BENEFITS OF CONTINUOUS MAINTENANCE IN AGILE SOFTWARE DEVELOPMENT: A CASE STUDY

GÖRKEM HONDOROĞLU

Master's Thesis

Graduate School
Izmir University of Economics
Izmir

January 2020

BENEFITS OF CONTINUOUS MAINTENANCE IN AGILE SOFTWARE DEVELOPMENT: A CASE STUDY

GÖRKEM HONDOROĞLU

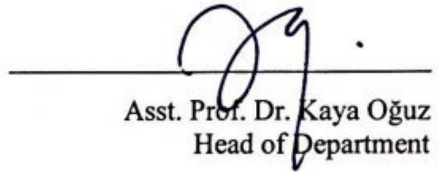
A Thesis Submitted to
The Graduate School of Izmir University of Economics
Computer Engineering Program in Graduate School

IZMIR
JANUARY 2020

Approval of the Graduate School

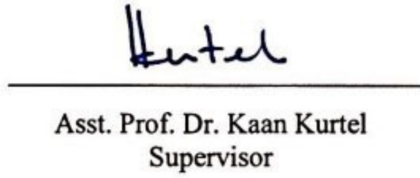
Prof. Dr. Mehmet Efe Biresselioğlu
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.



Asst. Prof. Dr. Kaya Oğuz
Head of Department

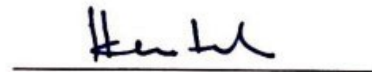
This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.



Asst. Prof. Dr. Kaan Kurtel
Supervisor

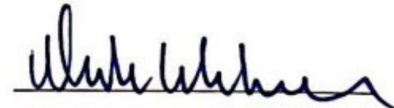
Examining Committee Members

Asst. Prof. Dr. Kaan Kurtel
Dept. of Software Engineering, IUE



Asst. Prof. Dr. Serap Şahin
Dept. of Computer Engineering, IYTE

Asst. Prof. Dr. Ufuk Çelikkan
Dept. of Software Engineering, IUE



ABSTRACT

BENEFITS OF CONTINUOUS MAINTENANCE

IN AGILE SOFTWARE DEVELOPMENT:

A CASE STUDY

GÖRKEM HONDOROĞLU

M.S. in Computer Engineering

Graduate Program

Advisor: Asst. Prof. Dr. Kaan Kurtel

January 2020

Software maintenance is an integral part of the software life cycle, and from the financial and quality perspective, it has a very high impact on a software product. Software maintenance describes the activities after delivery. That means software maintenance and development have many things in common, such as change of the design, code, and testing the existing product. These and other reasons make software maintenance a neglected area of software. Besides, Continuous Practices are getting to be an emergent area in software engineering. The academy and industry are increasingly paying attention to the practices; continuous integration, continuous delivery, and continuous deployment. Unfortunately, maintenance isn't the demanded area in academy. In this thesis, focus is the Continuous Maintenance in agile software development. In software, repositories usually refer to the main point to store data

about a system or a code. Weak repository management may adversely affect the success of maintenance. A badly managed software system can lead to a number of problems, including wasted time and programmer force in development, and difficult code tracking. The continuous software engineering practices are inherently more sensitive to dealing with such problems in particular. However, usage of continuous maintenance, can overcome these problems efficiently. We'll focus on the benefits of continuous maintenance in agile development and its challenges, analyze the effects of code changings as part of the continuous maintenance process and will identify the challenges such as impact analysis, failure tracking and etc., in continuous maintenance process by conducting a case study.

Keywords: Continuous Practices; Continuous Maintenance; Repository Management, Agile Development

ÖZET

ÇEVİK YAZILIM GELİŞTİRMEDE
SÜREKLİ BAKIMIN FAYDALARI:
VAKA ÇALIŞMASI

GÖRKEM HONDOROĞLU

Bilgisayar Mühendisliği, Yüksek Lisans

Lisansüstü Programlar Enstitüsü

Tez Danışmanı: Dr. Öğr. Üyesi Kaan Kurtel

Ocak 2020

Sürekli yazılım pratikleri, gelişmekte olan ve önemli bir yazılım mühendisliği alanıdır. Akademik dünya ve yazılım endüstrisi de bu sürekli yazılım pratiklerine giderek artan bir ilgi göstermektedir. Sürekli entegrasyon, sürekli dağıtım ve sürekli teslim bu ilgiyi görürken sürekli bakım gözden kaçmaktadır. Bu tezimde vurgulamak istediğim nokta; yazılım endüstrisin büyük ölçüde sürekli bakımın farkında olmamasıdır. Ancak, sürekli bakım; yazılım hayat döngüsünün ayrılmaz bir parçasıdır. Finans ve kalite bakış açısından bakarsak; yazılım ürününün üzerinde büyük etkisi vardır. Bu tezimde, Sürekli Bakım konusunu çevik yazılım geliştirme içerisinde ele alacağım. Yazılım mühendisliği açısında depo, genellikle verilerin saklandığı esas yerdir. Yazılım depoları geliştirme ve bakım aşamalarında, veri saklanmasına yardımcı olduğu gibi, sürüm kontrolleri ve çok kişilik ekiplerin kullanımına da yardımcı olur. Kötü yönetilen bir yazılım sistemi birçok soruna sebebiyet verebilir. Bunların başında; vakit ve iş gücü kaybı, kaynak kodu takibinin zorlaşması, deponun şişmesi ve sürüm kontrolünün zorlaşması gelebilir. Bunlara ek olarak, zayıf bir depo yönetimi bakım ve yazılım evriminin üzerinde olumsuz etki oluşturur. Sürekli yazılım mühendisliği pratikleri

doğası gereği bu tarz problemlerle ilgilenmek konusunda hassastır. Ancak, sürekli yazılım mühendisliği özellikle de depo arşivleme ve yönetimi bu problemlerin üstesinden gelmek konusunda verimli bir çözüm sunabilir. Bu tezde, sürekli bakım sürecinin, çevik yazılım geliştirme içerisinde kullanımıyla birlikte kazanılacak kazançlar incelenip, sürekli bakım çalışmalarında olan sorunların bir parçası olan kod değişikliklerinin etkisini analiz etmeye çalışıp karşılaşılan etki analizi, hata takibi gibi zorlukları bir vaka çalışmasıyla belirlenecektir.

Anahtar Kelimeler: Sürekli yazılım pratikleri; sürekli bakım; yazılım depo yönetim, Çevik Yazılım

ACKNOWLEDGEMENT

I wish to acknowledge the help provided by Asst. Prof. Dr. Kaan Kurtel. He provided me with valuable advice, and support while writing this thesis. He guided me about how to approach problems and finding ways to resolve them. I am thankful to Asst. Prof. Dr. Ufuk Çelikkan and Asst. Prof. Dr. Serap Şahin for their guidance's.

I am thankful to Mustafa Tufan for providing me the case study project. Hakan İnanç and Semih Ünaldı for their help and participation in the case study. Suzan Özşimşek and Burcu Kocakurt for helping with the language.

In addition, I am thankful to my family; my mother Nihal, my father Adnan and my brother Arda who always supported me. Their love and help encouraged me to do new things in my life. Their presence, whenever I need, always made me feel strong against any challenge. They helped me to become the person who I am. Thus, I am always grateful to them. Finally, to my dear girlfriend Suzan for her inspiration.

TABLE OF CONTENTS

ABSTRACT iii
ÖZET v
ACKNOWLEDGEMENT vii
TABLE OF CONTENTS viii
LIST OF TABLES xi
LIST OF FIGURES xii
LIST OF ABBREVIATIONS xiv
CHAPTER 1: THESIS STATEMENT 1
1.1 Purpose of the Study 1
1.2 Significance of the Study 1
1.3 Organization of the Study 1
CHAPTER 2: FUNDAMENTALS 3
2.1 Software Life Cycle 3
2.2 Software Maintenance 4
2.3 Continuous Practices 7
2.3.1 Continuous Integration 9
2.3.2 Continuous Delivery 9
2.3.3 Continuous Deployment 10
2.3.4 Continuous Release 10
2.4 Global Software Development for Small Teams 11
CHAPTER 3: MOTIVATION AND LITERATURE RESEARCH 12
3.1 Motivation 12

3.2 Literature Research 13
3.2.1 Research questions 13
3.2.2 Data sources 13
3.2.3 Data retrieval 14
3.2.4 Data analysis 19
3.2.5 Results of literature research 21
CHAPTER 4: CONTINUOUS MAINTENANCE 23
4.1 Pre-Production Continuous Maintenance 24
4.2 Post-Production Continuous Maintenance 25
CHAPTER 5: REPOSITORY MANAGEMENT 27
5.1 Repository 27
5.1.1 Version Control 29
5.1.2 Centralized Version Control System 31
5.1.3 Distributed Version Control System 32
5.1.4 Advantages and disadvantages of version control system types 34
5.1.5 Benefits of version control systems 34
5.2 Repository Management 36
CHAPTER 6: THE CASE STUDY 38
6.1 The Company and Project Background 38
6.2 Development Phase 40
6.2.1 Project instructions 40
6.2.2 Project structure 41
6.2.3 Installations 42
6.2.4 Analysis and design 43
6.2.5 Code structure 47
6.3 Observations of the Case Study 53
6.4 Findings of the Case Study 59

CHAPTER 7: CONCLUSIONS AND FUTURE WORKS 61

REFERENCES 65

LIST OF TABLES

Table 1: Search strings for selected data sources 15
Table 2: Selected number of papers obtained by searching the data repositories...... 15
Table 3: Publication type for 29 selected studies 16
Table 4: Selected studies sorting by their subject from SDLC to repository............. 17
Table 5: Papers and their touch points 18
Table 6: Software language and repository relationship 28

LIST OF FIGURES

Figure 1: The Organization of the Thesis ... 2
Figure 2: Software Life Cycle ... 4
Figure 3: Maintenance Categories Identify by IEEE 14764 ... 6
Figure 4: Components of the Continuous Practices and Its Relations ... 9
Figure 5: Difference Between CD and CDE ... 10
Figure 6: Categorizations of the Selected Papers by Their Subjects ... 16
Figure 7: Remote and Local Repository Relationship ... 28
Figure 8: Version Control Workflow of Git ... 30
Figure 9: Centralized Version Control ... 32
Figure 10: Distributed Version Control ... 33
Figure 11: Popular Version Control Systems ... 35
Figure 12: Repository Management Visualization ... 36
Figure 13: Aliv.ee Back-end Structure ... 41
Figure 14: Result Screen ... 42
Figure 15: Work Breakdown Structure ... 44
Figure 16: Use Case Diagram of Manual Link Creation Scenario ... 44
Figure 17: Use Case Diagram of Random Link Creation ... 45
Figure 18: Class Diagram of the Project ... 45
Figure 19: Modelling of Executables ... 46
Figure 20: Sequence Diagram of Manual Link Creation ... 46
Figure 21: Sequence Diagram of Random Link Creation ... 47
Figure 22: Dependencies of the Project Alivee ... 48
Figure 23: Entity Relationship Diagram ... 49
Figure 24: Repository Annotation JPA ... 49
Figure 25: Controller Class ... 50
Figure 26: Link Service Interface ... 51
Figure 27: Link Entity ... 51
Figure 28: Link Service Implementation ... 52
Figure 29: Continuous Maintenance Interaction ... 53
Figure 30: Configuration Error (Company info is hided) ... 54

Figure 31: Connection Error 56
Figure 32: Bitbucket Commit History... 57
Figure 33: Maintenance-Testing Relation... 58
Figure 34: Automated Unit Test System.. 59
Figure 35: Continuous Maintenance- Continuous Practices and SDLC Relation 60

LIST OF ABBREVIATIONS

CD	Continuous Delivery
CDE	Continuous Deployment
CI	Continuous Integration
CM	Continuous Maintenance
CR	Continuous Release
CP	Continuous Practices
DevOps	Development & Operations
IT	Information Technology
RM	Repository Management
SDLC	Software Development Life Cycle
SLC	Software Life Cycle
VCS	Version Control System
GSD	Global Software Development

CHAPTER 1: THESIS STATEMENT

1.1 Purpose of the Study

In this study, we focused on software continuous maintenance in agile development, and its challenges. We tried to increase the existing knowledge in Continuous Maintenance especially the effects of code and environment changings as part of the maintenance process in small development teams by designing and developing a custom case study.

1.2 Significance of the Study

Software maintenance and continuous delivery studies are complemented by software engineering concepts, both of which aim to encourage the efforts for both agile development and software delivery improvement. There has been great interest in continuous delivery in the last two decades; however, dealing with continuous maintenance is also of the same importance.

The focus of this thesis is to increase the knowledge available for continuous maintenance and repository management by conducting a case study to analyze the impact of code and environmental changes. Furthermore, this thesis aims to contribute to the continuous practices of small development teams especially in continuous maintenance, which requires new developments and improvements.

1.3 Organization of the Study

This thesis is organized as two main parts: “Fundamentals” and “Research” that are presented in Figure 1.

In Section 1, we explained the objective and significance of the thesis. Software life cycle, global software development for small teams, software maintenance and continuous practices have been explained respectively in Section 2.

In the second main part entitled "Research", the activities concerning the research process have been explained respectively. Section 3 begins with the clarification of the research procedures, it is followed by interview and literature research, and the obtained results presented at the end of this section. Sections 4 and 5, explained and discussed the Continuous Maintenance and Repository. Preparation of the Case Study, its structure and the outputs are presented in Section 6.

Finally, in Section 7, conclusion and future works will be gathered from the study.

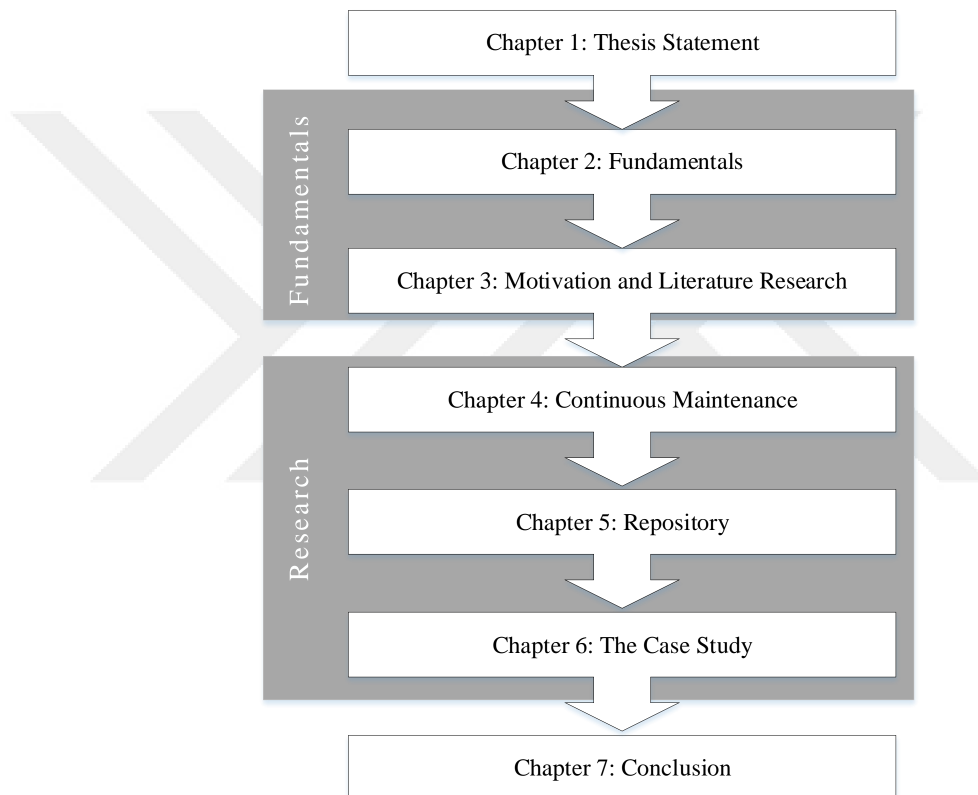


Figure 1: The Organization of the Thesis

CHAPTER 2: FUNDAMENTALS

2.1 Software Life Cycle

John W. Tukey is the first person to use the term “software” in (Tukey, 1958);

> “*Today, the ‘software’ comprising the carefully planned interpretive routines, compilers, and other aspects of automotive programing are “at least as important” to the modern electronic calculator as its ‘hardware’ of tubes, transistors, wires, tapes and the like.*”

Software and software engineering is defined by IEEE’s Software Engineering Body of Knowledge (SWEBOK) (Bourque, and Fairley, 2014).

> “*Computer programs and their associated documentation is called software.*”
>
> *“Software Engineering (SE) is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.”*

Software development follows a process called Software Development Life Cycle (SDLC). SDLC processes consist of a set of finite activities and contains a complete plan for developing. SDLC has four phases and those are; requirement analysis, design, implementation, and testing. Requirement analysis’s aim is to grab out all the details of the project from the client. In Design phase; developers and technical architects group up and determine the high-level design of the project. High-level design determined by the collected requirements and will be the core part of the implementation phase. After requirement collection and design, the third stage is

implementation. In this stage, developer starts coding in order to fulfill client's requirements and the design objectives. All the coding activities are organized in this phase. The final phase is testing. Before delivery, testers check the software if it is working as per user's expectations or not. Maintenance is the following phase after the SDLC process is completed. This means that; maintenance phase begins, after the software is deployed. When Software development life cycle phases combine with maintenance named Software Life Cycle (SLC). The relation between SDLC and SLC is shown in Figure 2.

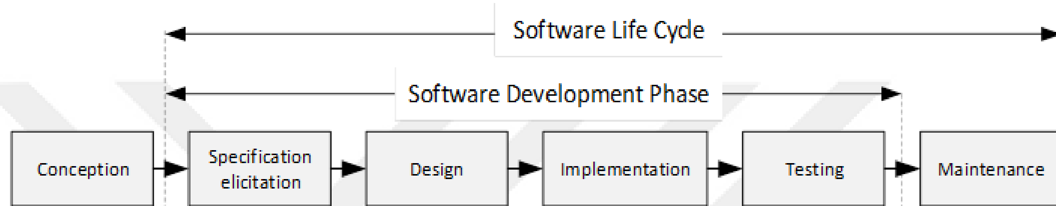


Figure 2: Software Life Cycle

2.2 *Software Maintenance*

Software maintenance is one of the fundamental topics in software engineering, and is an inseparable part of software life cycle (IEEE-12207, 2008). All the financial sources and expenses spent on software maintenance account for 40% to 70% of all the fiscal sources and expenses allocated to software life cycle (Grubb, and Takang, 2003). According to Boehm (Boehm, 1987), 50% to 90% of software life cycle expenditures pertain to software maintenance. In other words, the money spent on maintenance doubles the one spent on development. Jones (Jones, 2008) estimates that the total number of software engineers to work for software maintenance towards the end of 21st century will reach approximately 4,000,000. He also adds that this number constitutes 12% to 15% of the total workforce working in software industry.

Software maintenance is defined as providing technical support in a cost-effective way. The need for software maintenance will not end as long as a specific software program is used. Grubb and Takang (Grubb, and Takang, 2003) explains the reasons for software maintenance as follows.

- **To provide service availability:** It includes all the efforts and activities required to keep the system running and accessible.
- **To provide compulsory updating support:** It includes meeting changing needs of customers about the software programs they have already purchased on account of changing laws and technology.
- **To keep up with the changing needs of customers:** Users might want to use these programs for other functions as well. So, their needs and expectations might alter in time.
- **The possibility for maintenance need in the future:** The code or database might have to be restructured, and documentation to be updated because of commercial and financial reasons.

Software maintenance is one of the key phases/principles of SLC. Although maintenance and evolution specify different things, they can be classified under the same category. The term "evolution" was first introduced in the software domain by Mark Halpern (Halpern, and Shaw, 1966) in 1965 as the growth characteristics of software. Belady and Lehman (Belady, and Lehman, 1972) contribute the concept and publicize a set of principles to determine the evolution of software systems. In 1976, "Maintenance" term consists of "corrective", "adaptive", "perfective" maintenance activities introduced by Swanson (Swanson, 1976). Software maintenance is commonly defined to avoid software failure. On the other hand, software evolution means, enhancing the software. Formally, *Software Maintenance* defined by IEEE Std. 610 (IEEE-Std610.12, 1990) Glossary of Software Engineering Terminology Dictionary;

> *"The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment."*

Development process ends with the delivery. However, that is not the end of the project. Platform updates, environments changes or user requirements change the project. Software has to evolve or change to meet the new changes.

Maintenance refers to altering the system without changing its functionality. This definition refers to adding new functions, updating the current system or, adding logs or performance improvements but the working system is not affected by these changes. To do that IEEE identifies activities for maintenance in IEEE Std. 14764 (IEEE-Std.14764, 2006). Those activities are processed implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, and retirement.

Maintenance activities are fulfilled by maintainer or in small developer teams. There are five characteristics that maintainer has to follow and those are as follows:1) Maintaining control over the software's day-to-day functions, 2) maintaining control over software modifications, 3) improving existing functions, 4) identifying security threats, and fixing security vulnerabilities and 5) preventing software performance from degrading to unacceptable levels. Types of software maintenance presented (Figure 3) and described as follows (IEEE-Std.14764, 2006), (IEEE-Std.1219, 1998):

The types of software maintenance are shown in Figure 3.

	Correction	**Enhancement**
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

Figure 3: Maintenance Categories Identify by IEEE 14764

1. **Corrective maintenance:** In this type of maintenance, the software company corrects the program failures notified by customers after having bought the product. The maintainers must clear the fault to keep software product operational.
2. **Adaptive maintenance:** If the working environment, operating system, or hardware components change, then the old software program should be adapted correspondingly.
3. **Perfective maintenance:** This kind of maintenance is done to increase the performance of the program, or add some more functions.

4. **Preventive maintenance:** This sort of maintenance is conducted after delivery of the software product as a precaution to thwart possible defects. This keeps the performance and reliability of the system up to a certain extent.

2.3 Continuous Practices

Today's technology and business trends and global competition force software product to rapid change. Many of the foremost academician and industry authorities also pointed which that change is an inevitable factor/attribute in the software domain (Sommerville) (Sommerville, 2015). This critical attribute pushes a software product toward to continuously change. It means that when a software product's functionality, constraints, architecture, and operational environment change, then making the necessary changes in the software is inevitable.

Given all the above facts, the existing development approaches had to be changed and this change has resulted to propose new methodologies and approaches. Agile software development and its methodologies have emerged from one of those methodologies and brought a new breath to software development. Since high-functioning software project developments are frequent, Continuous Practices enable the usage of Agile Methodologies. Continuous Practices (CP) refer to organize the software development processes or phases such as requirement analysis, design, implementation and testing continuously. It helps to deployment and also provides quick feedback from software and customer in a very rapid cycle. It also monitors the software life cycle processes in order to see what is going on or what is wrong with the system.

With the spread of Continuous Practices; DevOps is getting popular. DevOps is an agile relationship between development and IT operations and the word DevOps is derived from the words development and operations. Continuous practices and DevOps are support each other and work better together. Yet, these concepts are different. DevOps emphasize the responsiveness and focus on the cultural structure. On the other hand, CP emphasizes automation and focus on the software life cycle. A way to identify a development life cycle in which operations and development worked together throughout all stages from design to development, from testing to release.

Stahl and Bosch (Stahl, and Bosh, 2017) try to differentiate the terms of Continuous Practices and DevOps in their study. The study found that; mostly DevOps believed to be an enabler of continuous deployment. Since it encourages the rapid, continuous and small deployments. Rather than finding the differences, they found out that continuous practices are “DevOps practices”. This study implied that DevOps and continuous practices work better together, and they are not two different concepts but concepts that are interwoven.

Most of the technological companies in industry, implements the continuous practices in papers mostly refers to continuous integration. The concept of continuous integration was firstly introduced in the 1995, book of Microsoft Secrets (Cusumano, and Selby, 1995), since that continuous practices are intermixed with each other. The book describe it as; *“do everything in parallel, with frequent synchronizations.”*

Continuous practices are commonly categorized as below:

- Continuous Integration (CI),
- Continuous Delivery (CD),
- Continuous Deployment (CDE),
- Continuous Release (CR).

From then, continuous practices refer to continuous integration. In this definition and usage, continuous integration is the combination of continuous integration, continuous delivery, continuous deployment and continuous release. Even though continuous maintenance is the extension of this defined continuous integration; it is rarely discussed in publications or educational papers. Pang (Pang, and Hindle))told in her conference paper Continuous Practices. Figure 4 (Shahin, Ali-Babar, and Zhu, 2017) shows the relations between continuous practices.

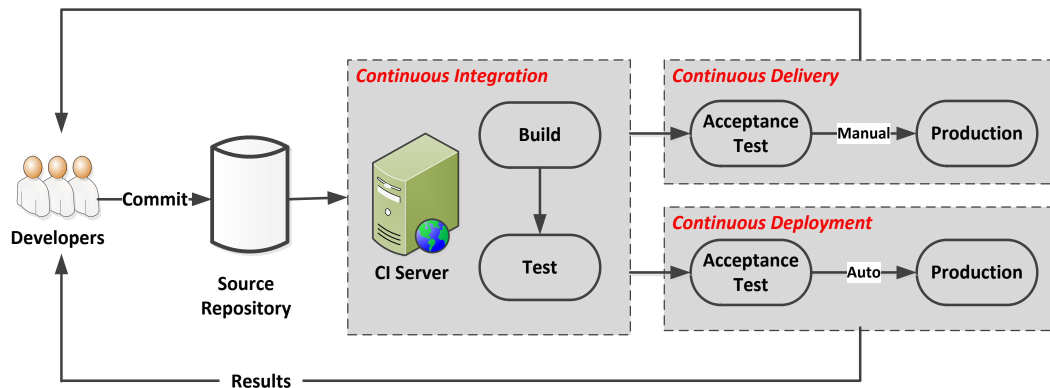


Figure 4: Components of the Continuous Practices and Its Relations

2.3.1 Continuous Integration

Continuous integration is a developer practice. Developers integrate their work frequently. Each person usually integrates at least daily, which means multiple integrations per day.

Since it's a developer practice and not a development practice, it distinguishes it from continuous delivery and deployment. That means, you may be using continuous delivery but that does not guarantee that developers actually follow continuous integration.

Continuous integration practices match with the agile methodologies. Agile methodologies suggest short release cycles between two to four weeks. Continuous integration is in alignment with agile in that. In addition to that, Continuous integration follows daily integration. It helps with shorter and frequent release cycle. Software quality and team's productivity improved with shorter period releases and multiple integrations.

2.3.2 Continuous Delivery

Continuous Delivery is a software development discipline unlike the CI, which is a developer discipline. In CD, you build a software in such a way that the software can be released to production at any time.

Performing a complete test and verification process is not possible for large-scale developments. Instead of this, code is in a state that can always be deployable. Which

means, every commit goes through the pipeline then sampled. Those commits become release candidates.

2.3.3 Continuous Deployment

As in the case of continuous delivery, continuous deployment is also developer discipline. Unlike CD, in continuous deployment, release candidates frequently and rapidly placed in a production environment. The nature of which may differ depending on technological context.

It is important to note that CD practice implies CDE practice, but the converse is not true. A continuous delivery pipeline automatically tests the applications but keeps the deployment decision as a manual step. A continuous deployment pipeline, on the other hand, will automatically deploy this working version. Differences between continuous delivery and continuous deployments are shown in Figure 5.

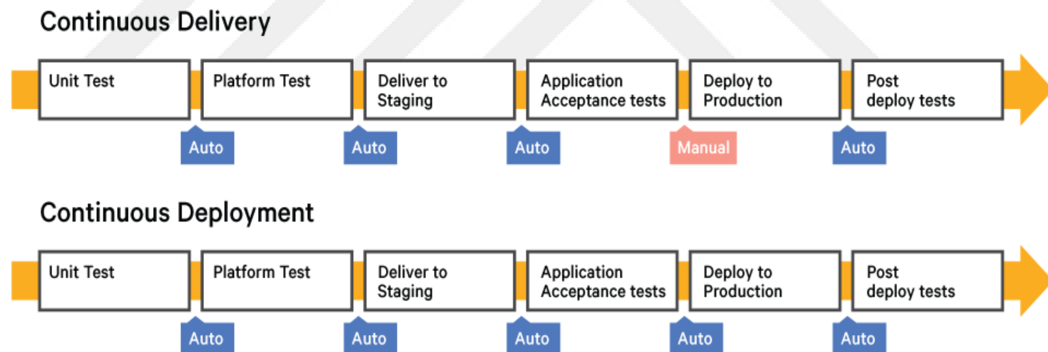


Figure 5: Difference Between CD and CDE

2.3.4 Continuous Release

Continuous release is a business practice. In continuous delivery, release candidates are deployed manually. Difference in continuous release is those deployed candidates are frequently made utilizable to users.

For better understanding, in the case of user-installed software, continuous deployment is not an applicable concept even though continuous release may very well be.

2.4 Global Software Development for Small Teams

Global Software Development (GSD) is described by Aranda (Aranda, 2008) as follows: when the distribution of the members of a distributed software development team exceeds the frontiers of a country. It is increasing practice among all the large companies with multiple teams and multiple locations. GSD's benefits are; 1) reduced development costs due to the salary savings, 2) reduced development duration due to great time zone effectiveness. Global software development's organization works with multiple locations. Teams divided into small groups to carry on the tasks.

In maintenance, software is going to change day by day. These changes could be code changes, environment changes or even developers can change as well. Maintenance is a difficult process because of this reason, and it is even more difficult in global software teams. In global software teams, communication is a problem. Developers are in different locations and even different time zones. And yet, industry giants usually work in different locations.

Wiredu (Wiredu, 2020) explain how global software engineering works. In order to be efficient in global software engineering coordination theory has to be followed. Coordination theory is a set of principles about how the software team work together and be coordinated. The book categorized coordination theory into four dominant perspectives which are technology, information, geography and organization. Those four different areas has to be coordinated harmonically.

CHAPTER 3: MOTIVATION AND LITERATURE RESEARCH

This section aimed to find the existed researches for the thesis' subject.

3.1 Motivation

Motivation of the study is based on my own experiences in business. I am currently working in a company which uses Agile development as a main methodology and use continuous development in each Agile sprint. Since a lot of people use the same repositories, management of those repos are really difficult and that cause some problems as well. In a discussion with my coworkers and managers we come to conclusion that repository management is a problem and also another conclusion is that, continuous maintenance is started to being used in business but in academy there is not enough papers on it. Hence, with that motivation my literature review shaped.

This research begins with the determination of the objective and review of the previous literature regarding the continuous practices and continuous maintenance. For these two studies, we organized and performed informal interviews for a deeper understanding of the existing issues of continuous maintenance and repository management. These interviews made with people from the software industry confirmed, that Continuous Maintenance is used for software development processes, especially in the software development life cycle. Even though the industry integrates the concept and the usage of continuous maintenance; in the academy not enough study in consideration of the significance of the CM. After the interviews, we carried out a detailed review of the existing literature. The primary focus of this review was to determine how continuous maintenance and repository management is addressed in the literature.

3.2 Literature Research

According to the motivation, we wanted to examine the position of Continuous Maintenance in the literature. To achieve this goal, we designed a systematic literature research by using the following methodology:

1. Research questions
2. Define the data sources
3. Data retrieval
4. Data analysis
5. Results

3.2.1 Research questions

To understand the position of the Continuous Maintenance in the literature, we prepared three research questions that are presented below:

RQ1: What is reported in the research literature about Continuous Maintenance in software engineering?

RQ2: What is reported in the research literature about the repository management that related Continuous Management?

RQ3: What are the challenges in continuous maintenance?

3.2.2 Data sources

The following data sources were scanned to retrieve any publications from conference papers, workshops, journal articles, books and theses.

- IEEE Xplore Digital Library (www.ieeexplore.ieee.org)
- Springer (www.link.springer.com)
- Google Academic (scholar.google.com)
- Elsevier (www.elsevier.com)
- ResearchGate (www.researchgate.net)

In some cases, papers were available both on IEEE and scholar.google so in that case one duplicate were removed manually.

3.2.3 Data retrieval

We cover five different data sources basically that means; millions of publications. In order to narrow it to our focus we need some search criteria. Table 1 shows the strategy behind the search criteria. The Boolean operator “AND” and “OR” used to form a string. Continuous Maintenance is relatively a new research area. Most of the studies started after 2000s but mostly after 2010s. Within the 29 papers, that we scanned only one of them was published before 1993 and that is the Tarek and Abdel-Hamid’s (Abdel-Hamid, 1993) paper called Adapting, Correcting, and Perfecting Software Estimates A maintenance Metaphor. It is not about continuous maintenance but its main idea is about continuous estimation. All of the other studies are conducted after the 2000s. Since, the research start date was established to be 2000 and the end date was set to 2018.

The papers that selected for this study had a significant contribution in the field in terms of new areas to research. The 29 studies were analyzed separately by each author. The author selected as the first name in the paper. Publication type was also analyzed for all selected papers. The types are conference, journal, workshop, thesis and book. Also, the publication year is analyzed and from 29 studies, only one (Abdel-Hamid, 1993) is before 2000s. Final analyze category is the publisher. Publishers categorized as IEEE, Springer, Elsevier, ResearchGate and Others. In Table 2, total selected studies listed according to their data source.

Table 1: Search strings for selected data sources

Data Source	Search String
IEEE	((Repository OR Repository Management OR Repository Management Activities) AND Software) OR ((Continuous Integration OR Continuous Delivery OR Continuous Deployment OR Continuous Release OR Continuous Practices) AND Software) OR (Continuous Maintenance OR (Continuous Maintenance AND Software) OR (Continuous Maintenance AND Agile)) OR Global Software Development OR Agile)
Springer	(Continuous Maintenance OR Continuous Software OR Maintenance)
Scholar Google	((Repository OR Repository Management OR Repository Management Activities) AND Software) OR ((Continuous Integration OR Continuous Delivery OR Continuous Deployment OR Continuous Release OR Continuous Practices) AND Software) OR (Continuous Maintenance OR (Continuous Maintenance AND Software) OR (Continuous Maintenance AND Agile)) OR Global Software Development OR Agile)
Elsevier	(Continuous maintenance OR Maintenance OR Software Continuous maintenance)
ResearchGate	(Continuous Maintenance OR (Continuous Maintenance AND Software) OR (Continuous Maintenance AND Agile) OR (Repository Management OR (Repository AND Software))

Table 2: Selected number of papers obtained by searching the data repositories

Data Source	Total Selected	Relevant	Irrelevant
IEEE	15	11	4
Springer	1	1	0
Scholar Google	7	7	0
Elsevier	1	1	0
ResearchGate	5	4	1
Total	29	24	5

Software development life cycle, agile, global software development, maintenance, continuous practices, continuous maintenance and repository are the areas that are scanned. After scanning period, 29 studies were filtered with search criteria and selected for this paper. The categorization of the selected papers are shown in the Figure 6.

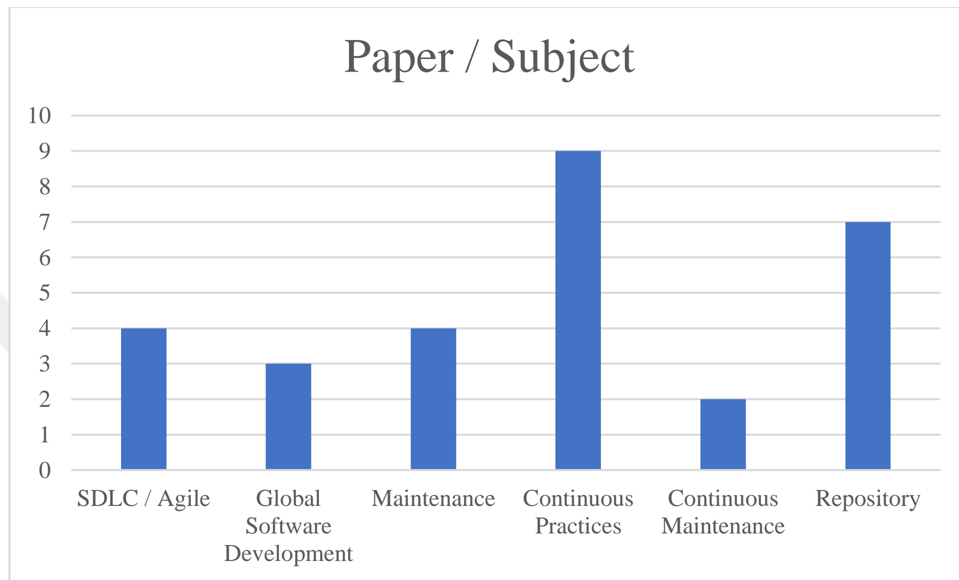


Figure 6: Categorizations of the Selected Papers by Their Subjects

Table 3 list the publication types for the selected studies and Table 4 shows the details of the selected studies. Also, Table 5 shows the 29 selected papers touch points.

Table 3: Publication type for 29 selected studies

Publication Type	**Number**
Journal	**15**
Workshop	**3**
Conference	**9**
Thesis	**1**
Book	**1**

Table 4: Selected studies sorting by their subject from SDLC to repository

Study	First Author	Year	Publisher	Type
A detailed study of Software Development Life Cycle (SDLC) Models (Usha-Rani, 2017)	S. Barjtya	2017	Others	J
Agile processes for the maintenance cycle (Cardoso de Mello, 2012)	M. Cardoso	2012	Others (IBM)	J
Enabling Agile Testing Through Continuous Integration (Stolberg, 2009)	S. Stolberg	2009	IEEE	C
Global software development: Where are the benefits? (O Conchuir et al., 2009)	E. Ó Conchúir	2009	ResearchGate	J
Global Software Development: Who Does It? (Begel, and Nagappan, 2008)	A.Begel	2008	IEEE	C
Impact of Agile Methodology on Software Development Process (Kumar, and Bhatia, 2012)	G. Kumar	2012	ResearchGate	J
Managing Software Risks in Maintenance Projects, from a Vendor Perspective: A Case Study in Global Software Development (Sundararajan, Bhasi, and Pramod, 2017)	S. Sundararajan	2017	ResearchGate	J
Challenges in Software Evolution (Mens et al., 2005)	T. Mens	2005	IEEE	W
Design for maintenance: An interview-based survey (Buinis, 2015)	M. Buinus	2015	Others	T
Maintenance and Agile Development: Challenges, Opportunities and Future Directions (Hanssen et al., 2009)	G.K. Hanssen	2009	IEEE	J
Continuous Software Engineering (Bosch, 2014)	J. Bosch	2014	Springer	B
Continuous Integration (Fowler, 2006)	M .Fowler	2006	Others	J
Continuous Software Quality Control in Practice (Steidl et al., 2014)	D. Steidl	2014	IEEE	C
An empirical study on principles and practices of continuous delivery and deployment (Schermann et al., 2016)	G. Schermann	2016	Others	J
A Software Repository for Education and Research in Information Visualization (Borner, and Zhou, 2001)	K. Börner	2001	IEEE	C
Mining Software Repositories- A Comparative Analysis (Olatunji Sunday et al., 2010)	S.O. Olatunji	2010	ResearchGate	J
Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices (Shahin, Ali-Babar, and Zhu, 2017)	M. Shahin	2017	IEEE	J
Continuous Delivery Practices in a Large Financial Organization (Vassallo et al., 2016)	C. Vassallo	2016	IEEE	J
Continuous Integration and Quality Assurance: A case Study of Two Open Source Projects (Holck, and Jørgensen, 2003)	J. Holck	2004	Others	J
Continuous Practices and DevOps: Beyond the Buzz, What Does It All Mean? (Stahl, and, Bosch, 2017)	D. Stahl	2017	ResearchGate	C
Implantation of continuous-Integration practices: An experience report in a software development and research laboratory (Pereira, Amorim, and Nunes, 2018)	I.M. Pereira	2016	Others	J
Continuous maintenance And the future– Foundations and technological challenges (Roy et al., 2016)	R. Roy	2016	Elsevier	J
Continuous Maintenance (Pang, and Hindle, 2016)	C. Pang	2016	IEEE	C
Adapting, Correcting and Perfecting Software Estimates: A Maintenance Metaphor (Abdel-Hamid, 1993)	T.K. Abdel-Hamid	1993	IEEE	J
Analysis of the ISBSG Software Repository from the ISO 9126 View of Software Product Quality (Cheikhi, Abran, and Desharnais, 2012)	L. Cheikhi	2012	IEEE	C

Table 4: (cont’d)

The Maven Repository Dataset of Metrics, Changes, and Dependencies (Raemaekers, Deursen, and Visser, 2013)	S. Raemaekers	2013	IEEE	C
Software Repository Analysis for Investigating Design-Code Compliance (Ozbas-Caglayan, Dogru, 2013)	K.Ozbas Caglayan	2013	IEEE	W
Tool Demo: Browsing Software Repositories (Reiss, 2014)	S.P. Reiss	2014	IEEE	C
University-Industry Collaboration and Open Source Software (OSS) Dataset in Mining Software Repositories (MSR) Research (Tripathi, Dabral, and Sureka, 2015)	A.Tripathi	2015	IEEE	W

Type: **C-** Conference, **W-** Workshop, **B-** Book, **J-** Journal, **T-** Thesis

Table 5: Papers and their touch points

Papers	**SDLC**	**Agile**	**GSD**	**CI**	**CD**	**CDE**	**CR**	**M**	**CM**	**R**
A detailed study of Software Development Life Cycle (SDLC) Models	X	X								
Agile processes for the maintenance cycle	X	X						X		
Enabling Agile Testing Through Continuous Integration	X	X		X						
Global software development: Where are the benefits?	X	X	X							
Global Software Development: Who Does It?	X	X	X							
Impact of Agile Methodology on Software Development Process	X	X		X						
Managing Software Risks in Maintenance Projects, from a Vendor Perspective: A Case Study in Global Software Development	X	X	X					X		
Continuous maintenance and the future–Foundations and technological challenges.								X	X	
Continuous Maintenance				X	X	X	X	X	X	X
Adapting, Correcting and Perfecting Software Estimates: A Maintenance Metaphor								X		
Continuous Integration		X		X						
An empirical study on principles and practices of continuous delivery and deployment					X	X				
Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices		X		X	X	X	X			
Continuous Delivery Practices in a Large Financial Organization				X	X					
Continuous Integration and Quality Assurance: A case Study of Two OpenSource Projects				X				X		
Continuous Practices and DevOps: Beyond the Buzz, What Does It All Mean?		X		X	X	X	X			
Implantation of continuous integration practices: An experience report in a software development and research laboratory		X		X						
Challenges in Software Evolution	X	X						X		
Design for maintenance: An interview-based survey								X		
Maintenance and Agile Development: Challenges, Opportunities and Future Directions	X	X						X		
A Software Repository for Education and Research in Information Visualization										X
Mining Software Repositories – A Comparative Analysis										X

Table 5: (cont'd)

Analysis of the ISBSG Software Repository from the ISO 9126 View of Software Product Quality										X
The Maven Repository Dataset of Metrics, Changes, and Dependencies								X		X
Software Repository Analysis for Investigating Design Code Compliance										X
Tool Demo: Browsing Software Repositories										X
University-Industry Collaboration and Open Source Software (OSS) Dataset in Mining Software Repositories (MSR) Research										X
Continuous Software Engineering				X	X	X		X		
Continuous Software Quality Control in Practice						X				

***SDLC**: Software Development Life Cycle,* ***GSD:** Global Software Development,*
***CI:** Continuous Integration,* ***CD:** Continuous Delivery,* ***CDE:** Continuous Deployment,*
***CR**: Continuous Release,* ***M:** Maintenance,* ***CM**: Continuous Maintenance,* ***R:** Repository.*

After the retrieval of the papers, we categorized to 29 papers based on their importance. Importance of the papers decided by the research questions and their relevance. Seven papers selected and these papers are; 1) (Pang, and Hindle, 2016) 2) (Roy et al., 2016), 3) (Stahl, and Bosch, 2017), 4) (Shahin, Ali-Babar, and Zhu, 2017), 5) (Raemaekers, Deursen, and Visser, 2013), 6) (Mens et al., 2005), and 7) (Bosch, 2014).

3.2.4 Data analysis

In this section, we summarized the selected seven of the twenty nine papers/books that identified the most crucial studies for our research.

Pang and Hindle's conference paper is one of the limited number of articles in the continuous maintenance area. They first introduced the idea that continuous maintenance (CM) is not getting enough interest as the other continuous practices such as continuous integration, continuous delivery, continuous deployment and continuous release. They give a quick information about the continuous practices and describe briefly that what continuous maintenance is. They categorize the CM and identify the possible research areas on continuous maintenance.

Roy et al.'s Continuous maintenance and the future-foundations and technological challenges focus on the business level of the continuous maintenance rather than the software maintenance. They analyze high-value products such as high-tech machine tools, aircraft engine, nuclear power station, train, defense equipment, high-end car,

medical equipment and wind turbine. Level of continuous maintenance is growing with respect to the products lifetime, deadliness and project's scope. With the new technologies and high usage rate of Additive Layer Manufacturing, Industry 4.0 and Internet of Things, better maintenance and overall health of the projects is available. They identify that CM process is changing due to the business model evolution. Industry 4.0 and Internet of Things are major factor and helper in the process of continuous maintenance.

Stahl et al.'s paper titled Continuous Practices and DevOps: Beyond the Buzz, What Does It All Mean. Their focus is mainly uncluttering the term understandings on DevOps and continuous practices. They follow a mapping study followed by in-depth review of relevant papers. They explain the both concepts and give proper definitions to them. Also, the differences between DevOps and Continuous Practices identified. Addition to these definitions, they describe and discuss the terms such as Continuous integration, continuous delivery, continuous deployment and continuous release. This study's importance for our research is; continuous maintenance is the extension of continuous integration so that the best definitions for these terms are identified by Stahl et al. Today, when we talk about these terms, we get a reference from this paper.

Shahin et al. research titled Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices focus on the subject continuous practices. The relationships between continuous integration, delivery and deployment is specified. Their aim is the fill the knowledge gap about these terms and their relationships. They identified the tools that are using to implement continuous practices and in addition to that they identified to challenges as well. In conclusion these areas gaining interest and also, they found statistical data about the tools, challenges and practices.

Raenaejers et al.'s paper titled The Maven Repository Dataset of Metrics, Changes, and Dependencies focus on the analyzing the maven dataset. The dataset they used provided by Delft University of Technology. Maven Dependency Dataset has 148.253 jar files which contains files such as metrics, changes and dependencies. They analyze to data with respect to database. They presented the schemas for three different kind

of databases. First one is relational database then the second database is key-value database. Finally, last database is graph database.

Mens et al. paper titled Challenges in Software Evolution focus is as clear as its title “Challenges in Software Evolution”. They proposed a list which contains 18 essential challenges in the software evolution process. This proposition is a result of concentrated effort of 20 European researchers that active in software evolution.

Bosch’s Continuous Software Engineering, actually it is a reference book, which provides essential insights on the adoption of modern software engineering practices at large companies producing software-intensive systems. Book mainly represent a model called “Stairway to Heaven” and it is a unique collaboration between research and industry. Book shows how big companies deal with challenges while using continuous practices with case studies. Also, provides concrete models, frameworks.

3.2.5 Results of literature research

From selected 29 papers, we obtained that only **two research papers** have the scope of continuous maintenance. Even in those two;

- Roy et al.’s paper focus on the business level of continuous maintenance rather than software projects; It’s not even %10 of all the papers that are scanned in this thesis. It is about 0.06 percent. Since the topic is nearly new to the academic area even though it is being used in some business processes.
- Pang and Hindle’s paper is the main research paper about continuous maintenance. They try to describe continuous maintenance and categorized it. Also, they mentioned that, continuous maintenance is not getting enough interest as other continuous practices.

Furthermore, another similar finding is that 7 out of 29 papers are about repositories and they are not even on the subject of Repository Management. Five out of seven study on repositories, which are selected for this study, is irrelevant to the subject and their study focus is Data Mining in Repositories. Literature review provides valuable information about repository and continuous maintenance. Main

focus of the papers about repository is data mining. As it shown from the literature review, thesis's subject is not getting any attention. There are many studies in continuous practices but unfortunately there are not so many in continuous maintenance in the literature.

When the causes of these results are analyzed, it is possible to find an answer in the well-known book of "Software Maintenance: Concepts and Practice by Grubb and Takang" (Grubb, and Takang, 2003). This book mentions "The Nomenclature and Image Problems" of software maintenance. The similar problem describes by Higgins as well (Higgins, 1988):

"Software maintenance, on the other hand, entails very little new creation and is therefore categorized as dull, unexciting detective work."

They found out that, even though software maintenance is a significant work, developers are not interested in software maintenance because of its lack of creativity. In addition, another finding is that there is not consensus on the terminology and that also affects the developer's point of view on software maintenance. Our findings in our literature review shows the similar result with the Grubb and Takang (Grubb, and Takang, 2003) indicates.

Result 1

There are limited number of academic research papers in software continuous maintenance, and repository management for CM areas.

Result 2

Continuous maintenance is one of the most crucial part of the software but academic studies and developers are not interested in it as much as it deserves.

Result 3

Maintenance is a really difficult and a costly job. Developers avoid doing it. Since it is not a creative job, developers find it boring to do.

CHAPTER 4: CONTINUOUS MAINTENANCE

Maintenance and continuous maintenance comes from the same origin but with differences. Maintenance of a software project has a timeline. It can be done in monthly, yearly or daily. On the other hand, continuous maintenance is a continuous process. Continuous maintenance can be classified as the extension of continuous integration. This definition tells us that CM is not only interested in basic things like server maintenance or time efficiency. It also collaborates with all processes. One other difference is, continuous maintenance is automated unlike maintenance. Automatization of the continuous maintenance identify the fact that the process is not done manually. Instead, continuous maintenance has a specific time for when to trigger the operations.

Continuous maintenance can be classified as Pre-Production and Post-Production. Pre-Production CM refers to the activities happen before release, as it is understood from its name. Post-Production maintenance activities practice after release period.

In software engineering, repositories usually refer to the main point to store data about a system or code. Mostly repositories are used for version control during development and maintenance. Version control means that a system records changes to a file or a project in time so that developer can recall the specific version whenever the developer wants. Whenever a change committed and deployed to the repository; these changes are saved, and new version is created. It's a cumulative process, which means; with every version, size of the repository will be growing as well. A badly managed software repositories can lead to a number of problems, including wasted time and programmer force in development, low performance, and badly source code tracking and version control in implementation.

Every project and especially continuous practices and applications highly depend on repositories and artifacts such as; databases, servers, virtual machines, storage, data, meta-data, various logs and reports.

Continuous maintenance seeks to maintain these repositories and artifacts properly and consistently through automation, summarization, compaction, archival and removal.

Continuous maintenance is an extension version of continuous integration instead of being part of it. Because, continuous maintenance not only helping to improve continuous integration processes but also, helps to maintain deployment level processes as well. It aims to improve productivity, sustainability and efficiency through automation. CM seeks to improve consistency and continuity within an IT department. CM mainly automate many manual processes to reduce maintenance costs.

CM processes can be classified into two phases based on the articles [16, 42].

4.1 Pre-Production Continuous Maintenance

Pre-production continuous maintenance is deals with the processes before the release of the project to the public use. It includes continuous integration practices, IT processes, test automations and so on. Even though maintenance' definition refers the "after product", continuous maintenance is an all different concept. Since, continuous maintenance activities carried out in the pre-production processes as well.

1. **Repository Archival:** Main essence of many continuous integration processes is repository. It provides automated build and testing. Continuous maintenance help with the monitoring content, size and the performance of the repositories. **CM will ensure that all types of repositories are maintained consistently.**
2. **Storage Cleanup:** Software projects drain a lot of storage. Continuous maintenance helps to manage metadata and artifacts. In addition to that, if needed continuous maintenance can recreate executables from repository.**CM can deduplicate, compress, and/or archive large files to use less storage.**

3. **Management Systems Automation:** Continuous Maintenance can help tracking bugs and errors when automated tests fail. In addition, continuous maintenance can report those bugs and errors on behalf of the programmers.

4.2 Post-Production Continuous Maintenance

Post-production continuous maintenance indicates the after-release period. Once the projects are in public use and accessible to users post-production is start working. Post-production makes the project's sustainable. It includes data cleaning, loggings and analytics.

1. **Temporary Data Cleanup:** Continuous Maintenance can automatically monitor, summarize, archive and clean up the temporary data that created.
2. **Logs Archival:** Continuous Maintenance can analyze the system logs about requests, exceptions and errors that kept in application. In addition, CM can compress these logs to archive and eliminate the unnecessary ones.
3. **Exception or Violation Analysis:** Exception/Violation Analysis is similar to Logs Archival. Continuous Maintenance analyze the execution and audit logs then identified the wrong behaviors.
4. **Data Elimination:** Eliminating obsolete data about customers is a legal obligation. Continuous maintenance can remove obsolete data for legal compliance. For example, holding credit card, phone or mail information has to be eliminated from the system after the retention period.
5. **Data Warehousing and Analytics:** Data warehousing is a must for lot of businesses. But data can be too complicated. Continuous Maintenance can employ Extract, Transform and Load (ETL) tools to provide a understandable analysis of the databases. In addition, it can detect the sensitive data and mask it before loading into the warehouse.

The focus of this thesis is post-production continuous maintenance. Even though repository archival is pre-production maintenance activity, repository management is all lot different. Since our goal is identify the repository management activities and minimize the errors, we will look into the post-production continuous maintenance. Data cleanups, logs and data analysis can be classified under the repository

management activities and addition to those we will look into repository archival in our case study.

CHAPTER 5: REPOSITORY MANAGEMENT

5.1 Repository

The central place that the developers can software version pull from when needed is called software repository. Easy storage, maintenance and backup of software modules can be enabled with using repositories.

Main benefits of using software repositories are as follows:

- Its makes software management easy.
  - Can be stored and grouped logically.
  - Makes it easy to identify, deploy and manage.
- Avoid multiple copies of software applications.
- Minimize security risks (read-only permissions for folders).
- Enable easy backup.

Repository contains two main location/workspace: local and remote. Remote repository is the main project location of the product. Selected repository tool keeps all the deployments. Local repository is the developer's repository which is held in their machines. Developers pull the project from the online repository and create a local one once they started to work with the project. Developer's work on their local repository since that, code changes happen in the local repository.

Once the changes pass the developer tests (unit test), changes commit and push to the remote repository to get into use from every developer. Repository relations shown in Figure 7.

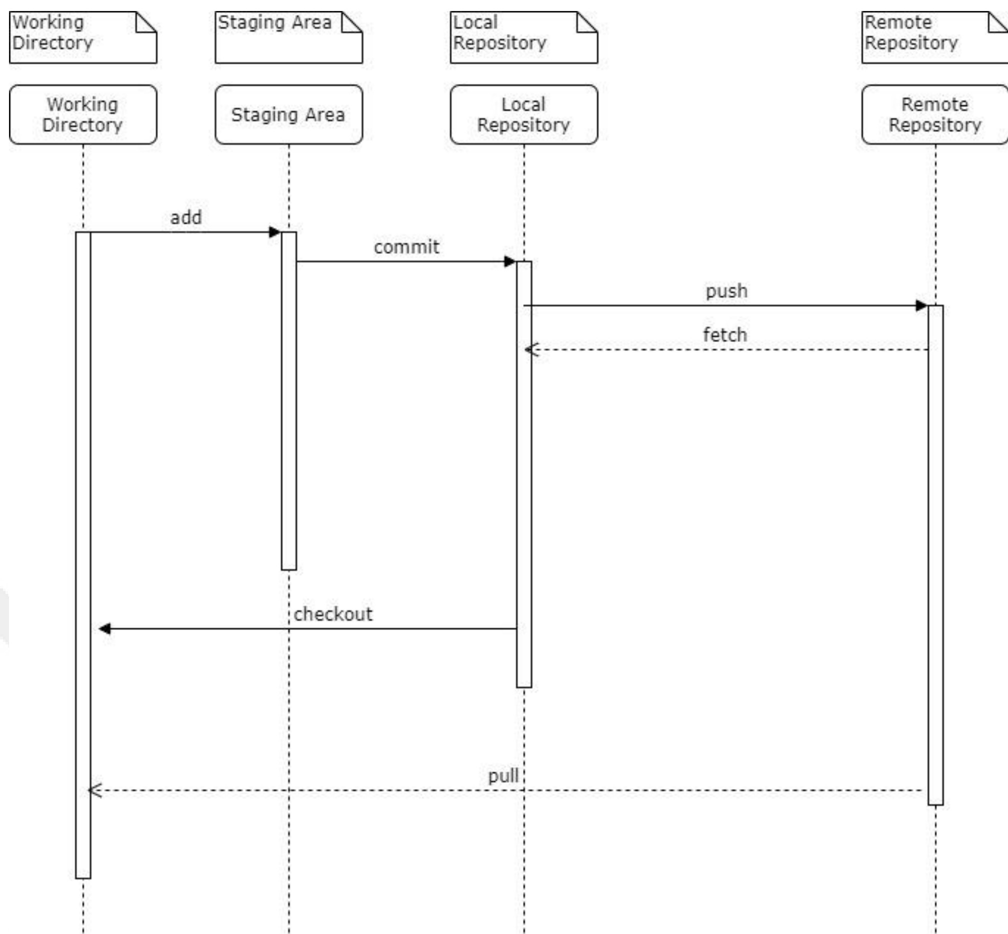


Figure 7: Remote and Local Repository Relationship

Programming languages also vary according to the repositories they used. Table 6 shows the relationships between software languages and their used repositories. Those repositories are generally accepted mechanisms in programming world.

Table 6: Software language and repository relationship

Language	Repository
Java	Maven
C#	NuGet
Python	PyPI
PHP	PECL, Packagist
Node.js	NPM

Repositories are mainly used for version control. Version control means that a system records changes to a file or a project in time so that developer can recall the specific version whenever developer want. For example, a software team updates the working system to perform better. By doing that, team changes some of the code parts. This could end up affecting the working system and make it unusable. In this case, team recall the changes that recently added and convert the system in a state which works. Whenever a change committed and deployed to the repository; these changes saved, and new version is created. Repository size increases cumulatively after each deployment. Which means; with every version, size of the repository will be grow as well. This may cause low performance and management problems.

5.1.1 Version Control

A version control system (VCS) is a software system, which tracks and manages changes in a file system in a repository over time. Modifications in software such us additions, removals and changes can be tracked using VCS. VCS can detect modifications file based and even the changes in a single line of code. Popular software industry VCS options are Git, Mercurial, SVN and perforce. In this thesis, our scope will be Git only. Because, Git is very commonly used in industry and it's a free and open source. Also, my current company and case study's company are using Git as well.

Version control systems have a special database to keep a record for every modification. If anything goes wrong, VCS offers a chance to turn back to previous working version. Thanks to this, other team members will not be affected that much. Only time they are affected and cannot work is the time between the error and going back to previous version. This way, you minimize the off-hours due to the error. Since software teams has parallel workflows that might affect each other, minimizing the losses is really important. Source code is the core of all software projects therefore human errors can be devastating. To prevent such errors version control protects the source code.

Version control systems manage these kinds of problems. To avoid that, version control systems can be used. VCS helps software teams to work without failing one another by tracking every single change by each developer.

Another important point is the relationship between VCS and testing. Code changes cannot be trusted until it is tested. Therefore, testing and development proceed together. Therefore, untested versions only increase the number of errors and difficulties in findings.

Version control software's works on any platform and supports a preferred workflow rather than dictating a single workflow to follow. Without version control systems, teams can't find out which code changes cause the error easily. This is an effort and time cost. With the version control, these problems can be inhibitable.

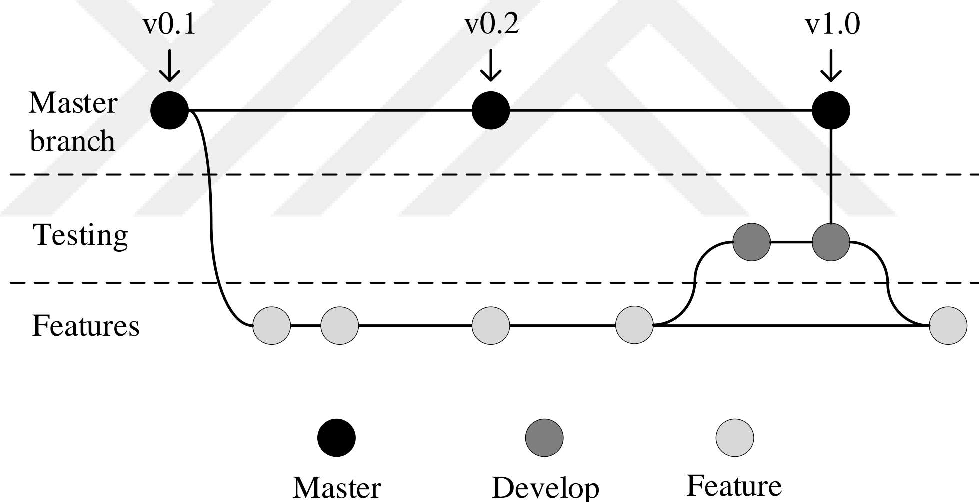


Figure 8: Version Control Workflow of Git

Figure 8 identifies the fundamental process of the version control system. There are many workflows for different kind of projects, but we choose this one since its most suitable for our case study. We have three different lines here. Lines are identified as branches. Version control system are actually tree structures. First line is called master; sometimes it is called prod or origin as well. Master line is the product's working version in real life. When developer wants to work on something or has to change/add some functions developer pulls from master branch to his/her local branches. The second line is called features. Feature branches where the coding actually happens.

Every developer or every task has its own feature branch. Feature branches are created from master/origin branch. That means when a new task comes to developer, developer pulls from the master branch into his/her feature branch and starts coding. When the developer is done with the task, again he/she pushes its changes to the feature branch first. After the developer tests are done, there is one last step before going to live. The changes are pushed to test branch. The third line is called develop/test, or we can say that this line is basically where the user tests happen. The developments from feature branches are pushed to the develop/test branch. Where the testers see if it is okay to put these changes in the working product. Develop is the exact same product as the working product in real life. Before tasks go live, they double check to see if the code changes affect anything. Usually this branch has automated updates to be able to exactly same with the product. When changes come into develop branch and tested it has to be return to the live version. Since that, twice a week, this branch is recreated with the live version/origin. There are two type of version control: Centralized and Distributed.

5.1.2 Centralized Version Control System

Centralized version control system has a single server and only works on client-server model. In this model, all version of code is in the master repository. When developer once to work with a significant part of the code, developer has to check out that part of the code from the central place and only that developer is allowed to work which means other developers cannot access that part of the code. CVS and Subversion are the best-known examples of centralized VCS systems. Centralized version control system shown in Figure 9.

The upside of this system is that every developer’s working files are known. It provides a simple way to keep track of changes. However, it has some disadvantages as well. Since, even if there is only one server failure, it affects everyone. If someone fails to deploy or cannot figure out how to solve their bugs, no one can work on the server until it is fixed. Since entire history of the project is in a single place/server; failure can risk to the lose everything.

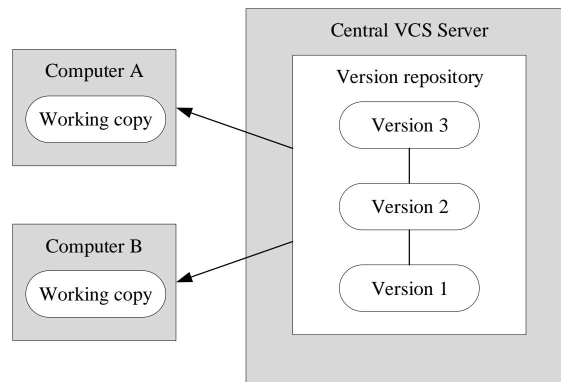


Figure 9: Centralized Version Control

5.1.3 Distributed Version Control System

Distributed version control work on a peer-to-peer model. Instead of single main repository, every developer has their own repository on their machines. Instead of checking out the central repository, every developer has a copy of the main repository. In distributed version control, since server is not centralized, if any server fails, any developer's copy can be used as a backup.

Every developer works on their local copy of the repository. There is no locking the working parts like centralized version control. When developer's task is finished and it's tested, they send a request to merge their changes into the master copy. Git and Mercurial are the best-known examples. Distributed version control system shown in Figure 10.

Every software team works different. In order to match that, distributed systems enable to work with different types of workflows. This feature is not supported in centralized systems.

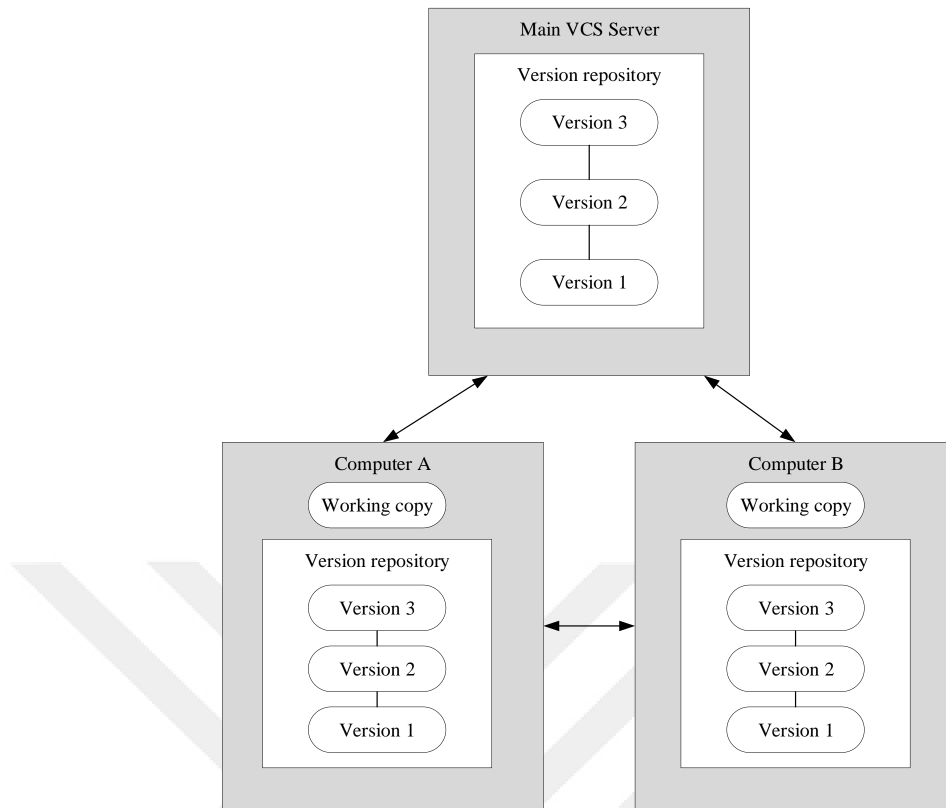


Figure 10: Distributed Version Control

5.1.4 Advantages and disadvantages of version control system types

Contrast of version control systems is going to show us why industry giants commonly use distributed version control systems over centralized and the comparison also guide us to choose a version control system for our case study.

Advantages of distributed systems. Since there is no lock mechanism, performance and time effectiveness is better. Branching and merging is easier. Finally, on account of every developer has a local repository, there is no need to connect to network all the time.

Advantages of centralized system. Easier to understand. Access mechanism is easier since it's controlled from one server. Since only one developer can work at a single time, there is no need for merge mechanism.

Contrast represent the lock mechanism is an inhibitor for software development process. Especially in agile development, teams meant to work parallel. Because of this, distributed version control systems are commonly used by industry giants. In our case study we will use Git which is a distributed version control system.

5.1.5 Benefits of version control systems

Version control systems ensure that no data is lost and backups existed in the repository.

Benefits of version control systems:

1. **History:** Every change including creation and deletion is logged. History also detect every author/developer date and any written description to the changes. This helps the identify the cause of any error. Also, it shows the efficiency of the team as well.
2. **Branching and merging:** Having branches for each developer or each task help to distinguish the workspaces and having an individual workspace can increase the performance of the developers. It also, keep multiple people to work at the same time without affecting from other developer's changes. And merging helps to detect any bugs that can cause system failure. When developer finish what

they are doing, they merge it to the master branch which facilitate to identify possible failures that the changes can cause.

3. **Traceability:** Every change is kept in the system with description and author can increase the analysis reports. Every chance is kept like this; id, author, code changes and description. Since this kind of information is kept, tracing something is much more efficient. Every code change and their purposes are logged in the system. Understanding the code, finding some function or fixing bug is faster and effective.
4. **Troubleshooting:** When something goes wrong, developer can easily compare the failed version with the latest version to see the error. This way developer spends less time identifying the issue.

The Figure 11 shows the popular version control system options based on the popularity rating based on a research made by a software company called g2 (G2, 2019). There are several version control systems in the industry but Git is the one that commonly used.

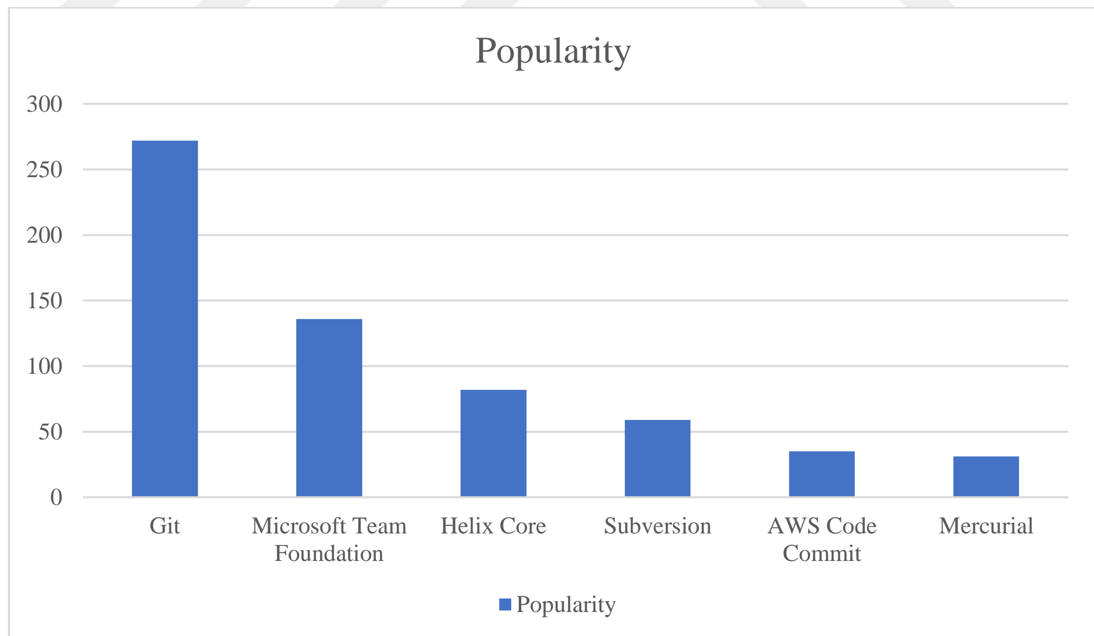


Figure 11: Popular Version Control Systems

5.2 Repository Management

Managing the repository files and interacting with the repository means repository management (RM). These tools interact with the repository to search artifacts, files and models. When developers download repository and install it their local machine's, repository management is identified the dependency without any customization. RM makes configurations more easily.

Repository management deals with:

- Interact and manage with artifacts and meta-data
- Dependency, library and version control
- Configuration
- Automated deployment
- Deployment management
- Failure reports

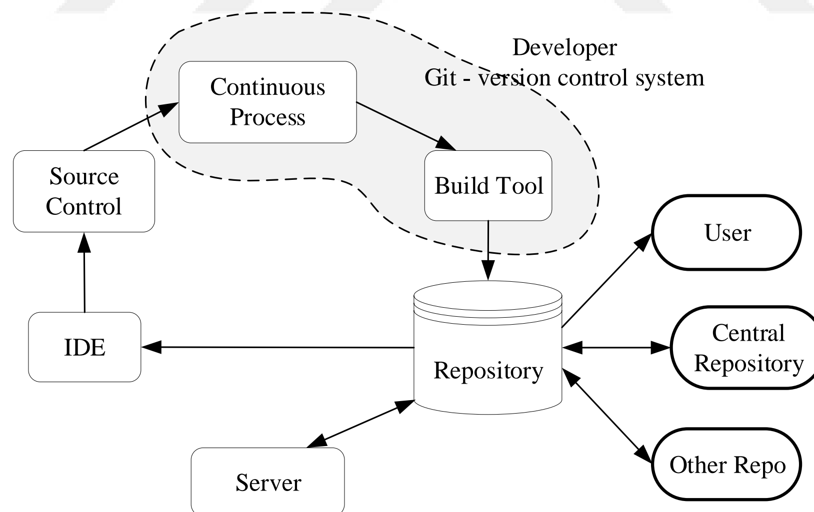


Figure 12: Repository Management Visualization

Figure 12 in the above shows the interactions of repository management. In the right hand, we have the repositories, which hold in the cloud. Central Repository represent the main projects' repository. Other repositories are the interacted repos for your main repository. Customer represent the live version that the customer is using. In the left hand, we have the SDLC process. Projects is pulling from the central repository to the developer's selected IDE. For example; eclipse, IntelliJ or NetBeans. Source control is referred to GIT. GIT is integrated into IDE. Once developer wants to push its changes, GIT integration come into use. Pushing starts the continuous process. Bamboo, Jenkins or Gitlab are the examples for the continuous process. Process that goes through continuous process is built in build tool. In java, this is referred to Maven. Build tool (maven) controls the dependencies, projects and their version. After all of these processes done and succeed, repository management push it to the main repository which is central repository. When central repository is updated, customer can see and use the latest version.

Using repository management has significant advantages. Advantages listed in below;

- Reduced bandwidth rate
- Reduced number of downloads from repository
- Less time spends
- Increased performance for SNAPSHOT repositories
- Integrate with other repositories
- Reduced build times
- Improved collaboration by providing a central location.
- Easier access to components for developers
- Simplified development environment
- Flexibility to us

CHAPTER 6: THE CASE STUDY

The aim of the case study is to contribute to existing literature by identifying and finding solutions to the challenges such as impact analysis, failure tracking and version controlling. In this case study, the small development team has added some new features to an existing product by mimicking continuous practices, continuous maintenance and repository management, and also behaving like global software development team. Also, continuous maintenance biggest strength is atomization, and in order to be beneficial, continuous maintenance has to be applied to repositories as well. In this case study; we are using Git as a repository management tool and we try to observe the lacking parts of the repository management in the name of continuous maintenance.

6.1 The Company and Project Background

Case organization is a software development company located in Tallinn, Estonia since 2016. IoniaOU is a start-up organization, founded by Mr. Mustafa Tufan. Their main focus is to ensure end user's satisfaction. They are developing web and mobile applications. IoniaOU currently running three projects: one mobile application and two web applications. Mobile application is actually a mobile game called Roll'n Break. Roll'n Break have 500 active users daily. Game is available for both App Store and Google Play. The two web applications are called Btckit.io and Alivee. Btckit.io as a web application that aims to server Bitcoin tools to everyone for free. It's also an open source project. Btckit will be an open portal all about bitcoin. Currently, they have only one application on Btckit and its wallet generator. Basically, what it does is randomly generating a wallet key and checking if there is a bitcoin wallet for that key. It's only for educational and experimental use right now. All operations are logged and provide a safety for all of it's generated keys.

The third application is our focus in the case study, titled as Alivee. Alivee is a web application for internet link shortener. Company's motto is "*your link is going to be Aliv.ee*". Any link located in internet can be transferred here and replaced by the desired short version. It is a basic application and easy to use. IoniaOU wants to update its project to a new version. They want to add a feature, which helps user to shorten their link randomly instead of manually entering. That is where our case study jumps into.

The plan of case study described in Figure 13 according to the objectives of the case study and company's principles. The crucial first stage in the planning process is defined as the description of the development environment and the technologies, and languages that are going to be used. The next stage addresses the information needs of the company. After articulating the environment and information needs, design of the project occurred. The implementation phase begins after the planning period. In the final stage we focus on the evaluations and the lessons learned that are collected from the case study. The plan of the case study summarized as follows:

1. Define development environment and technologies
2. Gathering requirements
3. Designing the project
4. Implementation
5. Evaluation

In our case study, the developer team consists of three developers who are deployed in different locations, and they bring together on the internet for this case study. However, one developer is located in İstanbul and two are located in İzmir. This feature provides some evidence, the project represents the global software development features and also the development team is a small team.

However, this team is responsible to add new features to the current version applying agile software development methodology and continuous practices. So that this work is mimicking the maintenance and repository management. Due to these features, the characteristics of the project are best suited to perfective continuous maintenance, including repository management for a small development team.

In this case study, the company used the following languages, tools, and frameworks: 1) Java, 2) Spring Boot, 3) JPA, 4) JavaScript, 5) PostgreSQL, 6) Maven, 7) HTML & CSS, 8) Rest Api Services

6.2 Development Phase

In this section, gathering requirements, designing the project, implementation is grouped as a development phase. Development phase described as follows.

6.2.1 Project instructions

Take into account the company's main objective and the problems mentioned in the previous section, we organized our case study as:

To generate new version that leads to new ideas with potential improvements of the current system and in the process finding major and minor bugs and making the project more efficient with the usage of the best practices. In addition, to investigate the research question 3 to identified challenges in continuous maintenance process.

The project selection for the case study practice based on the criteria: 1) simple, 2) easy to test, 3) familiarity with code and software languages, and 4) fitting requirements with the case study.

In addition to these criteria, project can be categorized as a continuous maintenance project. Since project is already working, additional features fits under the perfective category. Case study team members locations strictly suggest to follow global software development. With the usage of global software development, using a version control system is essential. That brings the repository management. So, in summary following categorizations provided for the case study:

1. Continuous maintenance
2. Perfective maintenance
3. Global software development
4. Repository management

6.2.2 Project structure

Aliv.ee is a Spring-boot project and its structure is shown in Figure 13. Project has a simple structure. Its main component is Link object. Link object is used with the RestApi which contains LinkService, LinkServiceException and LinkServiceImp. When the link conversion operation happen DataBlock is updated. PostgreSql is used for data storage. Link objects and their shortened versions are kept in PostgreSql.

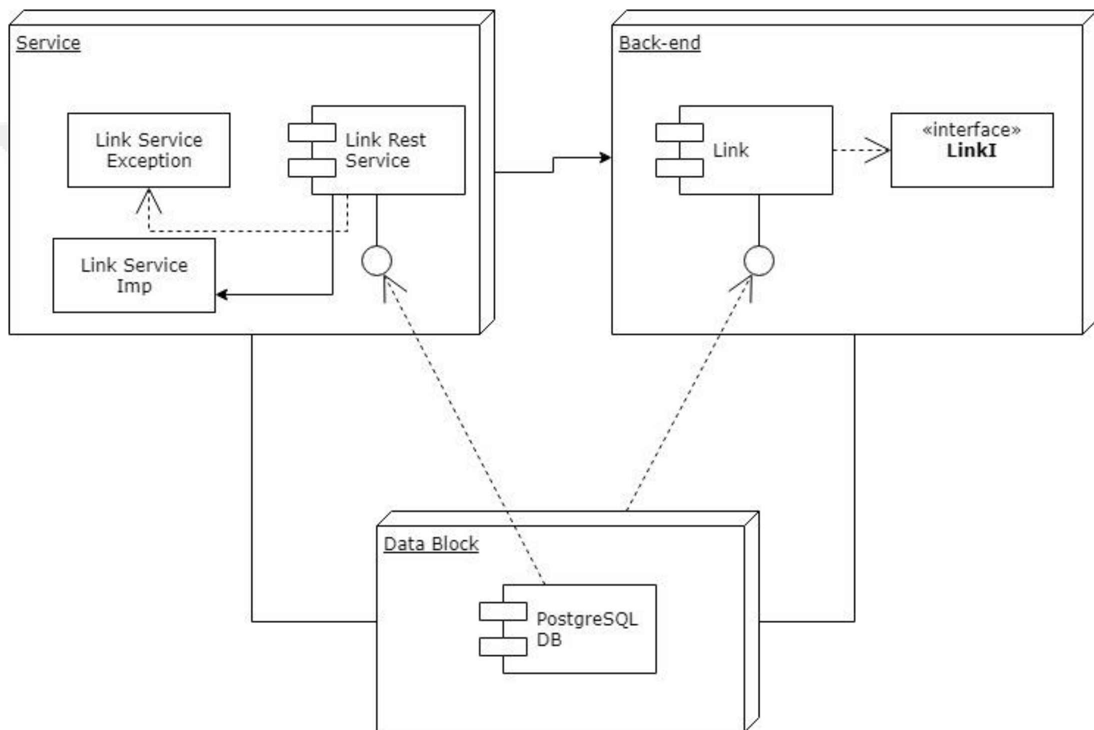


Figure 13: Aliv.ee Back-end Structure

Once the link converted the result screen is shown in Figure 14.

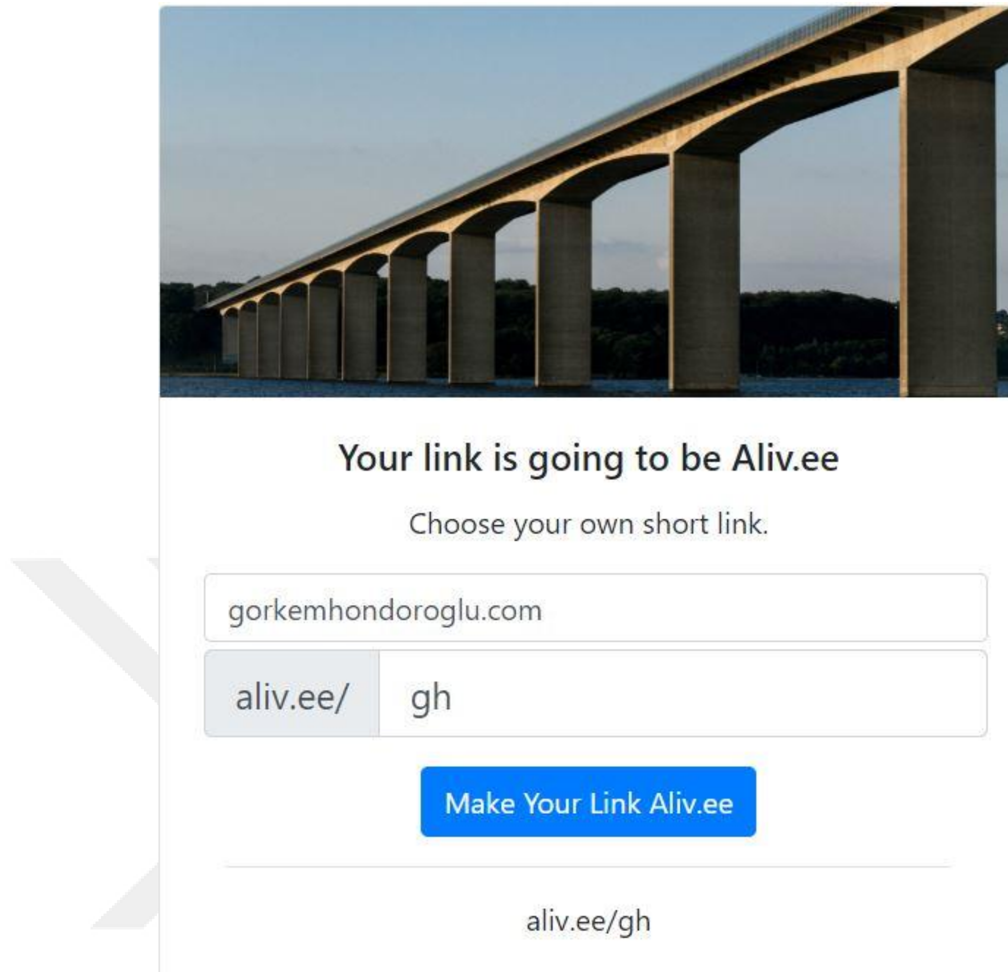


Figure 14: Result Screen

6.2.3 Installations

First step of the installation is getting the project from Git and clone it to the computer. In order to do that, following steps are implemented:

1. Download Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
2. Install Git
3. Check that Git is recognized by open command line and write git.
4. Create a folder for the project
5. Open command line and go the project's path that created.
6. Go to project's page on github and clone it.
7. Write the following command on command line git clone "project path on github"

6.2.4 Analysis and design

In this case study, we used UML tools to design general concepts and structures. UML diagrams help us to see the general structure of the project. It helps us to see the connections between packages and classes so much easier. In case study, we are trying to add new features to the existing structure. To compare the existing functionality to the new features; use case diagrams and sequence diagrams draw. Use case diagram, Class diagram, Sequence diagram, modeling of executables, work breakdown diagram used or this project.

1) **Work Breakdown Structure:** In work breakdown diagram, alive case study's work structure can be seen. From background works such as analysis, getting requirements, identifying team members to release phase shown. Diagram shown in Figure 15
2) **Use Case Diagram:** High level use case diagram is shown in Figure 16. The main scenario of user's signing up and usage of link shortener manually can be seen in the diagram. It's a high-level use-case as well. Every part can be seen as different use cases. Random link creation's user case can be shown in Figure 17.
3) **Class Diagram:** Alivee project structure from link models to controllers described in class diagram. Every class functionalities can be seen as well. The class diagram is shown in Figure 18.
4) **Modelling of Executables:** Main components that Alivee application uses shown in the diagram. The diagram shown in Figure 19.
5) **Sequence Diagram:** The sequence diagram shows the current working system which is manually changes the given link into shorter one. It represents the objects and classes used in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Operators in sequence diagrams are user, Alivee System and the Alivee database. The sequence diagram is shown in Figure 20. New added feature is random link creation and its sequence diagram can be shown in Figure 21.

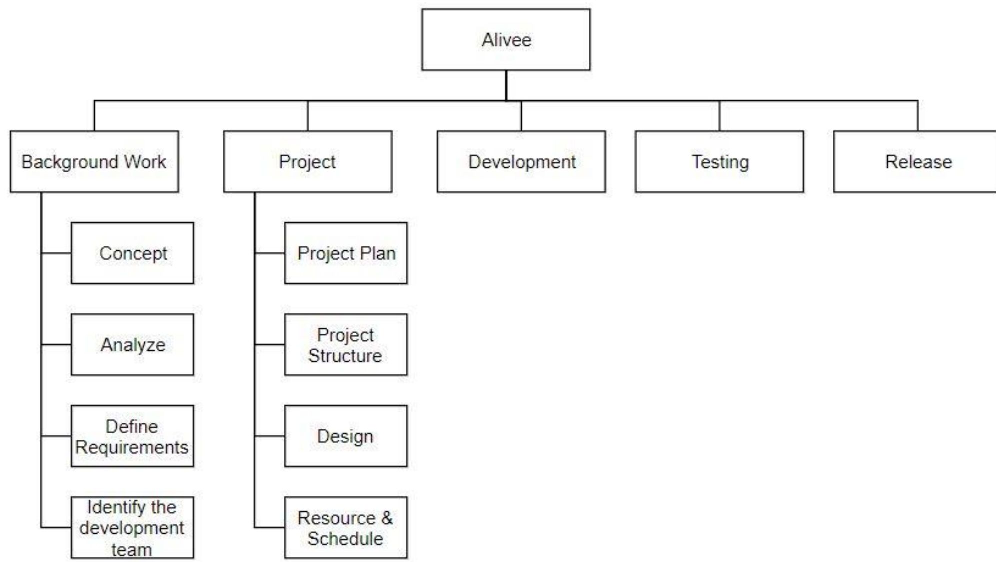


Figure 15: Work Breakdown Structure

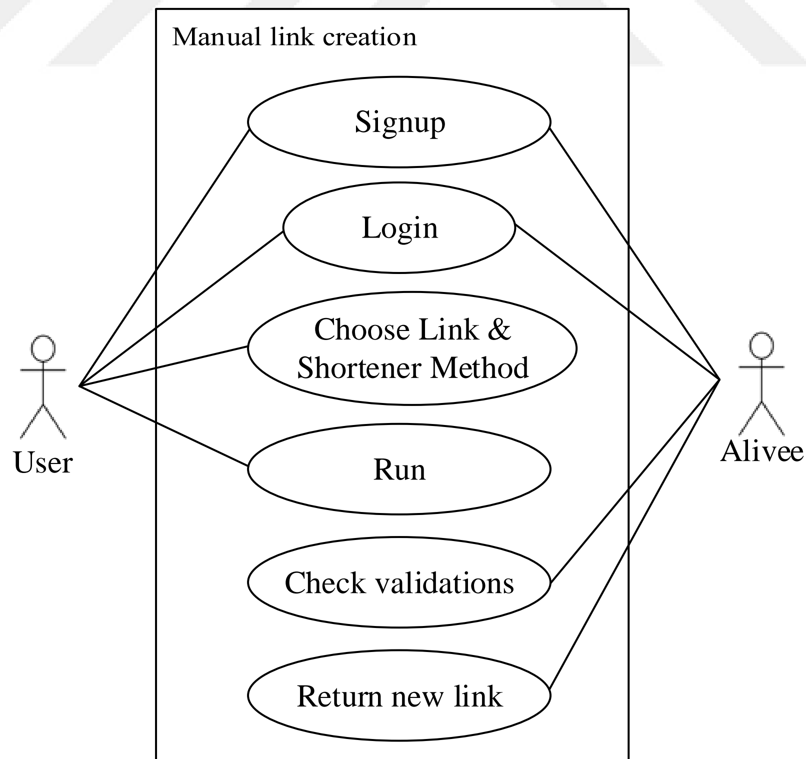


Figure 16: Use Case Diagram of Manual Link Creation Scenario

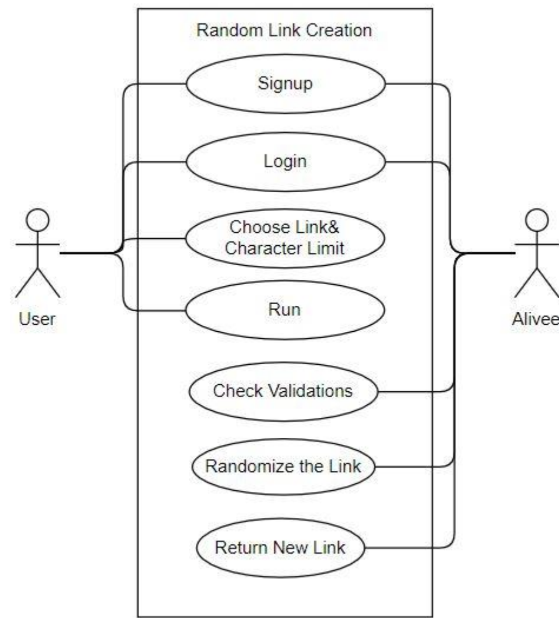


Figure 17: Use Case Diagram of Random Link Creation

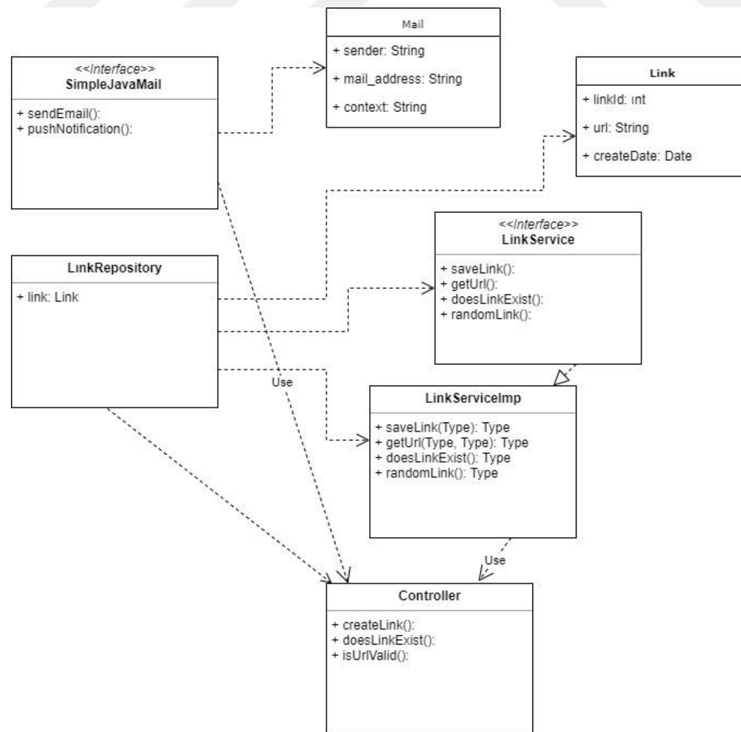


Figure 18: Class Diagram of the Project

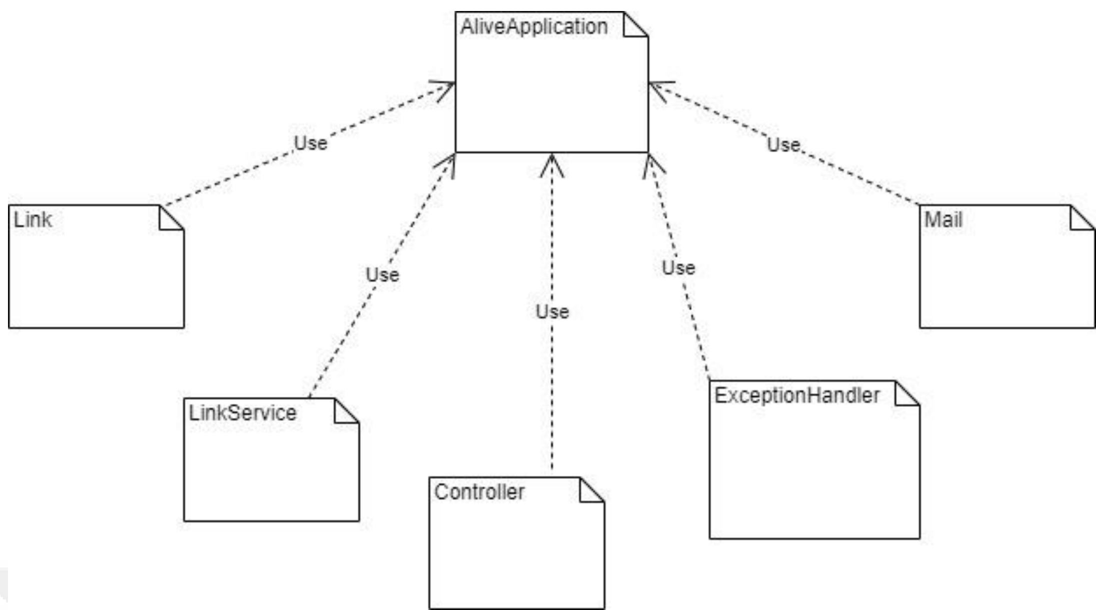


Figure 19: Modelling of Executables

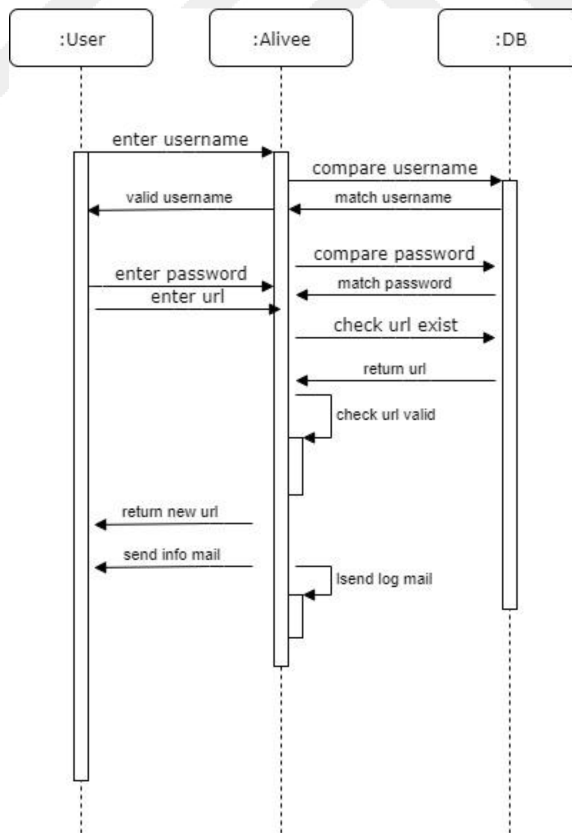


Figure 20: Sequence Diagram of Manual Link Creation

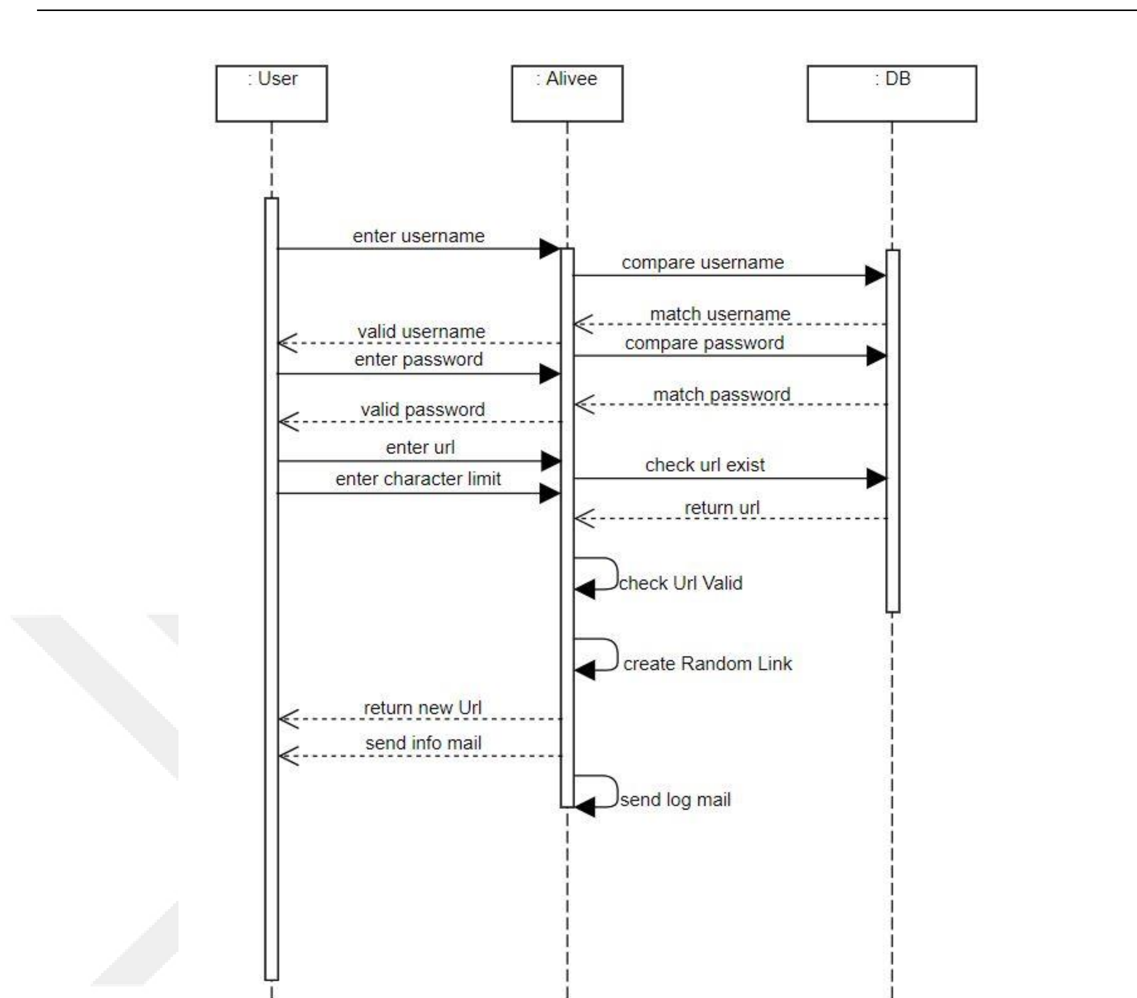


Figure 21: Sequence Diagram of Random Link Creation

6.2.5 Code structure

Project code based is developed with Spring framework. Spring framework created for Java enterprise projects and make it so much easier to develop such projects. It consists of modules such as Security, Web, Servlet etc. Our project Alivee is a web application so in our project we used Web, Security, maven, devtools, data-jpa and spring boot modules. Spring dependencies in pom file is shown in Figure 22.

```
<dependencies>
     <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-
web</artifactId>
      </dependency>
      <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-
thymeleaf</artifactId>
      </dependency>
      <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-devtools</artifactId>
           <optional>true</optional>
      </dependency>
      <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-data-
jpa</artifactId>
      </dependency>
      <dependency>
           <groupId>org.postgresql</groupId>
           <artifactId>postgresql</artifactId>
      </dependency>
      <dependency>
           <groupId>commons-validator</groupId>
           <artifactId>commons-validator</artifactId>
           <version>${commons-validator}</version>
      </dependency>
      <dependency>
           <groupId>oro</groupId>
           <artifactId>oro</artifactId>
           <version>${oro}</version>
      </dependency>
      <dependency>
           <groupId>javax.mail</groupId>
           <artifactId>javax.mail-api</artifactId>
           <version>1.6.2</version>
      </dependency>
      <dependency>

      <groupId>org.springframework.security</groupId>
           <artifactId>spring-security-web</artifactId>
           <version>${spring-security.version}</version>
      </dependency>

</dependencies>
```

Figure 22: Dependencies of the Project Alivee

We needed a relation database management tool to do our database works. Since that; postgreSql is selected. Alivee application is not a complex app since that we didn't need high complex queries or procedure structures and mainly used for read and write executions. In addition to that, PostgreSql is an open source system. Entity Relationship diagram of our project is shown in Figure 23.

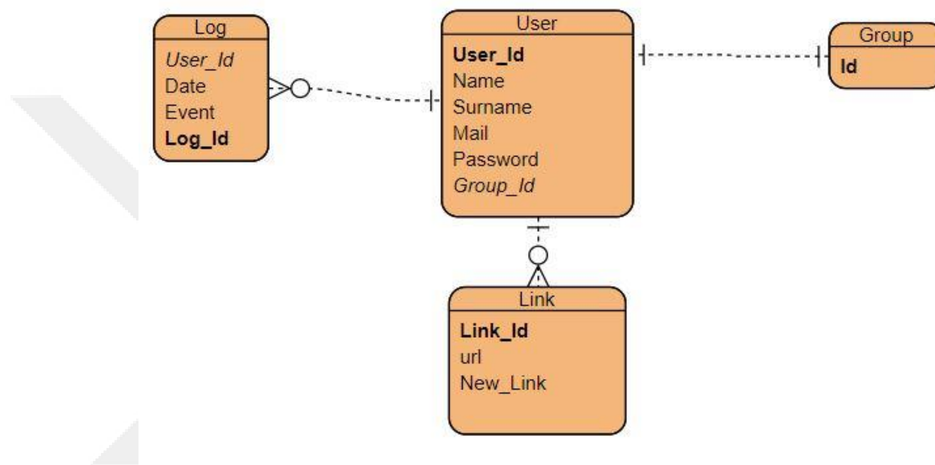


Figure 23: Entity Relationship Diagram

Database Connection operated with JPA. JPA helps with object persistence. In our relational database we have tables, attributes, constraints and keys. However, in Java we can't address that constraints, keys efficiently. In order to connect to database and operate some SQL queries we needed to write large number of line codes. With the help of JPA, this is not necessary. JPA is handled those methods and function with an easy annotation structure. In our project we used Repository annotation. Example of the usage is shown in Figure 24. Repository annotation provides us storage, retrieval, search, update and delete operation by itself.

```
@Repository("linkRepository")
public interface LinkRepository extends JpaRepository<Link,
String>
```

Figure 24: Repository Annotation JPA

Controller pattern is used to handle all request handling between our service and users. Annotations used for controller pattern too. In order to tell spring that the class is in fact controller class; annotation @controller must be used. Also, request functions handled with annotations as well. One of the controllers class' is shown in Figure 25.

```
@Controller
@RequestMapping("/api")
public class LinkController {

   private LinkService linkService;

@RequestMapping(value = "/createLink", method =
RequestMethod.GET, produces = { "application/json" })
   public @ResponseBody Link
createLink(@RequestParam(value = "linkId", required =
true) String linkId,
         @RequestParam(value = "url", required = true)
String url) {
      if (URLUtil.isURLValid(url)) {
         return null;
      }
      if (linkService.doesLinkExist(linkId)) {
         return null;
      }
      return linkService.saveLink(new Link(linkId, url));
   }
   @RequestMapping(value = "/createRandomLink", method =
RequestMethod.GET, produces = { "application/json" })
   public @ResponseBody Link
createRandomLink(@RequestParam(value = "linkLenght",
required = true) Integer length,
                                  @RequestParam(value =
"url", required = true) String url) {
      Link link = new Link();
      if (URLUtil.isURLValid(url)) {
         return null;
      }
      link = linkService.randomLink(length, url);
      return linkService.saveLink(link);
   }
   @RequestMapping(value = "/doesLinkExist", method =
RequestMethod.GET)
   public @ResponseBody boolean
doesLinkExist(@RequestParam(value = "linkId", required =
true) String linkId) {
      return linkService.doesLinkExist(linkId);
   }
```

Figure 25: Controller Class

Remaining operations perform with service and spring structures. Examples of Service Interfaces and Implementations can be found in Figures 26 to 28.

```
public interface LinkService {

public Link saveLink(Link link);

   public String getUrl(String linkId) throws
LinkServiceException;

   public boolean doesLinkExist(String linkId);
   public Link randomLink(Integer length, String url);
```

Figure 26: Link Service Interface

```
@Entity
@Table(name = "link")
public class Link {

   @Id
   @Column(name = "linkId")
   private String linkId;

   @Column(name = "url", nullable = false)
   private String url;

   @CreationTimestamp
   @Temporal(TemporalType.TIMESTAMP)
   @Column(name = "create_at")
   private Date createAt;

public Link(String linkId, String url) {
      setLinkId(linkId);
      setUrl(url);
   }
   public String getLinkId() {
      return linkId;}
   public void setLinkId(String linkId) {
      this.linkId = linkId;}
   public String getUrl() {
      return url;}
   public void setUrl(String url) {
      this.url = url;}
```

Figure 27: Link Entity

```
   @Override
   public Link saveLink(Link link) {
      return linkRepository.save(link);
   }
@Override
   public String getUrl(String linkId) throws
LinkServiceException {
      try {
         Link link = linkRepository.getOne(linkId);
         return link.getUrl();
      } catch (EntityNotFoundException ex) {
         throw new LinkServiceException();
      }
   }
@Override
   public boolean doesLinkExist(String linkId) {
      return linkRepository.existsById(linkId);
   }
   @Autowired
   @Qualifier("linkRepository")
   public void setLinkRepository(LinkRepository
linkRepository) {
      this.linkRepository = linkRepository;
   }
   @Override
   public Link randomLink(Integer length, String url) {
      if (length < 1) throw new IllegalArgumentException();
      StringBuilder sb = new StringBuilder(length);
      for (int i = 0; i < length; i++) {

         // 0-62 (exclusive), random returns 0-61
         int rndCharAt =
random.nextInt(DATA_FOR_RANDOM_STRING.length());
         char rndChar =
DATA_FOR_RANDOM_STRING.charAt(rndCharAt);

         // debug
         //System.out.format("%d\t:\t%c%n", rndCharAt,
rndChar);
         sb.append(rndChar);
      }
      if (doesLinkExist(sb.toString())) {
         return randomLink(length, url);
      }
      else {
         Link link = new Link(sb.toString(), url);
         return link;
      }

   }
```

Figure 28: Link Service Implementation

6.3 Observations of the Case Study

The project lasted two weeks. Our development team consists of three people located in Izmir and Istanbul. Because of the location differences and to follow best practices, we used version control system. Bitbucket is used for version control. Version control systems used git-flows which explained in Section 5 and we used feature branch git-flow with three branches. Our branches are feature, testing, and master. Everybody in the development team has to follow the same procedure to push code. The master branch is the main branch which is in use. When the developer started coding, he/she has to create a feature branch from the master branch. This created feature branch is the working copy of our machines. When the development finished, a developer has to open a pull request to copy these changes into the testing branch. Testing branch is a stabile version of the working copy with new features so if the developer's request to the testing branch is successful, then the other two developers start testing these features in the testing environment. Since the project is already in use and we have to add new features, continuous maintenance practices used in the case study. CM and testing goes hand to hand, but the lack of team members pushed us to do all phases together. Figure 29 shows the interaction of continuous maintenance. In better development teams, usually testers, analyst or even people from business test these kinds of changes. Since we were only three people, we had to test as well. Even though the developer team consist of three people, encountered problems are generic. Case study' findings can be generalized for every developer teams. Code based and repository-based problems can be minimized with continuous maintenance.

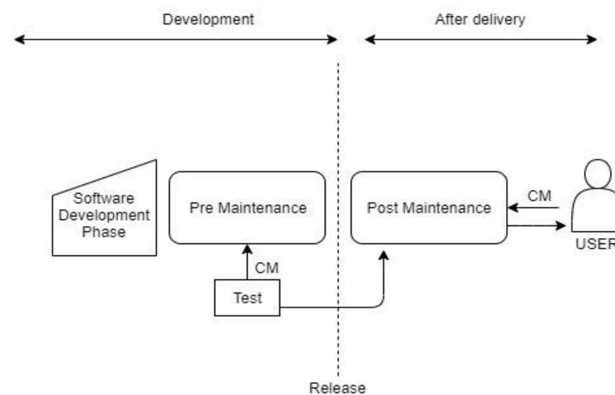


Figure 29: Continuous Maintenance Interaction

Repository activities practiced during the case study as well. Repository management is mainly done with version control. Every developer has the mirror version of the master code. This helps us to close the knowledge gap about the domain since we were all at the same page with the same exact copy of the master code. Since we used feature branch git flow; at the start of each task feature branches created. This way we minimize the impact of parallel development. But again, not everything was perfect. The location difference affected us. We did have communication issues and that leads us to miscoordination problems. Every developer in our team had a different understanding of the of the software development process, usage of git and testing. The main issue was the lack of development standards in the project.

Because of the location difference, in the development phase we did had communication issues. One of the developers, try to dockerize the application in order to perform better in local machines. However, to achieve that property file has to be updated, in fact, a second property file has to be created in order to separate application properties between dockerized and normal versions. But instead of creating individual property file specialized for docker, he updated the property file which used by all of the developers. When we pull the changes and try to run the application, the application did not run at all. Configuration files were updated but in our local machines, it did not configure to look new containers that docker created for the application. This problem could simply be avoided with communication. If we could communicate better and in line with coding decisions, we could have solved this kind of problem beforehand. Error example shown in Figure 30.

```
'> <Inconsistent security configuration, weblogic.management.configuration.ConfigurationException: Cannot find identity keystore file null on server AdminServer
<Not listening for SSL, weblogic.management.configuration.ConfigurationException: Cannot find identity keystore file null on server AdminServer.>
```

Figure 30: Configuration Error (Company info is hided)

The first thing that version control couldn't help is the used libraries. In every project, libraries are used for numerous subjects. In our case, in spring application we have to identify the libraries that we used as a dependency to the project. Dependency

is a keyword for spring to understand and read the library. These configurations made in pom.xml file. Pom file can be seen in Figure 20. Identifying dependencies requires the library's version. In our case study, we needed to send mail after every link creation. In order to do that, we add a dependency for the mail library. The version was set to 1.6.1 in the addition phase. Once we finished coding, we started to test the new features. Whenever we create a new link, mail should send to the user's mail address but instead of sending, the application crashed. We tracked the history in repo, we double check the code, but everything seemed fine. After hours of research, we found out that the library's version is changed from 1.6.1 to 1.6.2. Since there was no documentation in the library's maven page or anything, we could not find it that simple one-line code change.

Another big problem that we encountered during the case study is impact analysis. In every software project, code parts, methods, functions or even a simple variable could be used or called from different places. Thus, changing a function requires to do changes to its workspace as well. In our case study, three different modules used mail functions. Those are for mailing the user for their link creation, informing the admin about the new link and a daily log information mail to admin. One of the developers changed the main mail method. But he forgot to update its touchpoints/hierarchy. Without noticing the hierarchy, he commits the changes to the test branch. When we try to test our developments, we could not due to the incompatibility of the mail functions. Daily log mail works with the scheduler and runs every day at the same time, 23:00. Since the scheduler' functionality is different from the other mail methods, scheduler has to be updated as well. When the scheduler's time comes, the scheduler could not call the mail function and that returns a connection error. Skipping impact analysis, also damage the repository as well. Since the codebase in the main repository is incorrect, the repository's reliability decrease. Figure 31 shows the example of an error.

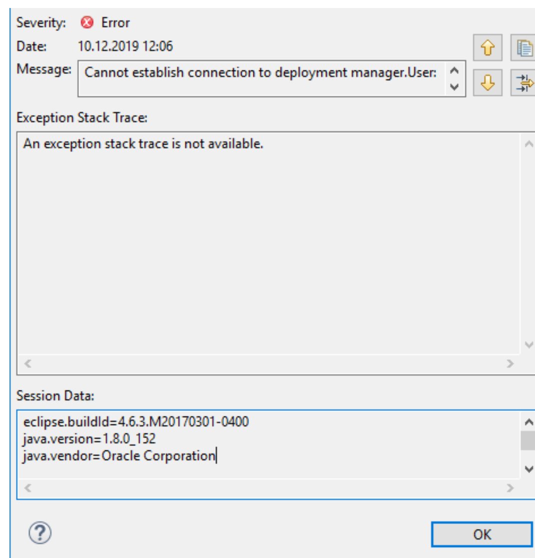


Figure 31: Connection Error

Another problem in repository management is failure tracking. For development teams that use some version control program--in our case it is bitbucket--, there are best practices to follow. Those best practices show you how to handle commits, merge, archive, etc. One of the best practices suggests that feature branches should be created for every task or in some development teams that case is every release. In addition to that, every task should be committed with the minimum number of commits. Every commit has a particular identification number and tracking is easier with minimum commits. In our development process, one of the developer's habit was committing every single part of the task independently instead of using one or two commits. In first, this habit did not seem harmful. However, when the app is crashed, it gets harder to find the main cause by analyzing the commits. If every task could have committed with the minimum number of commits, we could have easily detected the error. Because in one commit, each change made in a single class can be seen. The difference between the last version also available in commits. But he did use a lot of commits for one single task. For this reason, when the failure happens, we needed to check every commit and compare each commit in between too to understand what changed made. An example of a commit history can be seen in Figure 32.

gorkemh	918f821a334	Mail source changed	Nov 25, 2019
gorkemh	b25083bf05f	Random link creation method updated	Nov 25, 2019
semihu	909aac9ff7d	Null pointer hatası düzeltildi	Nov 22, 2019
hakani	ca8a79fc113	Validasyonlara, havuzda var mı kontorlü eklendi.	Nov 21, 2019
hakani	76958c48189	Mesaj içeriği değiştirildi.	Nov 21, 2019
hakani	ece16c2b65c	Düzenleme yapıldı.	Nov 21, 2019
hakani	bbe1460b057	Validasyon düzeltildi, Birden fazla mesaj verme güncellendi.	Nov 21, 2019
gorkemh	8ce31532627	Mesaj Bundle a taşındı. validasyonlar eklendi,e fatura kurumu olması durumundaki validasyonlar gerçekleştirildi.	Nov 21, 2019

Figure 32: Bitbucket Commit History

Coordination issues are another problem for teams that used version control systems. Since it allows parallel programming, developers could work on the same class or even on the same method as well. But that does not mean they are doing the same thing. One developer could change the method's structure, and another could add logging mechanism to the same method. Because of this conflict happens a lot. Changes have to be pushed on the main repository but since both developers' changes happen to be on the same class, the second developer gets a conflict. Conflict happens because the first developer changes the class and its not the same as the second developer's class. When the second developer tries to push it gets the conflict since class mismatch. This also can be categorized as a communication issue as well.

In summary, location difference caused a communication issue, which leads to fail repository and make it hard to maintenance. While doing continuous maintenance library issues may be occurred. While project firstly developed, libraries may be used and after a while those may be outdated. In maintenance, libraries must be controlled and updated due to the library or environment changes. Another big problem is impact analysis. In maintenance, while updating a method to better performance or adding new features etc., that methods workspace needs to be updated as well. Forgetting that can cause big problems and a fail repository as well. To failure track, minimum number of commits has to be push to the repository in order to perform better and used less time to figure the errors. In order to prevent that, version control systems especially Git, may have to add new features to its current versions. This feature can prevent developers from unnecessary commits by not allowing them if commits only consist of spacing, indentions or etc. Lastly, communication issues also cause coordination issues. In order to avoid conflict, developers have to be coordinated while using repositories. In order to do an efficient maintenance project, testing has to be integrated

throughout the entire process. Testing and maintenance are unified, and Figure 33 shows the relations between maintenance and testing. Continuous practices propose the "atomization" of the processes. Hence, in continuous maintenance, automated testing has to be integrated. Version control systems usually has a testing branch where testers and developers test the code changes manually. However, testing branches should be automated. Developers or testers prepare a set of unit tests that examine the entire project. These unit tests place on the testing branch and when a developer try to push to the testing branch, those tests will run automatically. If the code passes all the defined unit tests, then the code is set to be push to the master branch. Otherwise, developer has to update the code in order to fix. Figure 34 shows the automated unit test systems. There are a few automated test tools in the industry. Most known are; Jenkins, Travis and Bamboo. These tools are not specifically for continuous maintenance and automated tests, but they cover all the aspects of continuous integration phases. Even in those tools, continuous maintenance is ignored. There is no exclusive tool for automated unit test in version control systems.

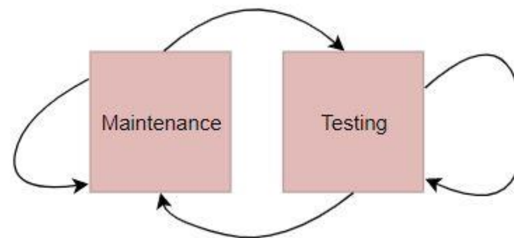


Figure 33: Maintenance-Testing Relation

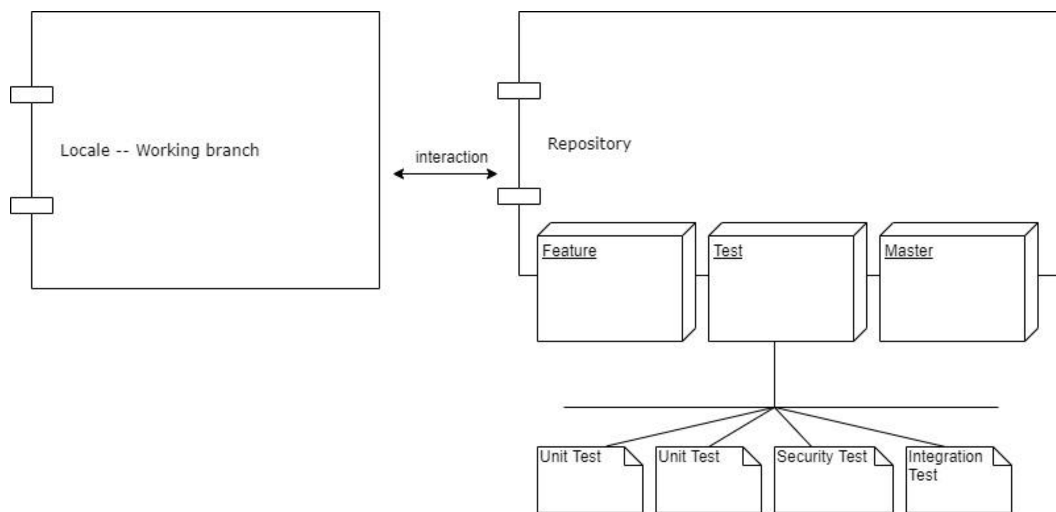


Figure 34: Automated Unit Test System

6.4 Findings of the Case Study

After the case study and literature research, we find out that continuous maintenance has no significant place to put in development processes. But in Agile world where almost every software development conducted with Agile standards. Which means that, software development happen rapidly, continuously. For this reason, continuous maintenance should be integrated into Agile development processes as well. We can say that, continuous maintenance is not only a part of continuous practices but also encapsulate. Figure 35 shows the continuous maintenance place in software development especially continuous software development processes. The figure also shows the defined tests on specific branches in order to automate the testing processes. Testing processes will return the build results of the tests to the developer whether it is successful or not.

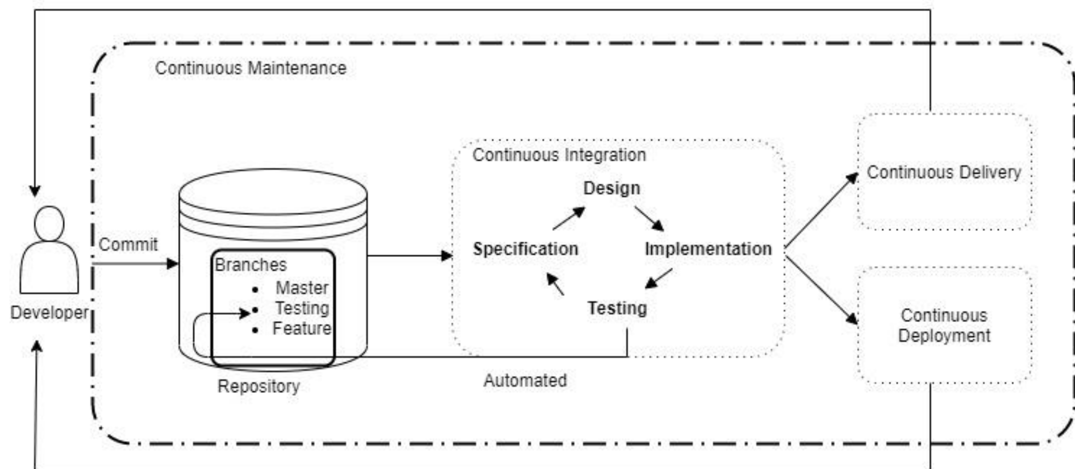


Figure 35: Continuous Maintenance- Continuous Practices and SDLC Relation

Result 4

After the case study, we find out the following problems:

1. Missing Documentation for libraries or projects

2.Impact Analysis

3.Failure Tracking

4.Coordination and Communication Issues

Usage of continuous maintenance tools/practices can minimize these problems. Impact analysis, documentation and failure tracking can be overcome with automated continuous maintenance. In addition, CM can help with the performance and security problems as well.

CHAPTER 7: CONCLUSIONS AND FUTURE WORKS

In this thesis, three research questions are mainly tried to be answered. To begin with, we made a literature review about the thesis such as global-software development, continuous maintenance, continuous practices and software repository. Findings of this review suggested that, there is not enough study on “continuous maintenance”.

We presented a case study of global software development. The case study is mirror the perfective continuous maintenance. To start the case study, we identified the high-level structure and user-level design decisions made. Project provided by a company and it’s a web application project. Analysis of the existing systems revealed how the system works and help us the guide how to proceed for the project. Since the project is a global-software development project, widely known best practices used to meet the necessary functionalities. Case study lasted two weeks.

Thesis aim is to identify the challenges in continuous maintenance and especially repository management. And again, the aim is increased the knowledge on the continuous maintenance. In case study, we try to identify the challenges in a continuous maintenance project. Case study shows us the difficulties in a continuous maintenance project with a global software team. Difficulties are; communication issues, library or environmental changes, impact analysis, failure track, coordination issues. These findings show that continuous maintenance is high stress work. Continuous maintenance activities nested with testing. Every change has to be tested in order to deploy.

To minimize the effects of changes in continuous maintenance such as repository fails, conflicts, and environment changes or even the deployment problems caused by communication or coordination problems; code analyzing tools can be used. Also in repository management perspective, testing branch can contain unit tests defined by testers for every possible case in order to fulfill the requirements to deploy into the master branch. This is basically referring to test automations. Without using test

atomization, continuous maintenance practices cannot be used efficiently. Test atomization is key for minimizing the code effects. Even though there are some continuous integration tools that cover the automated tests, there is no exclusive tool for automating the unit test in version control systems for especially continuous maintenance. The study extends the knowledge of challenges in continuous maintenance projects and suggest the usage of test atomization to minimize the effects.

Even though, IEEE STD: 610 [10] describes the maintenance as “The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment”; however, continuous maintenance activities start on the first day of the software project. Since that, classical maintenance terminologies are not sufficient enough. Every aspect of the project can be changed from day one. Requirement, environment, design or code can be changed in any day. Since, project has to be able to adapt these changes. Adaptation happens with continuous maintenance activities. Agile methodologies also suggest the adaptation to change. In a world of change and software mostly developed with Agile methodologies, continuous maintenance has to be used. The definition of continuous maintenance is the extension of continuous integration, and the continuous integration concerns with SDLC processes. Therefore, in a way by using continuous maintenance in software development, we integrated continuous practices to development life cycles as well. Since continuous maintenance starts with the projects run day, it’s a different process than classical maintenance. Classical maintenance terms are rigid, but they are not covering the modern software development terminology. In continuous maintenance, there is no need to wait for a release or after product period to start maintenance. But again, it also helps to improve the performance of the project as well as improving the security of the project just as maintenance. By adding and using continuous maintenance with SDLC and CP; we can minimize the testing processes cost. Defects and bugs can be found so much earlier and faster than the regular testing processes.

Classical maintenance is not sufficient enough to ensure Agile developments relation with continuous maintenance. And again, terminology differences affect the overall view of points. Because of these reasons, in order to be coherent with IEEE's maintenance definition, we propose new definition for "Continuous Maintenance".

Continuous Maintenance

> The process of modifying a software system or its units and components from the first day of a project, and during development to correct faults, improve performance or other attributes, or adapt to a changed environment or changed requirements under the Agile principles and values.

As a result, the study has achieved the researcher's objectives. The findings and case study clearly show that the aim of this study is realized.

This thesis, which is one of the few studies conducted on software continuous maintenance, is of crucial importance for the coming studies to be carried out in the fields of continuous practices. Any new research and application to be made on this field will make the author of this thesis lines very happy.

Future Works

Many different adaptations, tests, and experiments have been left for the future due to lack of time. Future research on continuous maintenance might focus on the improvements on the automation aspect.

Unit test design and modelling for automated test tools is a new area that can be expand by new researches. Exclusive continuous maintenance tools can be designed with the new researches. Also, the definition of maintenance conflicts with the definition of continuous maintenance. Since that renaming or changing the definition could be a future work.

Version control systems can add new features to adapt continuous maintenance or "Agile maintenance" such as commit limitations. To help with the failure tracking, version control systems can add commit limitations in order to minimize the affect of unnecessary commits.

REFERENCES

Tukey, J.W. (1958) *The Teaching of Concrete Mathematic*s. The American Mathematical Monthly, Vol. 65, no. 1, pp. 1–9.

Bourque P. and Fairley R.E. (Eds.) (2014) *Guide to the Software Engineering Body of Knowledge*. Version 3.0, IEEE Computer Society.

IEEE 12207-2008. (2008) *ISO/IEC/IEEE International Standard - Systems and software engineering -- Software life cycle processes*.

Grubb, P. and Takang, A.A. (2003) *Software Maintenance: Concepts and Practice.* 2nd Edition, World Scientific Publishing Company, Singapore.

Boehm B.W. (1987) *Improving Software Productivity*. IEEE Computer, pp 61-72.

Jones C. (2008) *Applied Software Measurement: Global Analysis of Productivity and Quality.* 3rd Edition.

Halpern M. and Shaw C. (Eds.). (1966) *Annual Review of Automatic Programming.* vol. 5. Elmsford, NY: Pergamon Press, pp. 239-307.

Belady LA, Lehman MM. (1972) *An introduction to program growth dynamics.* Freiburget W. (ed.) Statistical Computer Performance Evaluation, Academic, New York, pp: 503-511.

Swanson E.B. (1976) *The dimensions of maintenance*. . ICSE '76: Proceedings of the 2nd international conference on Software engineering, pp 492-497.

IEEE Std 610.12-1990. (1990) *ISO/IEC/IEEE International Standard, IEEE Standard Glossary of Software Engineering Terminology*, pp 1-84.

IEEE Std 14764-2006. (2006) *ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes – Maintenance*, pp.1-58.

IEEE Std 1219-1998. (1998) *ISO/IEC/IEEE International Standard for Software Maintenance*, pp 1-56.

Sommerville I. (2015) *Software Engineering* 10th ed., Pearson Education.

Stahl D., Martensson T. and Bosch J. (2017) *Continuous practices and devops: beyond the buzz, what does it all mean?* 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp 440-448.

Cusumano M. A. and Selby R. W. (1995) *Microsoft Secrets: How the world's most powerful software company creates technology, shapes markets, and manages people*, New York: Simon and Schuster.

Pang C., and Hindle A. (2016) *Continuous Maintenance. IEEE International Conference on Software Maintenance and Evolution* (ICSME), Raleigh, NC, pp. 458-462.

Shahin M., Ali Babar M., and Zhu L. (2017) *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*. IEEE Access, Vol. 5, pp. 3909-3943.

Caum C. (2013) *Continuous Delivery vs. Continuous Deployment: What's the Diff?* [Online]. Available at: https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff/. (Accessed 12 February 2020)

Perry C. (2017) *Continuous Delivery vs. Continuous Deployment: An Overview, Continuous Delivery vs. Continuous Deployment: An Overview* [Online]. Available at: https://dzone.com/articles/continuous-delivery-vs-continuous-deployment-an-ov. (Accessed 12 February 2020)

Aranda, G. N. (2008) *A Requirement Elicitation Methodology for Global Software Development Teams.* Encyclopedia of Information Science and Technology, Second Edition, M. Khosrow-Pour, IGI Global.

Wiredu G.O. (2020) *Global Software Engineering: Virtualization and Coordination*, Taylor & Francis Group.

Abdel-Hamid T. K. (1993) *Adapting, correcting, and perfecting software estimates: a maintenance metaphor.* Computer, Vol. 26, no. 3, pp. 20-29.

Usha Rani, S. B. (2017) *A detailed study of Software Development Life Cycle (SDLC) Models.* International Journal of Engineering and Computer Science, 6(7). [Online] Available at: http://www.ijecs.in/index.php/ijecs/article/view/2830. (Accessed 12 February 2020)

Cardoso de Mello M. (2012) *Agile processes for the maintenance cycle.* IBM.

Stolberg S. (2009) *Enabling Agile Testing Through Continuous Integration.* Agile Conference, Chicago, IL, pp. 369-374.

Ó Conchúir E., Ågerfalk P., Olsson H. and Fitzgerald B. (2009) *Global Software Development: Where are the Benefits?* Communications of the ACM. 52, pp 127-131.

Begel A. and Nagappan N. (2008) *Global Software Development: Who Does It?* IEEE International Conference on Global Software Engineering, Bangalore, pp. 195-199.

Kumar G. and Bhatia P. (2012) *Impact of Agile Methodology on Software Development Process.* International Journal of Computer Technology and Electronics Engineering (IJCTEE). Vol. 2, pp 2249-6343.

Sundararajan, S., Bhasi, M. and Pramod, K.V. (2017) *Managing Software Risks in Maintenance Projects, from a Vendor Perspective: A Case Study in Global Software Development.* International Journal of Information Technology Project Management. Vol. 8, pp 35-54.

Mens T., Wermelinger M., Ducasse S., Demeyer S., Hirschfeld, R. and Jazayeri, M. (2005) *Challenges in software evolution.* Eighth International Workshop on Principles of Software Evolution (IWPSE'05), Lisbon, Portugal, pp. 13-22

Buinus M. (2015) *Design for maintenance: An interview-based survey,* Master Thesis Work. Malardalen University Sweeden.

Hanssen G.K., Yamashita A., Conradi R. and Moonen L. (2009) *Maintenance and Agile Development: Challenges, Opportunities and Future Directions.* IEEE International Conference on Software Maintenance, Edmonton, AB, pp. 487-490.

Bosch, J. (2014) *Continuous Software Engineering,* Springer.

Fowler M. (2006) *Continuous Integration* [Online]. Available at: https://martinfowler.com/articles/continuousIntegration.html. (Accessed 12 February 2020)

Steidl D., Deissenboeck F., Poehlmann M., Heinke R. and Uhink-Mergenthaler B. (2014) *Continuous Software Quality Control in Practice.* Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME, pp 561-564.

Schermann G., Cito J., Leitner P., Zdun U and Gall H. (2016) *An empirical study on principles and practices of continuous delivery and deployment.* PeerJ PrePrints, Vol.4, e1889v1.

Borner K. and Zhou Y. (2001) *A Software Repository for Education and Research in Information Visualization.* Proceedings Fifth International Conference on Information Visualisation, London, England, UK, pp. 257-262.

Olatunji Sunday O., Idrees S.U, Al-Ghamdi Y.S and Al-Ghamdi J.S.A. (2010) *Mining Software Repositories - A Comparative Analysis.* International Journal of Computer Science and Network Security (IJCSNS) Vol.10 no:8.

Vassallo C., Zampetti F., Romano D., Beller M., Panichella A., Di Penta M. and Zaidman A. (2016) *Continuous Delivery Practices in a Large Financial Organization.* IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, pp. 519-528.

Holck J. and Jørgensen N. (2003) *Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects.* Australasian Journal of Information Systems; Vol. 11, No 1.

Pereira I., Lima L., Amorim V. and Nunes W. (2018) *Implantation of continuous integration practices: An experience report in a software development and research laboratory.* Computer on the Beach 2018.

Roy R., Stark R., Tracht K., Takata S., Mori M. (2016) *Continuous maintenance and the future – Foundations and technological challenges.* CIRP Annals, Vol. 65, Issue 2, pp 667-688.

Cheikhi L., Abran A. and Desharnais J. (2012) *Analysis of the ISBSG software repository from the ISO 9126 view of software product quality.* 38th Annual Conference on IEEE Industrial Electronics Society, Montreal, QC(IECON), pp. 3086-3094.

Raemaekers S., Deursen A. and Visser J. (2013) *The Maven repository dataset of metrics, changes, and dependencies.* 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, pp. 221-224.

Ozbas-Caglayan K., and Dogru A. (2013) *Software Repository Analysis for Investigating Design-Code Compliance.* Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement, Ankara, pp. 231-234.

Reiss S. (2014) *Tool Demo: Browsing Software Repositories.* IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, pp. 589-592.

Tripathi A., Dabral S. and Sureka A. (2015) *University-industry collaboration and Open Source Software (OSS) dataset in Mining Software Repositories (MSR) research.* IEEE 1st International Workshop on Software Analytics (SWAN), Montreal, QC, pp. 39-40.

Higgins D. A. (1988) *Data Structured Maintenance: The Warnier/Orr Approach.* Dorset House Publishing Co. Inc., New York.

G2 (2019) *Best Version Control Systems* [Online] Available at https://www.g2.com/categories/version-control-systems?utf8=%E2%9C%93&order= popular (Accessed 12 February 2020).