McAteer, *Quantitative methods in educational and social research using SPSS*. Maidenhead: Open University Press, 2011.

[62] "IBM Docs", *Ibm.com*, 2021. [Online]. Available: https://www.ibm.com/docs/en/db2/9.7?topic=associations-confidence-in-association-rule. [Accessed: 14- Sep- 2021].

دعم عملية صيانة البرمجيات عبر اكتشاف التغييرات المصاحبة باستخدام التنقيب في مستودعات البيانات

قدمت من قبل : علي الجيلاني خميس بن عبد الله

تحت إشراف : د. عبد السلام معتوق

الملخص

تعتبر صيانة البرمجيات العملية الأكثر تكلفة في دورة حياة تطوير نظام البرمجيات. قد تؤدي التغييرات التي يتم إجراؤها في هذه العملية على كيان برمجي معين إلى إحداث تغييرات مصاحبة في كيانات برمجية أخرى. يؤدي اكتشاف هذه التغييرات المشتركة يدويًا إلى زيادة وقت وتكلفة عملية الصيانة ، بينما قد يؤدي تجاهل تلك التغييرات المشتركة إلى حدوث عيوب في البرامج أو ضعف أداء البرنامج. قد يساعد تعدين البيانات التاريخية المخزنة في مستودعات البرامج في اكتشاف التغييرات المصاحبة للكيانات البرمجية. في هذا البحث ، نقترح نهج تغيير مسار الانتشار (CPP). نهج CPP هو نهج الكشف عن التغيير المصاحب الذي يعتمد على تعدين مستودعات البرمجيات. يتكون نهج CPP من ثلاث مراحل رئيسية. في المرحلة الأولى ، يتم جمع بيانات العمليات المخزنة في مستودعات Git. في المرحلة الثانية ، يتم إعداد البيانات المجمعة لتحليلها. يتم استخراج الميزات وإزالة العمليات المضللة وترميز أسماء الملفات. بعد ذلك ، يتم تجاهل الملفات التي تم وضع علامة محذوف عليها. أخيرًا ، يتم تقليل البيانات. ناتج هذه المرحلة هو قاعدة بيانات تحويلية تحتوي على مجموعة من قوائم أسماء الملفات المشفرة. تتضمن المرحلة النهائية أربع خطوات رئيسية. تتمثل الخطوة الأولى في إنشاء جميع الأنماط الممكنة من قوائم أسماء الملفات. الخطوة الثانية هي إنشاء قواعد من الأنماط التي تصف العلاقة بين الملفات. في الخطوة الثالثة ، يتم تجميع القواعد التي لها نفس السوابق. في الخطوة الرابعة ، يتم تقييد القواعد وفقًا لسيناريو تحرير البرنامج. تم اختبار مخرجات النهج يدويًا لتقييم المخرجات. تم بناء أداة (Assisting and Leading) بناءً على مفهوم CPP واختبارها لإثبات جدوى هذا النهج. أثبت اختبار نهج CPP أن مستودعات برامج التعدين قد تقلل من وقت عملية الصيانة بنسبة 50٪.

دعم عملية صيانة البرمجيات عبر اكتشاف التغييرات المصاحبة باستخدام التنقيب في مستودعات البيانات

قدمت من قبل :

علي الجيلاني خميس بن عبد الله

تحت إشراف :

د. عبد السلام معتوق

قدمت هذه الرسالة استكمالا لمتطلبات الحصول على درجة الماجستير في هندسة البرمجيات.

جامعة بنغازي

كلية تقنية المعلومات

قسم هندسة البرمجيات

مارس 2022