1-1-2014

Case Study Of Phased Model For Software Change In A Multiple-Programmer Environment

Yoann Senin
Wayne State University,

CASE STUDY OF PHASED MODEL FOR SOFTWARE CHANGE IN A MULTIPLE-PROGRAMMER ENVIRONMENT

by

YOANN SENIN

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2014

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

DEDICATION

To dad, mom, and Nish.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Václav Rajlich, for his limitless support and guidance throughout my master education. His expertise in the field facilitated both the writing of my thesis and my transit at Wayne State University. I would not forget Leon Wilson whose contribution to this work was priceless. His advice, availability, and formative influence paved this research.

Now, I would like to thank Laurentiu Radu Vanciu at the SoftwarE Visualization and Evaluation REsearch Group (SEVERE) lab for always finding time slots in his busy schedule to teach me some of the tools used in this study and give me thoughtful remarks and suggestions about the research. Also, I would like to thank Christopher Dorman whose previous work inspired the current research and whose subtle advice guided this work. Next, I would like to acknowledge the Department of Computer Science at Wayne State University for giving me the opportunity to learn software engineering skills via its more than talented faculty members. My thanks also go to Nisha Anbazhagan who did not hesitate to proofread this document.

I thank members of my Master Thesis Committee Dr. Marwan Abi-Antoun and Dr. Hamidreza Chitsaz, for taking their time to evaluate this work and offer any valuable comments.

Finally, I acknowledge my family and friends who believed in me and supported me throughout my curriculum.

Any remaining errors in this thesis are mine alone.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

The typical lifespan of software is its creation, its release to the general public, its evolution and maintenance, and the termination of any support on it. From all these stages, the evolution and maintenance both constitute the most important stage of software lifespan [30] since a considerable amount of time and effort are dedicated to them [6, 16]. These two stages of software lifespan consist of a series of software changes. They are the essence of software evolution [21] and they either correct defects, add new functionalities, or modify existing features to software.

Completing software changes generally requires programmers to go through various phases. Among many steps, programmers should initially make sure that the software change request itself is of the right level of abstraction or granularity. In other words, they should check whether or not a software change should be decomposed into smaller changes. Secondly, programmers should resolve inconsistencies. Such inconsistencies could be one of the following: contradiction in the statement of the software changes; inadequacy, where the requirements are too briefly explained; ambiguity in the software change requirements, which makes difficult for programmers to understand what needs to be done; irrelevance of the change requests to software; and unfeasibility due to the technology used for the project, the limited abilities of the team members, or the budget constraints.

Next, programmers should determine the modules or code snippets from the existing source code where the software change should be implemented. Assessing the impact of an application of a software change on designated modules is another phase of the software change process as well as the actual implementation the change in the

source code. A last phase is to make certain that the software change has been correctly implemented and it is in alignment with the requirement.

The software change process has been part of many researches and in most of them, the center of attention has been individual part of the process such as concept location or impact analysis [7, 25]. Concept location is a search based on the software change request terms that identifies the source code fragments that need to be updated, while impact analysis points out other modules that could be affected by changes made on the modules identified in the concept location phase.

One remark about the research mentioned earlier is that even though the software change process is crucial in software lifespan, not many researchers have demonstrated great interest in it. This is the reason why we believe that there should be more research on such an integrated process itself. Such research could aid with improving programmers' productivity and also make them produce a better quality of software faster. Harter and al. present in [18] some software process models that witnessed similar improvements.

In this thesis, we perform an empirical study to do a comparative analysis of programmers completing software changes using Phased Model for Software Change (PMSC) and those completing software changes without any assistance. PMSC is a proposed process that guide programmers in conducting software changes and which is discussed in detail in [29]. We also show that improves performance of both less experienced and more experienced programmers. Our results show that PMSC helps in reducing the time spent to complete software changes.

The rest of this thesis is organized as follow: chapter 2 presents previous related work. It is followed by chapter 3 which explains the motivation for this study and details its design. Next, chapter 4 reveals the results along with the potential threats of validity and the measures taken to mitigate them. Finally, chapter 5 concludes the thesis and suggests future work.

CHAPTER 2: PREVIOUS WORK

2.1. Software Processes

The intent throughout the years of research on software processes, within software engineering, has always been to both help improve the efficiency in which software programmers develop software and to improve the quality of their programs. After software is initially published, there is also usually a team that is dedicated to the evolution or maintenance of the new software until the end of its lifespan. Also, depending on the size of the programming team or the size of software to build, there exist different methods to lead a project successfully to its goal. This section gives an overview of software evolution and then presents some team software processes related to software evolution.

2.1.1. Software Evolution

Software evolution consists of the continuous improvement of initially developed software via a series of software changes. Software changes have been standardized and differentiated by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) in four types as shown in figure 2.1[20]:

- Corrective changes, which fix software defects or malfunctions;
- Preventive changes, which detect and correct dormant bugs existing in the software before they become active;
- Adaptive changes, which adapt software to its changed or changing environment;
- Perfective changes, which add new functionalities to software.

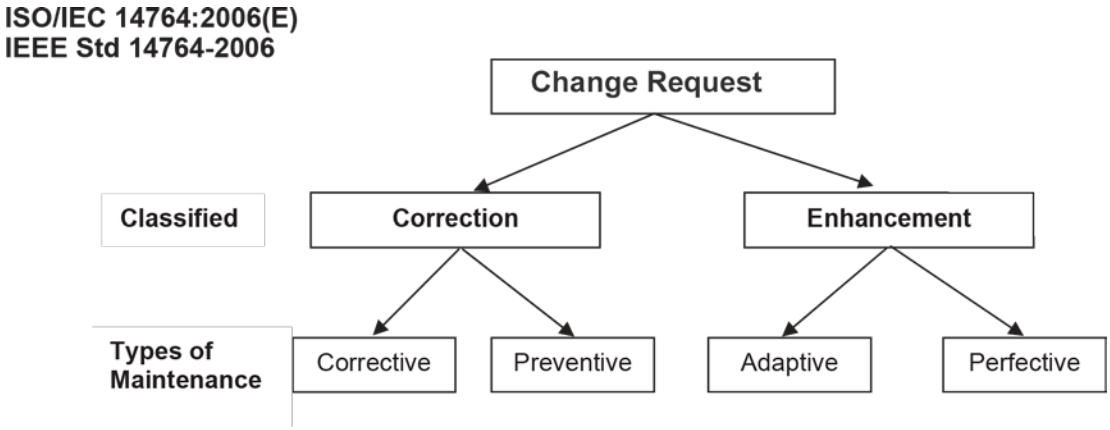


Figure 2.1. Types of software changes

Rajlich [30] considers software evolution as a very important phase of software development since a considerable amount of time is spent on it. Its position in the software lifespan is shown in figure 2.2. Software evolution requires programmers to understand the complexity of software before being able to evolve it. The product of software evolution is the delivery of software releases, which upgrade and/or replace previous versions of the existing software.

It is the responsibility of software managers to determine whether to release software or not, considering the urgency of deadlines or the completeness of the implementation of desired functionalities. To be done efficiently, the evolution of software should follow a process. Section 2.1.2 presents some software processes that assist with software evolution.

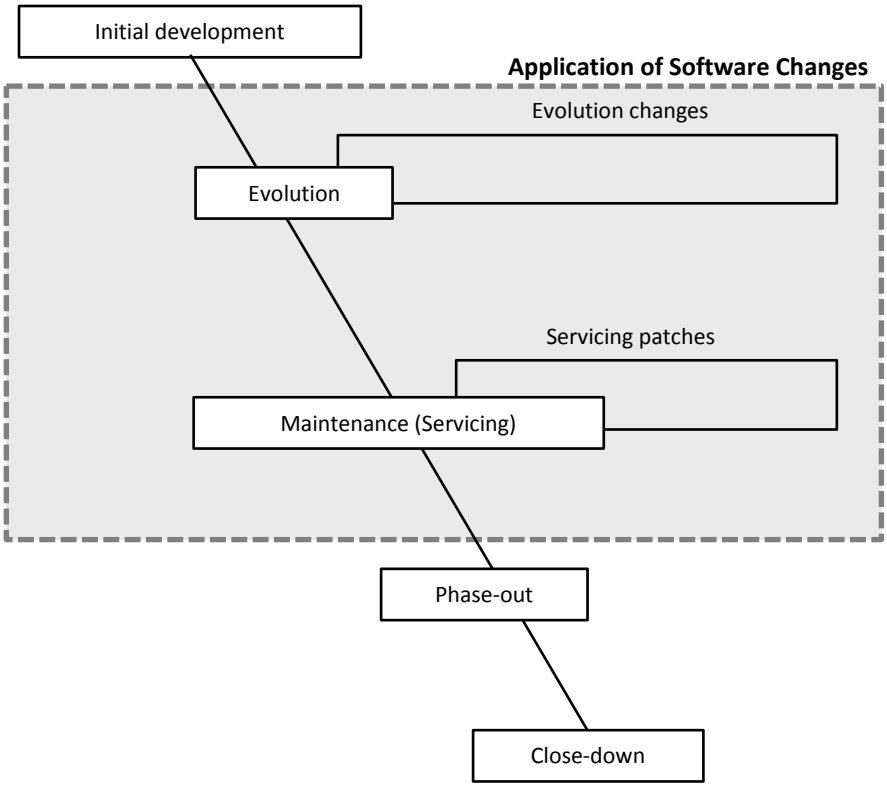


Figure 2.2. Staged model of software lifespan

2.1.2. Team Software Processes

There are various team software processes that support teams of programmers with handling major software development issues such as scheduling tasks, estimating deadlines, or communicating poorly inside the development team. Three of these processes are:

- SCRUM;
- Extreme Programming (XP);
- Team Software Process (TSP).

2.1.2.1 SCRUM

Scrum, a widely used agile process, developed by Ken Schwaber and Jeff Sutherland in the 1990s gets its name from rugby football where 'scrum' is used to refer

the process of restarting the game after a minor violation. After such a violation, players gather to decide on a strategy to play the next phase of the game [17]. In the software engineering world, scrum is a product development strategy that finds efficient alternatives to the dominant and widely followed traditional Waterfall sequential approach. Scrum is designed to be a flexible and iterative software development framework.

The Scrum process itself has three crucial types of meetings: sprint planning meetings, daily scrum meetings and end meetings. The sprint planning meeting is held every seven to thirty days at the beginning of the sprint. It should be noted that a sprint, also identified as an iteration, is a period of time where specific tasks need to be done. The backlog, which is the list of requirements for the project, is prepared during this meeting and the team members decide the tasks they will work on. The daily meeting is held every day for team members to review updates. The end meetings refer to the two team meetings held during the end phase of the scrum process called the 'sprint review meeting' and the 'sprint retrospective'. The purpose of the sprint review meeting is to review the work that has been completed as planned, while the sprint retrospective meeting reflects on the improvements that could have been made in the process [33]. Additionally, a thirty days sprint duration for a single sprint is recommended by scrum. During that interval of time, changes to the plan are not allowed.

2.1.2.2. Extreme Programming

Extreme programming (XP) is another agile process containing 12 key practices [2, 28]. A few of the practices resemble the ones used in Solo Iterative Process (SIP)

and Agile Iterative Process (AIP) explained in more details in section 2.2.2.; some others are unique to XP and generally used in an extreme way.

These XP practices are the following [30]:

1. The Planning Game
2. Simple Design
3. Small Releases
4. Metaphor
5. Pair Programming
6. Immediate Testing
7. Immediate Refactoring
8. Collective Ownership
9. Continuous Integration
10. On-Site Customer
11. Coding Standards
12. 40-hour Week

XP is a successful agile process that is used by various companies of all sizes across the world. XP goes a step further than the scrum by ensuring customer satisfaction amidst very volatile changing requirements. Extreme programming assumes a working environment where everyone is an equal member of the team. The software project is developed only by effective and constant communication with the rest of the team and the customers by feedback for every product testing, and by a simple product design.

Extreme Programming does not have complex rules [14]. The flowchart in figure 2.3 depicts the rules for extreme programming and how the rules work together to accomplish a successful and effective working environment. Customers are embedded in the development process and communication between team members is highly integrated [15]. XP also recommends software development iterations of one week.

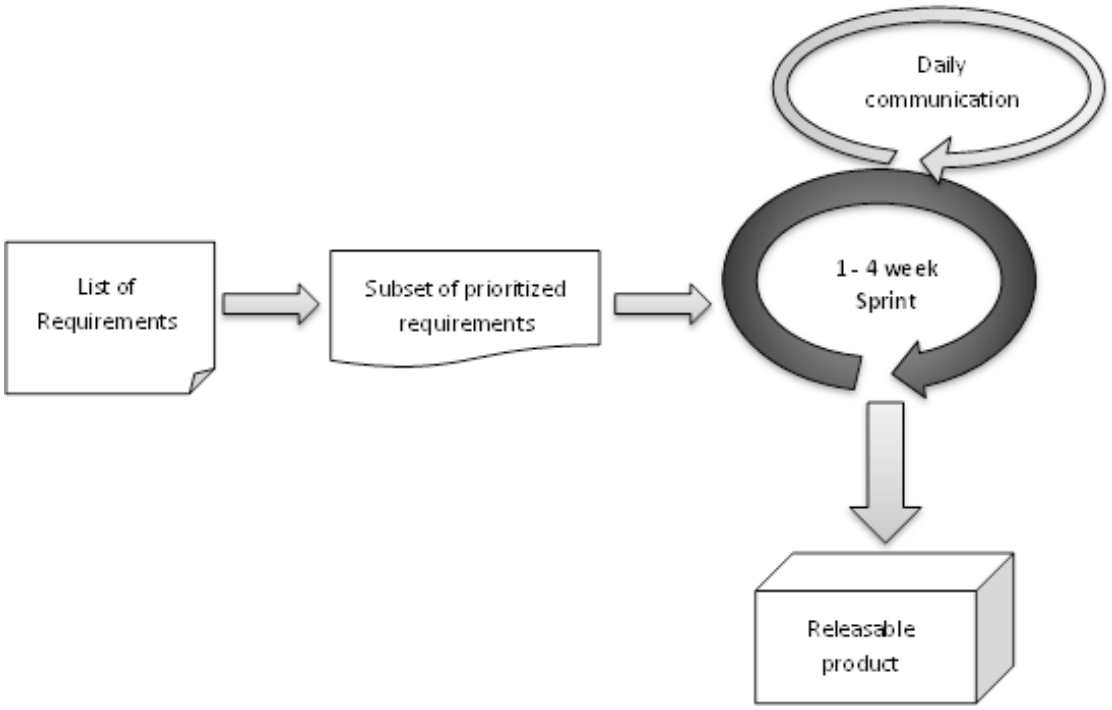


Figure 2.3. XP cycle

In this process, a list of requirements is created based on the needs of the customers. These requirements are then prioritized and only a subset of these prioritized requirements is part of the next iteration. The selection is done during an iterative planning meeting. In that meeting, the development team settles on the amount of work that could be completed by the end of the next iteration, acknowledging feedback from previous iterations if any. Next, the sprint of one to four weeks long starts without possibility to change neither the end date nor the features or stories to deliver. Every single day of the sprint, the stakeholders of the project gather for about 15 minutes to talk about the progress of the project. In that meeting, the team members, one at the time, tell what they have done, what their task for the day is, and what challenges they have encountered. By the end of the iteration, the product should be tested and ready for a release.

2.1.2.3. Team Software Process

Team Software Process (TSP) is one of the many directed processes available. A directed process is a process where the team using that process can manage itself by planning and monitoring its work. TSP teams are composed of software engineers trained in Personal Software Process (PSP). Dan Van Duine [11] defines PSP as “a set of practices that engineers can apply to most structured personal tasks to improve predictability, quality, and productivity.” In other words, it is a structured software development approach that helps to enhance single developers’ skills.

Along with PSP, TSP can be used to establish a working process that facilitates a project team to deliver software products. The product size can range from small to large. The Team Software Process can help the team of software engineers and Managers to deliver a quality products irrespective of the size of the project [23].

The development cycle of TSP starts with a TSP trained person planning the process for the project. This is called the 'Launch' phase. During this initial phase, team members and the project manager define goals and assess the risks, and also produce a team plan and assign tasks. During the implementation of the process, the team meets to give status reports and revise the plan on a regular basis. At the end of the development cycle performance is measured, and ways to improve the process are discussed. TSP thus help to form a software development environment that enables the heightening of a team's productivity [19].

2.1.3. Iterative processes

Iterative software processes are processes that continually rework software. Rajlich [30] presents four of these processes: Solo Iterative Process (SIP), Agile Iterative Process (AIP), Directed Iterative Process (DIP), and Centralized Iterative Process (CIP).

2.1.3.1. Solo Iterative Process

SIP is a software process that involves only one programmer. In this process, the programmer defines a product backlog, which is the list of requirements for a project, and then creates or updates the code of the software depending on the priority of the requirements in the product backlog. SIP is described in Figure 2.4.

The programmer, represented as “Solo” in the figure below, receives requirements from users and generates a product backlog with these requirements. After analysis of the requirements, the programmer prioritizes the requirements and turns the ones with the highest priority into the iteration backlog. The next step for the programmer is to select change requests from the iteration backlog and implements them in the code. During the implementation, the programmer builds new baselines and run system tests. Once the selected change requests are implemented, a new software version is released to the users.

Throughout the process, the programmer keeps time logs and defects logs and use them to estimate future tasks, baselines, and releases.

Most of software development projects need more efforts than a single programmer can handle. This is why there exist team processes that divide the tasks to

do and allocate them to several persons. The next three sub-sections are examples of team iterative processes.

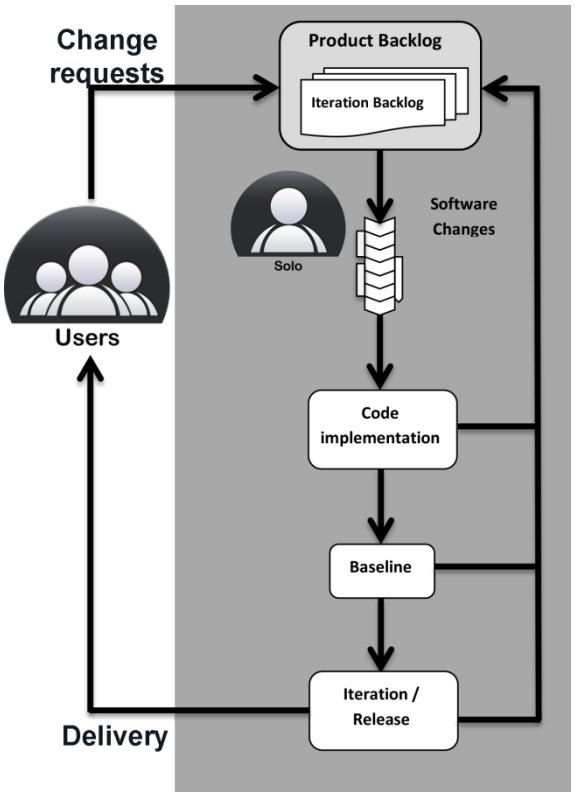


Figure 2.4. SIP Model

2.1.3.2. Agile Iterative Process

AIP is a team iterative process where most of the decisions are made by consensus whether it is to assign tasks or to solve problems. The programming team in AIP counts approximately five to ten persons and the team members do not need any specialized skills; the diversity of programming skills makes the tasks allocation very easy.

AIP has two types of managers: the product manager and the process manager. The product manager focuses more on the development of software. He supervises the

business decisions, controls the change requests, checks the programmers' work, and decides on the way to release a product. The process manager on the other hand makes sure that the AIP process is properly followed. He ensures the functionality and the productivity of the team and protects them from external interferences.

In Figure 2.5 which represents the AIP model, the product manager creates a product backlog based on the users' requirements and then generates an iteration backlog. The programmers, after discussing and allocating tasks from the iteration backlog, simultaneously make their software changes. In a daily loop, a new baseline is created via the build process generally overnight. A daily meeting then takes place where the programmers discuss the results from the last build, the progress of their assignment, and the challenges they encounters. The daily meeting lasts roughly 15 minutes. During the meeting, the product manager helps resolve business related issues of the project, while the process manager makes sure that the meeting is short and professional.

A software development iteration in AIP lasts about one to four weeks, with a common duration of two weeks. An iteration ends with an iteration meeting, where the programmers, managers, and users participate. The first part of the meeting is the iteration review, where the stakeholders listed above assess the current version of the product and the expected version. The second part of the iteration meeting is to plan the next iteration based on what happened in the previous iterations. The release of a new version of the product to the users is also discussed during the meeting.

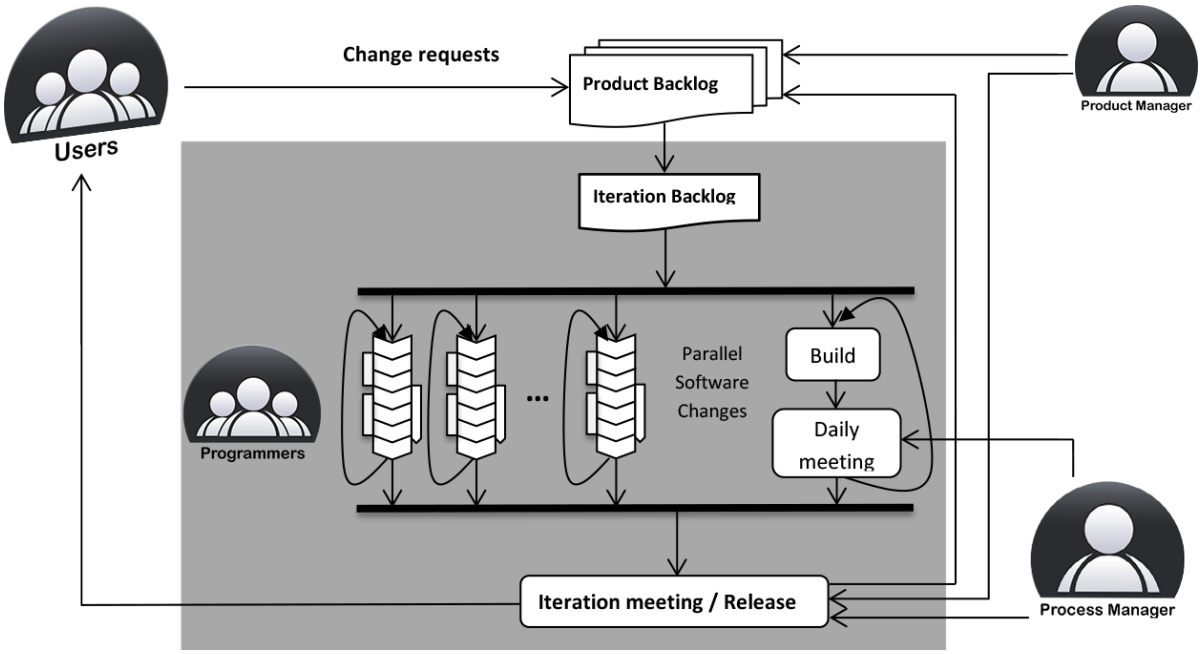


Figure 2.5. AIP model

2.1.3.3. Directed Iterative Process

DIP is a team iterative process where the process managers make the decisions, the planning, and the allocation of tasks. Unlike AIP, the programming team members have specializations. In fact, there is a group of developers, who produce code, and a group of testers, who validate the developers' commits, test and certify baselines. The product manager in DIP has the same role as the one in AIP: understand the software and its position in the market. There could be more than one product manager in large projects. As Rajlich states, "the process managers enact, monitor, and plan DIP." They allocate tasks to the developers and testers and make sure that the programming team works without any interference.

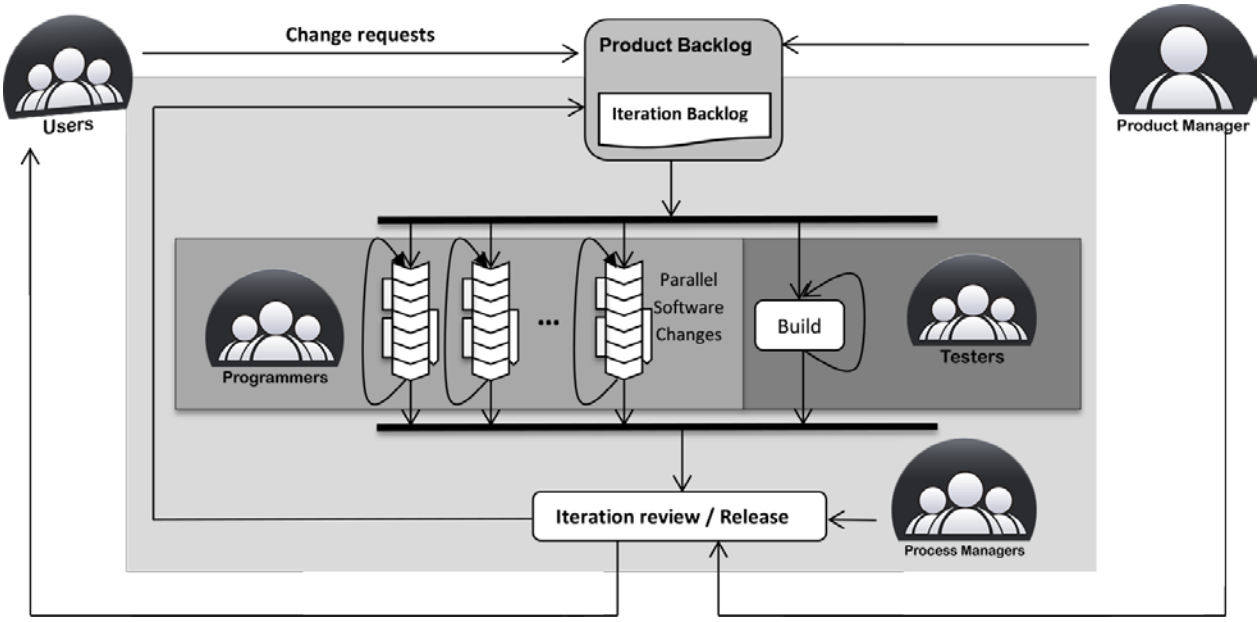


Figure 2.6. DIP model

In the DIP model depicted in Figure 2.6., both the product and process managers work together to produce the iteration backlog at the beginning of the process. The process managers then assign change requests from the iteration backlog to the developers, who implement their changes in parallel and submit them to the version control system. In the build loop, the testers run daily system tests on the code and generate new baselines.

Communication is a key element in DIP as accurate and regular feedback from the development team eases the decision-making of managers.

At the end of the iteration, an iteration review informs the stakeholders about the current state of the project. The length of iterations varies from one to six months.

2.1.3.4. Centralized Iterative Process

CIP, shown in Figure 2.7., is an iterative process highly recommended for teams with a very diverse set of skills or teams where a high level of quality is expected. An example of place where CIP is used is an open source community that is composed of

volunteers. Another field where CIP is employed is avionics. Software developers in that field are highly skilled because human life is at stake. This is a reason why the commits of these programmers are rigorously checked by code guardians before being accepted in the repository of the version control system.

Code guardians are in general software architects, quality managers, code owners, and so on. They all inspect and validate the programmers' commits to protect the quality of the code in the version control system. Especially, "architects guarantee that the program architecture will be preserved through the evolution; code owners guarantee the quality of the commits in the parts of the code they own; and quality managers guarantee the general quality of the commits."

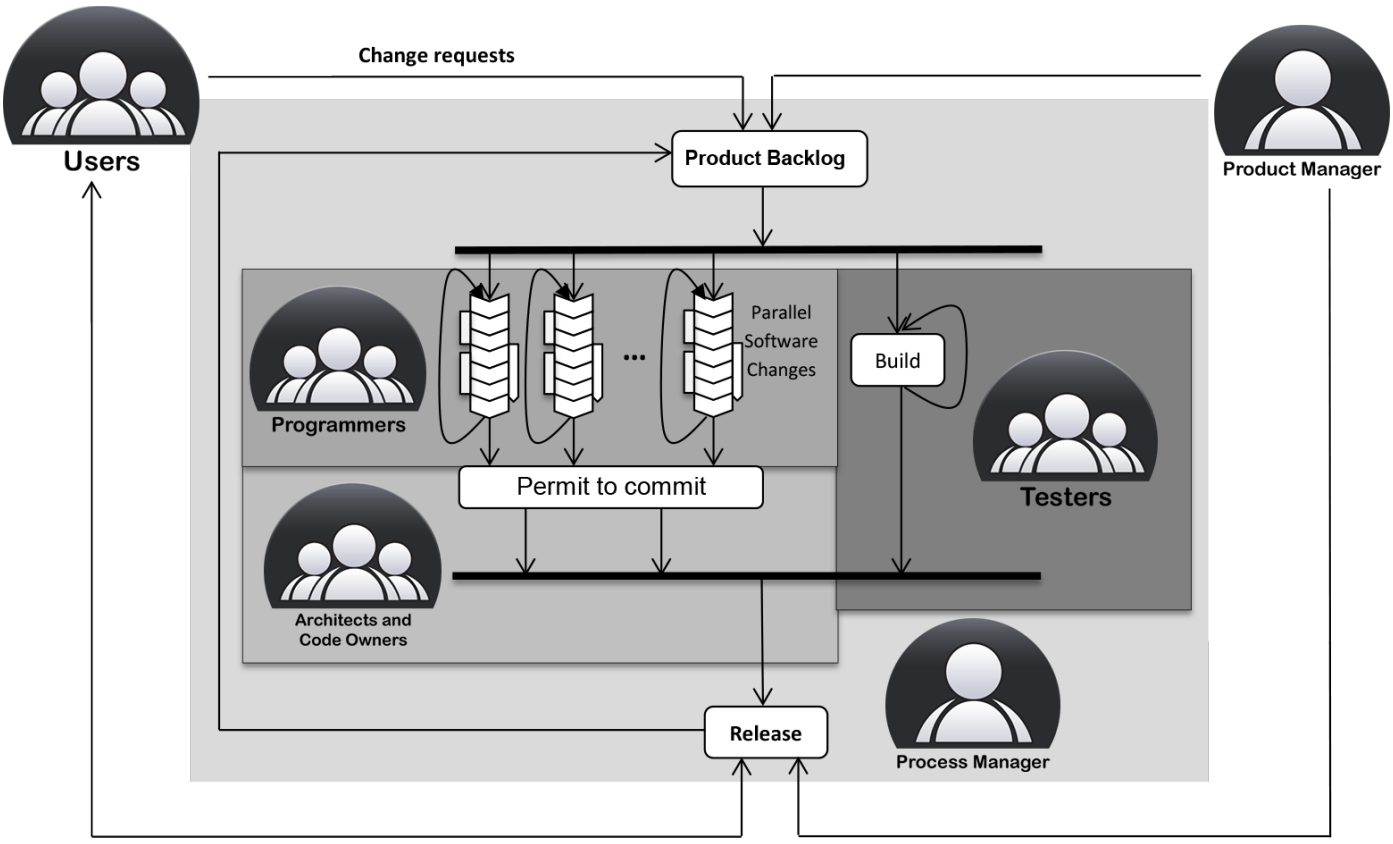


Figure 2.7. CIP model

2.2. Phased Model of Software Change

In his book Software Engineering: The Current Practice [30], Vaclav Rajlich talks about the way software changes are managed in projects. He introduces an approach that guides software engineers in modifying software: the Phased Model of Software Changes (PMSC). This section describes the different phases that compose PMSC. It then outlines processes where PMSC is used and finally present some work that used both PMSC and one of the iterative processes mentioned earlier.

2.2.1. Phases of Software Changes

PMSC is a software development process that assists programmers in applying software changes, where each step of the process is a phase. Figure 2.8 shows an overview of the whole process.

At the beginning of the process, programmers prioritize and select the change request to implement. This is the *initiation*. This phase is followed by *Concept Location* (CL), which identifies in the software the module or the piece of code that needs to be updated whether to correct a defect or to add a new functionality. It happens sometimes that concepts are scattered within the code. It is in this circumstance that *impact analysis* becomes useful. In addition to the modules identified by CL as the potential areas to make changes in the code, *impact analysis* points out other modules related to the modules to modify and it also determines the impact of changes on these related modules.

So far, the steps stated constitute the design of software change. They happen just before the phases where the actual changes of the code are made. These phases

in which the actual change is made are the actualization, the refactoring, and the verification.

Actualization is the phase where modifications in the code are implemented. These modifications could affect some other parts of the code. It is the reason why, similarly to impact analysis, change propagation identifies the parts affected by the early changes in the code. The difference between impact analysis and change propagation is that modifications are actually made in change propagation.

Another phase in PMSC is *refactoring*. It consists of changing the structure of the code without changing any functionality. It is called *prefactoring* when it happens before *actualization* and *postfactoring* when it happens after. *Prefactoring* restructures the old code to make *actualization* easier, while *postfactoring* cleans up any mess that could have occurred during actualization.

The next phase is called *verification*. This phase reduces bugs and any other problems that may exist in the code at the time of the prefactoring, the actualization, the postfactoring and the conclusion.

The last phase of software change is the *conclusion*. During this phase, programmers commit their final version of the code into a version control system. They can also create a new baseline along with an updated documentation and other materials useful to the development of software.

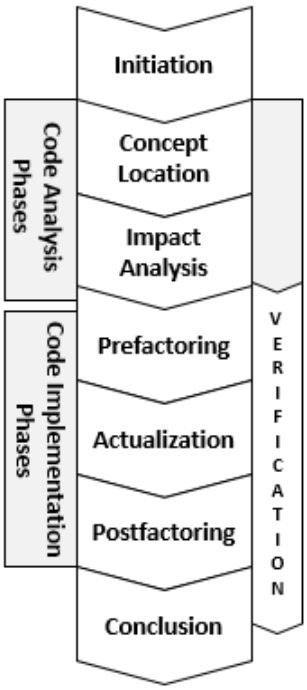


Figure 2.8. Phased Model for Software Change

2.2.2. Previous work using PMSC

In an experience report using SIP [8], Christopher Dorman successfully implements software changes on a medium sized open source tool despite his limited experience in Java programming and his newness in the selected project. At the end of his case study as a part of future work, he recommends the enactment of the other iterative processes; AIP for small teams and CIP and DIP on large teams.

As opposed to Dorman's work, which concentrates primarily on SIP and a single programmer, the experiment in this document addresses the use of PMSC among small programming teams; it extends the use of SIP to a multiple programmer environment. To do so a two phase user study was conducted to capture small team programmers' experience with PMSC. In the first phases, the groups of students are assigned change requests to apply on a specific Java-based application without knowledge of any PMSC procedures. In the following phase, the students are taught PMSC techniques and are

asked to make change requests on other Java-based applications following PMSC. Furthermore, each group works on a different application at each phase.

2.3 Software Process Tools

As Dorman mentions in his experience report [8], software evolution is not only limited to a set of process steps to follow, but it also needs tools to assist the programmers in the processes. Two types of tools could be distinguished: the ones embedded in the Interactive Development Environment (IDE) and the other ones, which are stand-alone applications. The IDE used in this study is Eclipse Classic 4.2.2.

2.3.1 PMSC-based Technologies

2.3.1.1. JRipples

JRipples is an Eclipse plug-in designed to assist java developers in making their software changes easier. Buckner, Buchta, Petrenko, and Rajlich [4] created a tool that helps to keep track of dependencies in a program and that guide programmers in what their following step could be. At the same time, JRipples reduces the risk of errors that would occur if that tracking is done manually. The tool focuses mostly on the concept location, impact analysis, and change propagation aspects of PMSC.

2.3.1.2. JUnit

JUnit is a framework to write repeatable tests in Java programming [3]. It was created by Kent Beck and Erich Gamma. It could be downloaded separately at junit.org or be downloaded as part of the Eclipse IDE.

2.3.2. Other Supporting Tools

In addition to the required tools listed above, the students are given a set of tools that assist them in their work. Because these tools are not mandatory to PMSC, any

other similar tools could be used in similar research. The supporting tools used for this case study are the following.

2.3.2.1. Rabbit

Rabbit is a plug-in used in Eclipse to track time. It is a tool that runs anonymously in the background and records the time spent by the user in Eclipse. The information gathered by Rabbit can be retrieved when the user requires it, via a view designed for that purpose [13].

2.3.2.2. EclEmma

EclEmma is an Eclipse plug-in used to get the code coverage for a java program. In other words, it helps to see how much Java code has been executed during the execution of the code [12].

2.3.2.3. Abbot Java GUI Test Framework

Abbot is a testing framework for both functional and unit testing of Java Graphical User Interfaces. It helps to generate user actions and to test the state of the components without any human interaction with the source code being tested [36, 37].

2.3.2.4. Subversion & TortoiseSVN

Subversion is an open-source version control system from Apache. It saves different versions of files for developers [1].

TortoiseSVN is a Windows subversion client, which helps to manage different versions of files or any other documents saved in subversion [34, 35].

2.3.2.5. DiffStats

DiffStats is a tool created by Christopher Dorman to count the number of lines added, deleted, or moved in Java source code. It also works with C++ code [9].

CHAPTER 3: CASE STUDY

This chapter explains the motivation and goal of the experiment and then details how the study was performed from start to end.

3.1. Motivation and Goal

In previous research conducted by Chris Dorman in [10], the experience of a solo programmer is captured and analyzed using the PMSC and Solo Iterative Process (SIP). The motivation for the study reported in this thesis is to extend that research by observing the performance of multiple developer environments when applying software changes using the PMSC approach. Henceforth, a user study is conducted using graduate students (both Masters and PhD) enrolled in a graduate software engineering course. These students conducted software changes in a desktop application development environment.

3.1.1. Desktop Application Development

The teams are tasked with performing software changes on three different Java-based applications. The first set of software changes is performed without the guidance of the PMSC technique or assistance of any specified tools beyond the use of the IDE; thus constituting a pre-test performance baseline. The other set of software changes are performed respectively on two different applications following the PMSC technique, and selected tools are adopted to support the PMSC phases of work.

During the software change efforts, programmers within each team record their experience and their results in software change logs. A post analysis is then conducted along with any follow-up inquiries to ascertain any observations and

findings. Additionally, the software change logs are used to collect quantitative data similar to data collected by Dorman in his earlier research.

3.2. User Study Design

This study follows the recommended guidelines for empirical research in software engineering [5, 24].

Six hypotheses have been formulated for this experiment. The first hypothesis is that programmers complete their change requests faster when they use PMSC, whether they are less experienced or more experienced. The second one is that they take less time to analyze their code with PMSC. The third hypothesis is that the programmers implement their code faster when they follow PMSC. The next hypothesis suggest that PMSC helps experienced programmers in completing their change requests faster, while the fifth hypothesis suspects that PMSC accelerates the code analysis of experienced programmers working on their change requests. The last hypothesis is that experienced programmers do not need much time to implement their code when they follow PMSC. These hypotheses are stated as the following alternative hypothesis:

H1: *PSMC shortens the completion of change requests*. That is, there is a significant difference between programmers using PSMC and those not using PSMC.

H2: *Programmers require less time during code analysis following PMSC*. Specifically, there is a significant difference in the time spent by programmers performing code analysis using PMSC and those not using PMSC.

H3: *Programmers require less time during code implementation following PMSC*. Specifically, there is a significant difference in the time spent by programmers performing code implementation using PMSC and those not using PMSC.

H4: *PMSC shortens the completion time of software changes done by experienced programmers*. Specifically, there is a significant difference between experienced programmers using PSMC and those not using PSMC.

H5: *PMSC shortens the code analysis portion of the time of software changes done by experienced programmers*. Specifically, there is a significant difference in the time spent by experienced programmers performing code analysis using PMSC and those not using PMSC.

H6: *Experienced programmers require less time during code implementation following PMSC*. Specifically, there is a significant difference in the time spent by experienced programmers performing code implementation using PMSC and those not using PMSC.

A “Before versus After” type of experiment, also known as a “within-subject” experiment design, is conducted to measure how programmers perform when they apply change requests without using PMSC and how they perform with the assistance of PMSC. The “Before” portion of the design or pre-test serves as a baseline performance for each individual in the study. Afterward, PMSC is introduced in the post-test. This within-subject design is used because it provides a “higher degree of experimental control” [31].

For this study, the systems assigned to the groups are interchange at each stage. Practically, two systems are initially identified in order to eliminate the learning effect, which is noticed when participants’ performance improves when they do the same task repeatedly. As shown in table 3.1, the participants are separated in two groups, where Group 1 is assigned tasks from System A and Group 2 is assigned tasks

from System B for the pre-test (i.e. stage 1). In a training stage, PMSC is introduced to the groups along with a set of supporting tools. This part of the experiment is a transition stage that serves as training to get the participants familiar with the new process and the new tools that will be used in the next stage. Thereafter, it is the second stage, where the students are assigned a system C and another set of changes to complete.

	Stage 1	**Stage 2**
Group 1	**System A (jAdvisor)**	**System C (JabRef)**
subject #2	change request 4	change request 2
subject #5	change request 3	change request 1
subject #6	change request 3	change request 3
subject #8	change request 2	change request 4
subject #10	change request 2	change request 2
subject #12	change request 5	change request 1
Group 2	**System B (jEdit)**	**System C (JabRef)**
subject #1	change request 1	change request 2
subject #3	change request 4	change request 4
subject #4	change request 5	change request 3
subject #7	change request 2	change request 4
subject #9	change request 1	change request 2
subject #11	change request 4	change request 1

Table 3.1. Case Study Design

Furthermore, we have considered the matter of general repetition, also related to the learning effect issue. The application of this technique to the study is the random assignment of the system in each stage and also the random distribution of the change requests to the participants. This way, each student has a different set of change requests experience across the stages.

This study design is very similar to the one used in [38].

3.2.1. Objects of the User Study

This section presents the applications used for the study. To reduce the potential of programmer learning between stages, different applications are used for software changes. Therefore, the following open-source candidate applications shown in table 3.2 are used.

Program	Version number	Lines of Code (KLOC)	Number of packages	Number of classes	Number of methods	Number of files
jAdvisor	0.4.6	4	4	34	353	34
jEdit	4.3 pre 9	100	42	850	5375	517
JabRef	2.6	78	56	835	4265	577

Table 3.2. Case Study Applications Metrics

3.2.1.1. jEdit

jEdit (http://jedit.sourceforge.net/) is an open-source text editor intended for programmers. It is a user-friendly tool written in Java which could be customized with a large variety of plugins. Some of its features are "Kill ring" which automatically remembers previously deleted text, side by side windows, intelligent bracket matching, and auto indenting [22]. The size of jEdit is about 100 KLOC.

3.2.1.2. jAdvisor

jAdvisor (http://jadvisor.sourceforge.net/) is a program that schedules classes, plan courses, and search courses. It is designed for college students and it allows them to graphically see and improve their schedule. jAdvisor could be personalized to a specific school via adapters.

jAdvisor is written in Java and it counts 34 source code files grouped in 4 folders and the size of project is about 4 KLOC spread over 34 classes [27].

3.2.1.3. JabRef

JabRef (http://jabref.sourceforge.net/), which stands for Java, Alver, Batada, Reference, is an open-source program that manages bibliographical references. It is a cross-platform tool written in Java and whose native file format is BibTeX. BibTeX is a popular file format used to store bibliography. Some of its features are advanced BibTeX editor, search of pattern in whole bibliography, import of various formats, and automatic key generation. JabRef counts approximately 78 KLOC.

3.2.2. Subjects

This study is conducted on both Master and Ph.D. students taking a graduate software engineering course during the fall semester 2013, where the phased model for software changes is taught. However, we made sure that the knowledge of PMSC techniques was not transmitted to the students before the suitable time in the study. The use of students as subjects for this study is appropriate since it has been proved that “there are only minor differences between the conception of students and professionals in certain software engineering circumstances.” [26]

The students are divided in small groups in order to make software changes on the desktop applications listed above, with one application per stage. At the beginning of each stage, the students are taught the manipulation of the supporting tools to use for that particular stage. The next section explains how the division in groups was made.

3.2.3. Division into teams

Before their assignment in groups, the students took a pre-study survey. The survey allowed us to discover the students’ years of experience in object oriented programming languages such as Java and C++ and also

to know their familiarity with the candidate supporting tools to use when working on their change requests. Afterward, two groups were formed and the students were randomly assigned to these groups. It should be noted that being part of a particular group did not matter much because the results of the pre-study survey allowed us to differentiate the less experienced programmers and the more experienced programmers. The average number of programming years between the participants was approximately 2.7 years, with the years ranging from one to seven years. Moreover, none of the students had any programming experience with the systems selected for the study. The graph in figure 3.1 shows the overall number of programming years and the number of Java programming years for each of the students participating to the experiment.

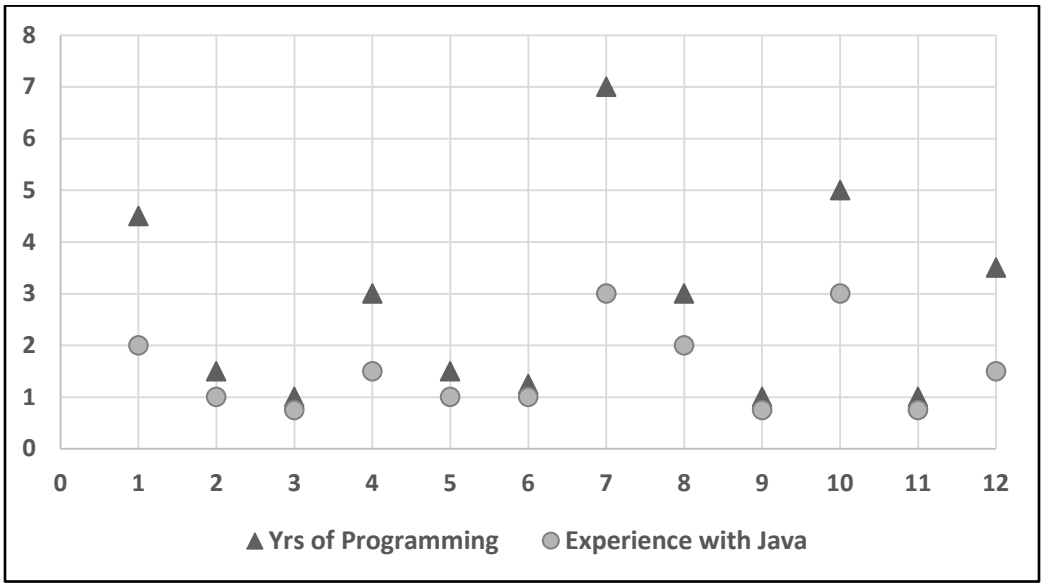


Figure 3.1. Programmer Experience

3.2.4. Data Collection

The data collection is essential to the user study to build upon Dorman's prior research. Therefore, from a quantitative perspective the same data is collected from the

teams based on Table 2 of Dorman's paper [8]. The participants are provided with log templates to capture both their quantitative and qualitative data. After each stage, follow-up inquiries are performed to clarify any of the data collected. Finally, a post-study questionnaire is given to the students to understand their learning experience during the study.

CHAPTER 4: RESULTS AND INTERPRETATION

In this chapter, we analyze the collected data from a statistical point of view in order to justify the hypotheses mentioned in the previous chapter. We also interpret the results. The first section of this chapter explains how the analysis of the data is accomplished. The next section talks about the efficiency of PMSC, while the last section of the chapter presents how the supporting tools helped participants of the study.

4.1. Statistical Analysis

The population used for this study is composed of programmers with various levels of expertise. The feedback from the pre-study survey permitted the classification of the participants in groups of “more experienced programmers” and “less experienced programmers”. Section 4.6 gives more details about the two groups.

For our statistical analysis, we have followed a set of steps in order to verify the hypotheses of the study. First, we determined the difference between the total amount of time spent to complete a change request without the assistance of PMSC and compared it to the total amount of time spent to complete a change request with the assistance of PMSC. Next, a set of normality tests is run to observe whether the collected data follow a normal distribution or not. The existence of a normal distribution would suggest the use of parametric tests as opposed to the use of non-parametric tests when there is no normal distribution of the collected data.

The normality test selected is the Anderson-Darling test and the p-value resulting from the collected data presented in table 4.1 is p=0.283. In the Anderson-Darling test, if the p-value is greater than $\alpha = 0.022$ and this means that the data do not follow normal

distribution. A consequence for having this distribution is to use a non-parametric test to prove the hypotheses. Thus, the Wilcoxon matched pair signed rank non-parametric test is used to justify our hypotheses.

Moreover, in the proof of our hypotheses, we compare a null hypothesis to each of our alternate hypotheses listed in section 3.2. An alternate hypothesis represents what we plan on proving and it is noted H_a: μ_a or H_i: μ_i where i represents the index of a specific hypothesis and μ_i is the median of the population. Conversely, the null hypothesis designates the opposite of the hypothesis to prove and it is noted H_0: μ_0, with $\mu_0 = 0$.

4.2. PMSC Completion Results

The experiment assesses programmer's performance before learning PMSC and after learning PMSC. In other words, the programmers are given a first set of change requests, which they complete based on their own methodology. Meanwhile, they record the amount of time they took in the completion of those change requests (i.e. stage 1 in Table 4.1). Next, the PMSC approach is presented to the participants and another set of change requests is assigned to them. They also record the average amount of time spent on those changes requests while following PMSC (i.e. stage 2 in Table 4.1). The size and complexity of the system used in the second stage of the study ensured that any "learning effort" was minimized.

Two of our hypotheses are related to the time spent to complete software change requests following PMSC: hypothesis H_1 that suggest that PMSC shortens the completion of software change requests for all programmers and hypothesis H_4 that states that PMSC shortens the completion time of software changes done by

experienced programmers. As mentioned earlier in this section, the first step for our statistical analysis on PMSC completion is to determine the difference of time spent to complete software changes with and without PMSC. The other step is to apply the Wilcoxon matched pair signed rank non-parametric test to justify the hypotheses H_1 and H_4.

Phased Software Change Model User Study Data						
	Code Analysis Stage 1	Code Implement Stage 1	Stage 1 Total	Code Analysis Stage 2	Code Implement Stage 2	Stage 2 Total
subject #1	180	1080	1260	40	330	370
subject #2	731	337	1068	300	270	570
subject #3	165	250	415	30	150	180
subject #4	480	150	630	300	150	450
subject #5	840	660	1500	150	120	270
subject #6	360	240	600	420	70	490
subject #7	380	170	550	107	56	163
subject #8	720	420	1140	20	50	70
subject #9	310	125	435	165	75	240
subject #10	175	278	453	121	448	569
subject #11	90	82	172	150	480	630
subject #12	210	360	570	140	270	410
median	335	264	585	145	150	390
std. dev.	242.63	267.10	388.94	116.41	145.31	174.50
average	386.75	346.00	732.75	161.92	205.75	367.67
time reported in minutes						

Table 4.1. PMSC Study Data for All Programmers

A statistical analysis is undertaken to prove the hypotheses listed in section 3.2. To justify hypothesis H_1 that suggests that PSMC shortens the completion of change requests, a measurement of the difference between the amount of time spent by the participants completing the change requests without PMSC and the amount of time

taken to complete the change requests while using PMSC is necessary. The null hypothesis H_0: $\mu_0 = 0$ (median = 0), which infers that PMSC does not assist programmers in the completion of a change request, is originally considered. It is then compared to hypothesis H_1: μ_1. More precisely, the comparison consists of the differentiation of the time spent to complete a change request with the assistance of PMSC and the time spent to complete a change request without the assistance of PMSC.

The results from table 4.1 show that the programmers completed their software changes in stage 2 faster than they did in stage 1. More precisely, without assistance of PMSC they took on average 6 hours 27 minutes (387 minutes) to perform code analysis and 5 hours 46 minutes (346 minutes) for the coding, thus totaling an average of 12 hours 13 minutes (733 minutes). The median is 9 hours 45 minutes (585 minutes) and the standard deviation (std. dev.) is 6 hours 29 minutes (389 minutes). On the other hand, when following PMSC, programmers took on average 2 hours 42 minutes (162 minutes) to perform code analysis and 3 hours 26 minutes (206 minutes) for the actualization, totaling an average of 6 hours 08 minutes (368 minutes) to complete the change request with the assistance of the PMSC approach. The median is 6 hours 30 minutes (390 minutes) and the standard deviation (std. dev.) is 2 hours 54 minutes (174 minutes). This constitutes a 49.8% overall improvement or 6 hours 05 minutes (365 minutes) reduction in time, consisting of 58.1% improvement in code analysis and 40.5% improvement in code implementation. The significant reduction in the standard deviation between stage 1 and stage 2 might show that PMSC helps programmers to perform better.

When the results are narrowed to experienced programmers only as shown in table 4.2, one could notice that PMSC has helped experienced programmers to reduce the time they spent to complete their change request. In fact, without the assistance of the PMSC approach, it took them approximately 357 minutes to perform code analysis and 410 minutes to implement their code during the first stage, thus totaling an average of 767 minutes to complete the change request. The median is 600 minutes and the standard deviation (std. dev.) is 312 minutes. On the other hand, experienced programmers spent 121 minutes to analyze their code and 217 minutes in the actualization phase with the assistance of PMSC, totaling on average 338 minutes to complete their change requests. The median is 390 minutes and the standard deviation (std. dev.) is 171 minutes. This constitutes a 44.1% overall improvement or 429 minutes reduction in time, consisting of 66.1% improvement in code analysis and 47.1% improvement in code implementation. Once again, the remarkable difference in the standard deviation between the two stages may prove a consistent performance among experienced programmers following PMSC.

Phased Software Change Model User Study Data - More Experienced Programmers						
	Code Analysis Stage 1	Code Implement Stage 1	Stage 1 Total	Code Analysis Stage 2	Code Implement Stage 2	Stage 2 Total
subject #1	180	1080	1260	40	330	370
subject #4	480	150	630	300	150	450
subject #7	380	170	550	107	56	163
subject #8	720	420	1140	20	50	70
subject #10	175	278	453	121	448	569
subject #12	210	360	570	140	270	410
median	295	319	600	114	210	390
std. dev.	197.27	314.68	312.37	90.70	145.62	170.57
average	357.50	409.67	767.17	121.33	217.33	338.67
time reported in minutes						

Table 4.2. PMSC Study Data for More Experienced Programmers

The next step for this statistical analysis is to perform a Wilcoxon matched pair signed rank non-parametric test. The result of this test is a p-value p = 0.025, which is less than α = 0.05 and implies 95% confidence. This result means that there is statistically significant evidence of a difference between the null hypothesis H_0 (i.e. μ_0 = 0) and the actual median size μ_1 (i.e. μ_1 = 6 hours 09 minutes). Therefore, the hypothesis H_1 is verified. In other words, PMSC shortens the completion of change requests.

Regarding hypothesis H_4 that states that PMSC shortens the completion time of software changes done by experienced programmers, the verification requires a measurement of the difference between the amount of time spent by the experienced programmers completing the change requests without PMSC and the amount of time taken to complete the change requests with the assistance of PMSC. The null hypothesis H_0: μ_0 = 0 (median = 0) considered in this case is that PMSC does not assist experienced programmers in the completion of their change request. H_4: μ_4 is then compared to the null hypothesis H_0: μ_0 = 0 and the p-value generated from the Wilcoxon matched pair signed rank is p = 0.059. This p-value is slightly greater than α = 0.05 at a 95% confidence test, thus implying that although we find improvements, there is no evidence of statistical significance that experienced programmers require less time to complete their software change.

4.3. PMSC Code Analysis Performance Results

For our experiment, analyzing PMSC code analysis performance implies the verification of hypothesis H_2 and hypothesis H_5. For this purpose, the data in table 4.1 is once again statistically analyzed in an attempt to validate the hypothesis H_2, which

states that programmers require less time during code analysis following PMSC. The null hypothesis considered to prove H_2 implies that PMSC has no effect on the time spent by programmers to analyze source code. Thus, H_2: μ_2 is compared to the null hypothesis H_0: $\mu_0 = 0$ and the p-value generated from the Wilcoxon matched pair signed rank is $p = 0.009$. This p-value is less than $\alpha = 0.05$ at a 95% confidence test. An immediate conclusion from this result is that there is statistical significant evidence that programmers need less time in code analysis when they use PMSC.

A similar statistical analysis is done for hypothesis H_5: μ_5. That is, PMSC shortens the code analysis portion of the time of software changes done by experienced programmers. The null hypothesis H_0: $\mu_0 = 0$ considered to evaluate H_5 infers that PMSC does not assist experienced programmer in the code analysis phase. After comparing H_5 to the null hypothesis H_0, the Wilcoxon matched pair signed rank generated a p-value $p = 0.036$, which is less than $\alpha = 0.05$ and implies 95% confidence. Thus, there is also statistical significant evidence that experienced programmers need less time in their code analysis when they follow PMSC.

4.4. PMSC Code Implementation Performance Results

Finally, the same statistical analysis is repeated for hypothesis H_3 and hypothesis H_6, which are both related to the code implementation performance of PMSC. Hypothesis H_3 suggests that programmers require less time during code implementation following PMSC. The null hypothesis H_0: $\mu_0 = 0$ considered in this case is that PMSC does not assist programmers in code implementation. For H_3, performing a Wilcoxon matched pair signed rank after the comparison of the null hypothesis and hypothesis H_3: μ_3 does not generate convincing statistical evidence supporting that programmers

require less time during code implementation when they follow PMSC. In fact, the p-value obtained is p = 0.131, which is greater than α = 0.05 at a 95% confidence test.

We also used the previous statistical analysis to evaluate hypothesis H_6: μ_6. This hypothesis asserts that experienced programmers require less time during code implementation following PMSC. The null hypothesis H_0: $\mu_0 = 0$ adopted to prove H_6 indicates that PMSC does not assist experienced programmers in code implementation. Next, the hypothesis H_6 is compared to the null hypothesis H_0 and the Wilcoxon matched pair signed rank provided a p-value p = 0.281, which is greater than α = 0.05 at a 95% confidence test. Therefore, there is not conclusive statistical evidence that substantiates that experienced programmers require less time during code implementation following PMSC.

From the verification of these hypotheses, an early conclusion can be drawn. Even though the PMSC approach aid programmers in code analysis, any improvement in code implementation or testing might heavily depend on the individual programmers' native programming skillsets and experience.

4.5. PMSC Qualitative Review

This section examines whether some of the qualitative goals of the user study are reached. Specifically, it elaborates on the effectiveness of PMSC, its sufficient definition, and its completeness.

4.5.1. PMSC Effectiveness

The data collected and presented in table 4.3 provides evidence to substantiate that PMSC is an effective process for programmers completing software changes from the participants' perspective. In fact, 67% of the participants claimed in their post –

experiment survey that it was helpful for them to complete their change request with the assistance of PMSC as opposed to completing it without any assistance. Besides, 83% of the programmers reported in their stage log reports having saved time when following PMSC. Specifically, the reduction of time was more noticeable in the concept location and impact analysis phases, both constituting the code analysis phase, rather than other phases of the software change process. 80% of the participants affirmed having saved time in the concept location phase and 70% of them reported having saved time in the impact analysis phase. In addition to that, respectively 37% and 30% of the programmers mentioned having saved time in the refactoring and verification phases. Finally, 60% of the participants whom PMSC saved the time reported time savings during actualization.

Table 4.3 shows the results from the post-study survey questionnaire. One important remark from this survey is that programmers on average spent less time during the stage using PMSC even though they claimed that the software change requests were more challenging in that stage. Also, the examination of the programmers' comments and observations from their stage logs and post-study survey indicates that the programmers found the concept location and impact analysis phases as the most laborious phases because of the necessity to decipher the system and its source code. Regardless, less time is needed to quantitatively and qualitatively perform code analysis with the assistance of PMSC than without.

Post Study Survey Questionnaire		
Question	*Yes*	*No*
Was PMSC More Effective?	8	4
Did PMSC Save Time?	10	2
If so in which phases?		
Concept Location?	8	2
Impact Analysis?	7	3
Refactoring (Pre & Post)?	3	5
Actualization?	6	4
Verification?	3	7
On a scale of 1 - 5 rate the difficulty performing the change request in the following stages:	Avg.	$\pm\sigma$
Stage 1?	3	1.16
Stage 2?	4	1

Table 4.3. PMSC Post-Study Survey Data

4.5.2. PMSC Sufficiently Defined

In theory, the current definition of PMSC phases is easily understandable, but it is not always the case in practice. In fact, some participants perceived limitations in impact analysis. They were not able to accurately discern whether that phase was complete or sufficient without having to inspect all the pieces of code reported by the impact analysis tool JRipples. As a result, there was a wide variation and inconsistencies in their time, effort, and performance. Some participants found necessary to perform thorough class inspections in the impact analysis phase. A comment from one of these participants was that "*The change request required a lot of analysis based on inspecting all of the classes flagged as next for the estimated impact set*". Other participants inspected selected classes instead. One of them reported that

“*During Impact Analysis, 174 classes were marked as a neighboring class. Seemed too many classes to analyze*”.

These remarks from the participants are in alignment with what Dorman mentions in his report [8] when he points out that some aspects of impact analysis needs more clarification. More precisely, he emphasizes that impact analysis needs a better definition of its exit criteria. As a reminder, exit criteria refer to the conditions at which a phase should stop as opposed to entrance criteria which refer to the conditions at which a phase should start. From this observation, it should be understood that further research in determining entrance / exit criteria would be valuable to improve PMSC.

Another remark is made on the students’ logs concerning concept location. That is there is no guideline about the appropriate circumstances in which the concept location techniques available in JRipples should be used. This lack of indication in the choice of concept location technique led us to allow the participants to select a technique according to their liking. As a result, this freedom of choice brought the participants to confusion as they did not know whether they should perform grep analysis, dependency search, or both. This aspect of PMSC would also benefit an in-depth investigation.

4.5.3. PMSC Completeness

Although the participants’ feedback infers that all the activities and tasks they performed are addressed by PMSC, it is essential to mention that the comments from the participants’ log reports also reveal that the PMSC phases do not always succeed

each other as presented on figure 2.4. Two main observations are made from these log reports.

The first observation concerns the code analysis phase. Specifically, the transition between conception location and impact analysis is not always perceivable since it occurs mostly in an intermixed way. Sometimes programmers can start impact analysis without knowing exactly whether all the code fragments highlighted by concept location are found or not. It may happen that other code snippets that were missed from conception location get discovered during the impact analysis phase. Thus, there exist an iterative aspect between concept location and impact analysis.

The other remark is related to code implementation. Specifically, refactoring and actualization must also allow iterative characteristics between phases. The reason is that performing code change during refactoring could generate a ripple-effect for additional impact analysis or actualization supporting the software change.

4.6. Less Experienced versus More Experienced Programmers

The data collected from the pre-experiment survey allowed us to divide the participants in two main groups. The criteria for the differentiation were the number of years programming in multiple languages and their experience level. The first half of the participants was classified as “more experienced programmers” with an average of 4.33 years programming in Java, while the second half was identified as “less experienced programmers” with an average of 1.13 years programming in Java. Figure 3.1 gives more details about the ranking of the participants. The outcome of the in-depth analysis conducted on the students’ performance differences over the stage 1 and the stage 2 of the experiment was that there was an obvious improvement observed with both the less

experienced programmers and more experienced programmers. As depicted in figure 4.1, 5 out of 6 less experienced programmers reduced their overall time in completing software changes with the assistance of PMSC compared to their overall time in completing software changes without PMSC. Similarly, figure 4.2 shows that 5 out of 6 more experienced programmers completed their change requests faster with the assistance of PMSC than without PMSC.

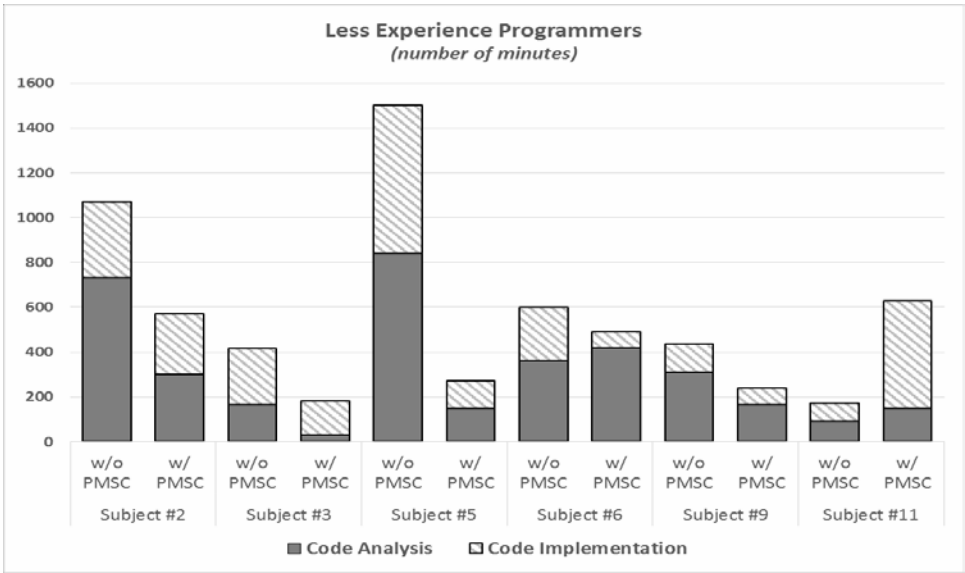


Figure 4.1. Less Experienced Programmer (Individual) Comparison

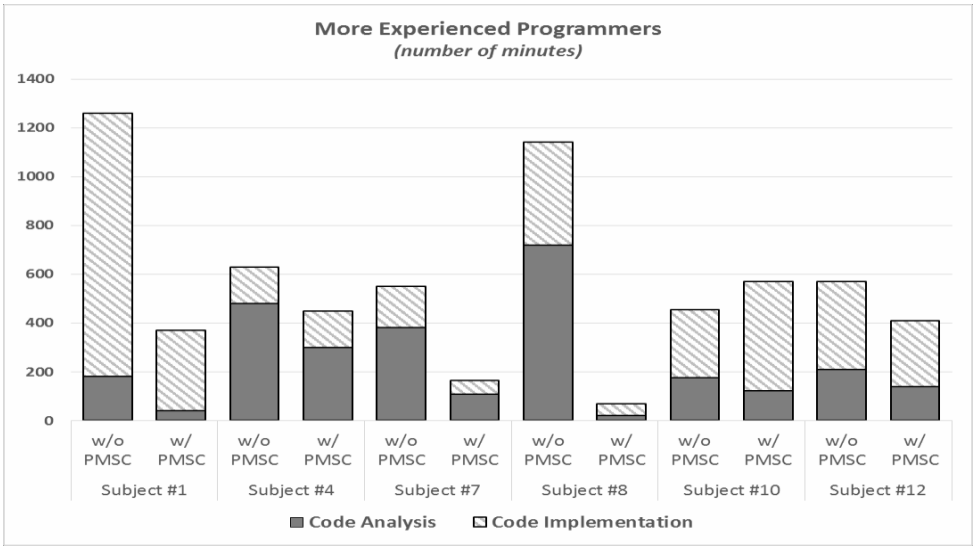


Figure 4.2. More Experienced Programmer (Individual) Comparison

The less experienced programmers completed their software changes with on average 44% less time. They precisely used 51.3% less time for the code analysis constituted of concept location and impact analysis, and they 31.2% less time during code implementation composed of refactoring, actualization, and verification.

The more experienced programmers reduced their overall time to complete their software changes by 55.8%. While following PMSC, they spent 66% less time to perform their code analysis and finished their code implementation in 46.9% less time than the time they spent without the assistance of PMSC. One could presume that performing software changes is relatively obvious for experienced programmers. However, the data results summarized in figure 4.3 and figure 4.4 indicate that more experienced programmers gain from using PMSC.

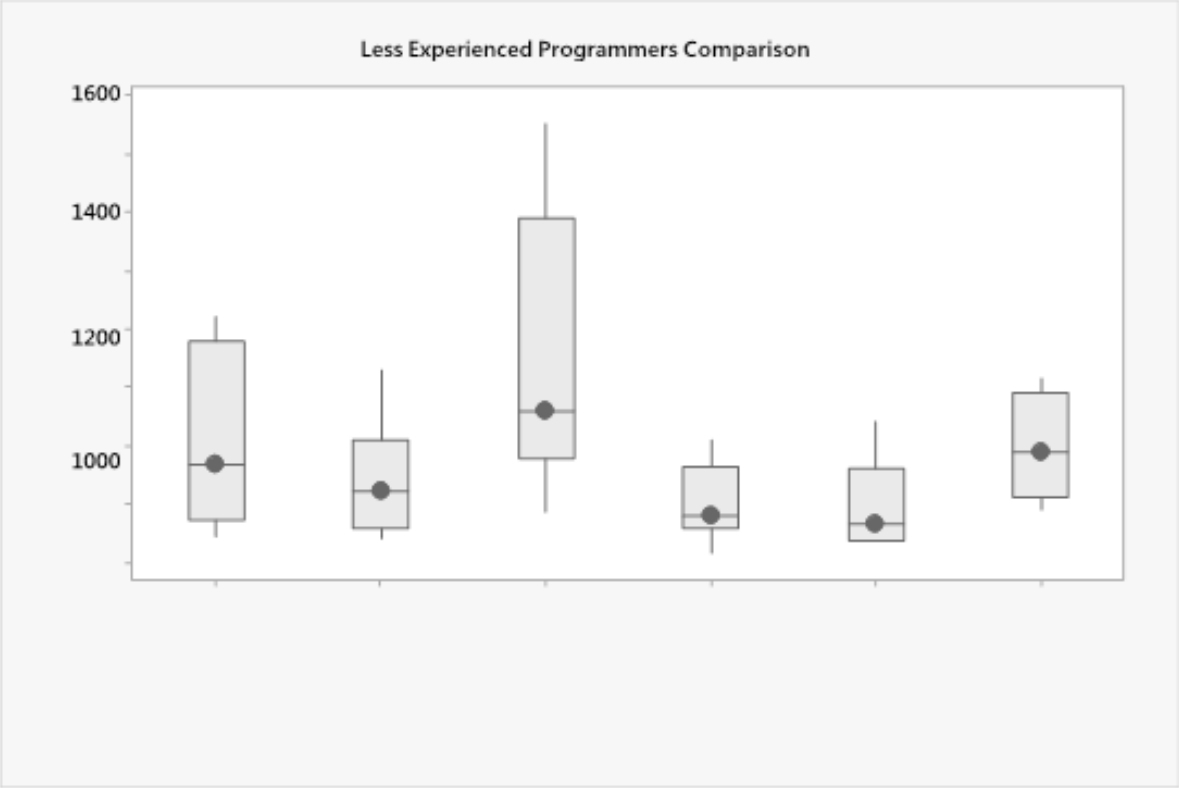


Figure 4.3. Less Experienced Programmer (Group) Comparison

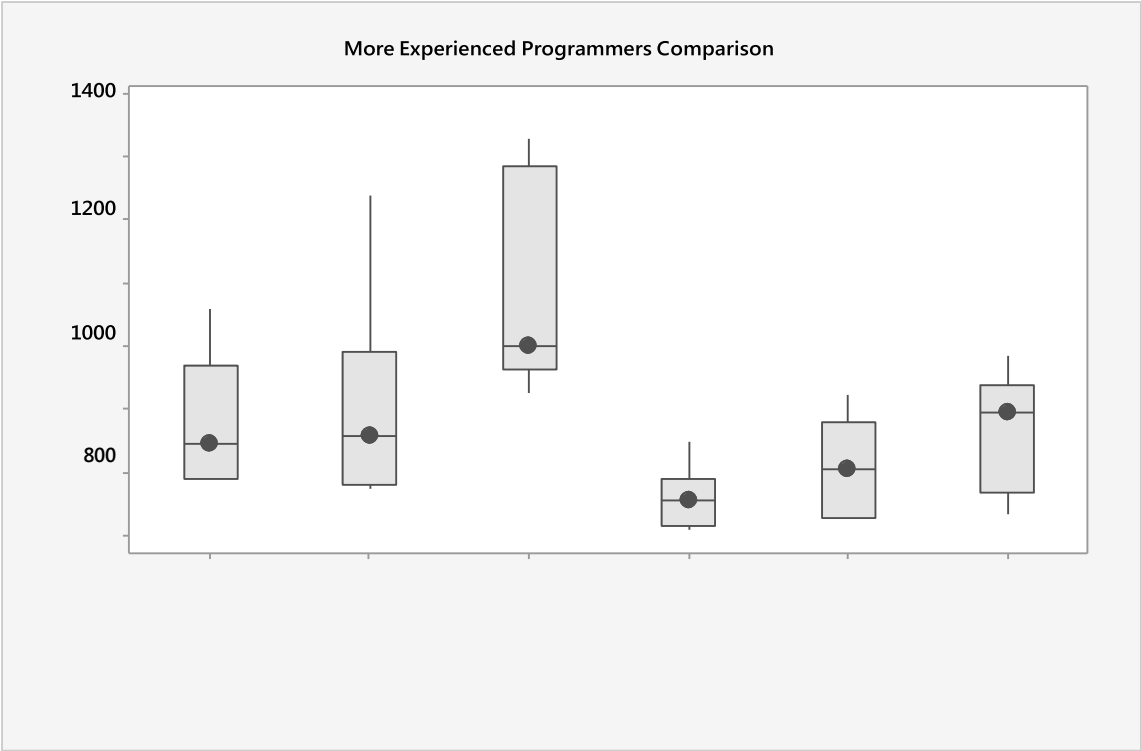


Figure 4.4. More Experienced Programmer (Group) Comparison

The statistical analysis of the data collected substantiates that PMSC effectively assists both less experienced and more experienced programmers in the concept location and impact analysis phases of software change. However, the same remark cannot be deduced for the code implementation phases. As a matter of fact, statistical findings do not provide significant evidence to confirm that PMSC aids programmers in code implementation. Thereby, it could be inferred that any improvement in both code implementation and verification phases essentially depends on individual programmers' innate programming skills and personal experience.

4.7. Tool Support

The use of supporting tools within PMSC phases produces diverse outcome as not all PMSC phases dispose of specific tool. Feedback from the participants sustained

that the supporting tools used during concept location and impact analysis were useful, yet the lack of integration of the set of tools within the experiment was problematic. In addition, it was quite challenging for the students to fully understand a set tools before being able to implement their change requests. Because of that, it was imperative to have training sessions. JRipples played an important role in the study as it helped in supporting integration between concept location and impact analysis. On the contrary, there was no such integration support in the code implementation and verification phases. An example of integration in the later phases could be to mark all source code fragments modified during refactoring and actualization. Therefore, an integrated support could be to do unit testing and regression testing based on the modified modules. This type of integration would have been immensely helpful to the students. Lastly, although the PMSC process can be followed without any tool support, integrating tool support within the process would further improve the quality of work achieved by programmers.

4.8. Threats to Validity

As in almost every experiment, there exist multiple factors that can compromise the results of an ongoing research and lead to a misinterpretation of the findings. One of the main concerns for us then becomes the identification these confounding factors and how to counter them. As a reminder, confounding factors are variables not under investigation, which may somehow affect the results of the experiment. In this study, we made sure that quantitative data was properly collected in order to effectively prove the hypotheses listed in section 3.2. In addition to that, pre-experiment and post-experiment qualitative data was collected to strengthen the quantitative data. The rest of this

section explains how the subjects of this study were selected and identifies potential threats. For the later matter, it elaborates on how internal and external validity were both ensured.

While designing the user study, we determine the potential candidate for the experiment. The options available to us are to choose between professional software engineers and students in a computer science department. The decision is eased by [5, 26], which address the problem of using students instead of professional programmers. In fact, it is usually advantageous to select students for software engineering research, yet there exist circumstances where it is more appropriate to use professional programmers. In [5], claim is made that there are contexts where the use of students in research should be embraced as a complementary approach to attempt to rely on a sampling of professional programmers rather than being considered as an inadequate technique. Similarly in [26], it is demonstrated that the use of students instead of professional software engineers only generates minor differences in performance results. However, the difference in performance between a beginner and an expert programmer is obvious when they perform much complex task such as leveraging asynchronous programming or security when necessary. This idea concurs with the research presented in [32], which identifies how novice programmers handle exceptions as opposed to senior developers. To verify the hypotheses of this study, we estimate that working with graduate students rather than undergraduate students is suitable. The reason for that is there is a distinction between less experienced and more experienced programmers and all the observed similarities and differences are reported.

Key dependent and independent variables are identified to undertake potential threats to validity. For this experiment, these variables are the size of the candidate systems, the variety and complexity of the software change requests, the programmers' experience. Consequently, we made sure to precisely address these variables in the design of the user study.

Regarding internal validity, we made certain that the participants' programming background was known. Having that information helped them in determining and countering the major confounding factors that could influence the participants. These confounding factors were years of programming experience, familiarity with the objects within the user study, familiarity with the proposed approach, and the supporting tools. Moreover, the learning effect issue was addressed. More precisely, the knowledge acquired during the first stage of the experiment might have increased the participants' effectiveness in completing the later software change requests. Therefore, actions outlined in section 3.2 were used to attenuate this learning effect.

Another effort to enforce internal validity was to standardize as many of the conditions as possible. These conditions included the establishment of the tools to be used within the study and the production of a standard format for the documents required to collect both qualitative and quantitative data at each stage of the study. Regarding data collection, it was acknowledged that self-reporting of performance was limited at times. To lessen these limitations, a quality control review of the participants' log was incorporated in the process of reviewing the different log reports. In other words, for any log report with unclear or doubtful data results, a follow-up questionnaire was sent to its author for further clarification. Internal validity was also imposed by

selecting and designing the user study. Specifically, the software change requests and the subject systems under study were randomly selected and assigned to the participants.

While internal validity is more related to the technical aspect of the study, external validity focuses more on the knowledge and the experience of the participants. The programming background from the subjects of a study may be diverse and some programmers may be accustomed to specific programming technologies, source code, and application domain more than others. For this experiment, the participants might have more practical knowledge about various searches and comprehension approaches. Again, the use of students as opposed to professional programmers only generates minor differences in performance results in certain circumstances [26]. In the interpretation of the results, care was taken not to generalize the results too much. The main confounding factor to address for external validity was the classification of the participants in groups of more experienced and less experienced programmers. Specifically, we confined in a narrow and relevant space the comparison of the results from the more experienced programmers against the less experienced programmers.

Despite the careful design and the knowledge acquired by its participants, weaknesses were noticeable in the experiment and some of them are the following. First, the participants of the study did not work in a control environment. They worked at home and at their own pace, so the data that some students entered in the logs could be considered trustworthy. Therefore, we had to discard some of the student logs from the study. Additionally, the students had so much freedom about their working environment that we run into problems when some of them could not use some of the

Windows-based supporting tools (i.e. TortoiseSVN) on their Mac system. Second, PMSC was exhaustively taught in class. Nevertheless, we failed to practically present all its facets during the experiment. For example, we did not show the participants how to determine the estimated impact set nor the actual impact set. A direct consequence was that we did not collect any data about it. Furthermore, we were not able to collect enough data about neither the refactoring nor the verification phases because of the lack of training on the supporting tools. One reason is that refactoring is not required, so some students did it and some others did not. About the verification phase, we probably did not get enough data from students because they did not know how to do testing appropriately. There was also confusion among the students about change propagation. In JRipples for instance, it was unclear when to mark a class “Next” or “Propagated”. A last remark is that the participants were working more as individual programmers than team members in their respective groups because the change requests were not designed in such a way that the students work together as a team.

We also acknowledged that the two classes of programmers could have performed differently under others software engineering conditions. It is clear that assigning different change request, software systems, and application domains may require different efforts to complete the changes. This outcome may happen as well if PMSC is merged with supporting tools different from the ones used in this user study. It should additionally be kept in mind that the subject systems of this experiment are not a stereotype of all types of software systems. However, they were selected such a way that they were diverse in size and complexity. Any attempt to generalize the findings to

other software change requests and types of software systems should be done thoughtfully.

CHAPTER 5: CONCLUSION AND FUTURE WORK

This thesis explored how a phased model for software change (PMSC) affects the time programmers spend to complete software change requests. An exploratory study was conducted by graduate students implementing a set of software changes. The first stage consisted of the participants using their previous knowledge of software engineering to complete software change requests, while in the second stage, the same participants completed another set of change requests, but this time with the assistance of PMSC that was introduced to them after stage 1. Our findings show that there is statistically significant evidence that PMSC considerably reduce the time used to implement a software change. In fact, the required time to do so may be reduced by half. We additionally discovered that PMSC aids both experienced and less experienced programmers not only in code analysis activities, but also in code implementation phases.

Upon acknowledgement of the strength and weaknesses of this current study, a repeat experiment should have the students working in a controlled environment such as a lab, with all the required tools installed on the computers from the lab. This way, their work can be monitored appropriately and the data collected from them would be more reliable. Another recommendation would be to incorporate more training to the study to avoid any sort of confusion among participants.

Although there is evidence that PMSC is an effective process, offering additional guidance for specific PMSC phases would improve the process. Also, greater integration and a smoother transition between the tools would further improve impact of PMSC.

Future work would include an exploration of entrance and exit criteria of the PMSC phases.

APPENDIX A: Sample of Software Change Requests

Sample of jAdvisor Change Requests

1. Permutation Request: Generate the various permutations of a schedule that exists from a set of selected courses. The user will just enter the courses that they want to take and the program should create the various schedules and present them to the user in an organized way.

2. School Adapter: Research the creation of a school adapter like the ones included with the program. This school adapter should download the appropriate school schedule for Wayne State, if possible. If not possible we should implement a school adapter for any other Michigan school. Once a proper document describing the creation of a school adapter is finalized, the programmer will implement the adapter.

3. Schedule Display: Currently, the user is not allowed to add a class that has a time confliction. This is immensely unusable. Instead the user should be able to add such a class, however, all classes that overlap at any time should be shown as red instead of the default. Also they should be labeled as conflicting.

4. XML Class Schedule Support: Research the usage of XML as a way to save the output schedule of the user. The XML structure must be clear and readable. Once this structure is documented, as a DTD file, the user should be able to save their output in the XML format, and also the program should be able to read the XML format into both the planner and schedule tabs.

5. Color Coding Schedule: JAdvisor allows the user to enter block time to the schedule; however, it is very limited. You are to add a classification drop down box to the block time menu. It is to display various classifications like study time, food break

time, homework time, etc. The amount of different classifications is up to you, but should reflect various parts of a schedule. Each classification should then show up as a different color block on the schedule. This way the user can easily identify their allocated time.

6. Planner Duplication: The planner is supposed to allow the user to plan all four years of their curriculum, but it allows for duplicates. Since usually students do not repeat their courses, the program should ask the user for an overwrite if they do add a duplicate course either in the same semester or future semesters. Calculate the number of credits taken for each duplicate course.

7. Planner Courses: Create a planner wizard that allows the user to enter in all of the courses necessary for their degree and the maximum and minimum number of credits for each course. These should then be saved in XML format. Next show these courses in the planner tab on the right hand side. If a course has been taken and the maximum credits is satisfy the name should appear in red and the user should not be allowed to select it. Also the mandatory courses should appear in red in the wizard. A mandatory course has the minimum credits greater than 0. In our department, for example, CSC6580 and CSC6500 are mandatory courses.

8. HTML Support: The current implementation allows the user to output HTML, but not to read it in. The user needs this ability. The program should read an HTML file and fill in the scheduler or the planner as if it were saved. You can decide if an HTML is a planner or a scheduler based on the column names of the table.

Sample of jEdit Change Requests

1. “Modify the splash window”: Currently the splash window of jEdit is a static picture. Add the names and emails of your group members to it. And add moving text as the same effect shown in “About jEdit” dialog. Adjust the scrolling speed so that all text can be shown.

2. “Zoom the text editor”: Under menu View, add two menu items “zoom+” and “zoom–“ to scale the editors. At this stage, the scaling factors are not defined. The view should be able to be scaled multiple times.

3. “Search and mark all”: Under menu Search, add menu item “mark all”. Locate all matches and add markers to all of the lines.

4. “Add timestamps to log”: Locate where the activity.log is. Currently there are no timestamps in the log file. Add timestamps to all kinds of messages.

5. “Duplicate data when creating a new view”: Currently clicking View | New View will create a new view for the same data; which means that modification in one view will affect the other one. Add menu item New View&Buffer under menu View to allow the user to duplicate data for the current shown view only.

6. “Show/Hide whitespace”: Currently jEdit shows a red dot at the end of every line. Newline is the only whitespace symbol that jEdit shows. Add menu item Show/Hide whitespace under menu View to allow the user to choose whether all whitespace symbols (newlines, blanks, and tabs) will be shown. At this stage you do not have to worry about editing of the text with whitespace showing.

7. “Search list”: Currently jEdit allows users to access the text that was previously search by pressing page_up or right-click keys in Search Dialog. Display in a listbox the last 5 text fragments that were previously search.

8. “Signature”: Allow the user to specify a signature to be used as the footer in all printed documents. An option should be available to enable/disable the signature. When the option is enabled, the signature will appear in the status bar.

9. “Edit remote files”: The user can indicate the URL of the file; jEdit retrieves the document to local machine for modification; then puts it back to the location named by the URL to overwrite the original one. At least protocols of HTTP and FTP should be supported.

10. “Simulate notepad appearance”: Draw horizontal black lines, which separate continuous lines. The appearance is like paper in a notebook.

11. “Record the typing speed of the user”: Record how many characters and words the user types in this session and show how fast he/she is. Use words per minute to measure the speed. The information will be shown at the status bar.

13. “vi-style input”: The procedure is: 1) double press key “ESC”; 2) input a number X; 3) type in something; 4) press “ESC” again, the sentence which you just typed will be inserted X times at the current position. Any key sequences not following the procedure exactly will not invoke this behavior.

14. “open read only”: Allow the user to open the file in read-only mode. All the features except editing should be available.

Sample of JabRef Change Requests

1. Consolidating BibTeX files

Input: a folder, **output:** a .bib file

Hints:

- scan recursively the input folder and its sub folders
- find all BibTeX files
- parse these files to BibTex Databases
- merge these databases, remove conflicts if any
- save the consolidated databases to a output file

Create GUI for this functionality

2. Shrinking BibTeX files

Input: a .bib file, a folder containing .tex files**, output:** a new .bib file

Hints:

solution 1:

- scan .tex files
- find citation command (\cite, \citet, \citep) to collect the keys of the BibTeX items used
- compare to the keys in the .bib file, remove any redundant items

solution 2:

- compile the .tex file using bibtex commands
- open the output .bbl file of the .bib file
- read all the bibtex items that have been used
- compare and remove any redundant items

Create GUI for this functionality

3. Unicity of bibTeX key

In the feature "Autogenerate BibTeX keys" keys are generated in this format [author][year].

Make a change so that the BibTeX keys have the timestamp added to the format like this [author][year]_[hhmmss] (e.g. Brooks2010_083025).

4. Auto-update timestamp on edit

The current format of the timestamp when adding an entry is [year].[month].[day] (e.g. 2013.11.18).

Make a change so that the timestamp has the format [year][month][day].[hh][mm][ss] (e.g. 20131118.083025) and it is auto updated when the button auto is clicked.

APPENDIX B: Pre – Experiment Questionnaires

This appendix contains the pre - experiment questionnaires of the study. Only a sampling of these questionnaires is shown in this thesis to preserve the length of the document. Six out of twelve pre - experiment questionnaires are kept for this purpose and the criterion of selection of reports is the level of experience of the participants. Specifically, three less experienced and three more experienced participants had their questionnaires selected.

The full list of pre - experiment questionnaires is available online via the following link:

https://drive.google.com/folderview?id=0BwkmElTjUf2qVjZVXzdDaWpSdHc&usp=sharing.

CSC 6110 Student Pre-Experiment Questionnaire

Please answer the following questions for the CSC 6110 team project assignments to the best of your ability.

Student # 4

Graduate Status
- (•) Masters
- ( ) Ph.D.

Please indicate programming experience

Java
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

C/C++
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

Other (Please list)

- ( ) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience

Experience / Familiarity with the following applications

Eclipse	( ) None / NA	(•) Beginner	( ) Intermediate	( ) Expert
SVN / TortoiseSVN	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
Abbot Java GUI Test Framework	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
JUnit	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert

CSC 6110 Student Pre-Experiment Questionnaire

Please answer the following questions for the CSC 6110 team project assignments to the best of your ability.

Student # 5

Graduate Status
- (•) Masters
- ( ) Ph.D.

Please indicate programming experience

Java
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

C/C++
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

Other (Please list)

- ( ) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience

Experience / Familiarity with the following applications

Eclipse	( ) None / NA	(•) Beginner	( ) Intermediate	( ) Expert
SVN / TortoiseSVN	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
Abbot Java GUI Test Framework	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
JUnit	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert

CSC 6110 Student Pre-Experiment Questionnaire

Please answer the following questions for the CSC 6110 team project assignments to the best of your ability.

Student # 6

Graduate Status
- (•) Masters
- ( ) Ph.D.

Please indicate programming experience

Java
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

C/C++
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

Other (Please list) _____

- ( ) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience _____

Experience / Familiarity with the following applications

Eclipse	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
SVN / TortoiseSVN	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
Abbot Java GUI Test Framework	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
JUnit	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert

CSC 6110 Student Pre-Experiment Questionnaire

Please answer the following questions for the CSC 6110 team project assignments to the best of your ability.

Student # 7

Graduate Status
- ○ Masters
- ◉ Ph.D.

Please indicate programming experience

Java
- ○ Beginner
- ○ Intermediate
- ◉ Expert

Years of experience 7

C/C++
- ○ Beginner
- ◉ Intermediate
- ○ Expert

Years of experience 2

Other (Please list) Python, JavaScript
- ○ Beginner
- ○ Intermediate
- ○ Expert

Years of experience

Experience / Familiarity with the following applications

Eclipse	○ None / NA	○ Beginner	○ Intermediate	◉ Expert
SVN / TortoiseSVN	○ None / NA	○ Beginner	◉ Intermediate	○ Expert
Abbot Java GUI Test Framework	◉ None / NA	○ Beginner	○ Intermediate	○ Expert
JUnit	○ None / NA	○ Beginner	○ Intermediate	◉ Expert

CSC 6110 Student Pre-Experiment Questionnaire

Please answer the following questions for the CSC 6110 team project assignments to the best of your ability.

Student # 8

Graduate Status
- (•) Masters
- ( ) Ph.D.

Please indicate programming experience

Java
- ( ) Beginner
- (•) Intermediate
- ( ) Expert

Years of experience 1

C/C++
- ( ) Beginner
- (•) Intermediate
- ( ) Expert

Years of experience 2

Other (Please list)

- ( ) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience

Experience / Familiarity with the following applications

Eclipse	( ) None / NA	(•) Beginner	( ) Intermediate	( ) Expert
SVN / TortoiseSVN	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
Abbot Java GUI Test Framework	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
JUnit	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert

CSC 6110 Student Pre-Experiment Questionnaire

Please answer the following questions for the CSC 6110 team project assignments to the best of your ability.

Student # 9

Graduate Status
- (•) Masters
- ( ) Ph.D.

Please indicate programming experience

Java
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

C/C++
- (•) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

Other (Please list) PRO C
- ( ) Beginner
- ( ) Intermediate
- ( ) Expert

Years of experience 1

Experience / Familiarity with the following applications

Eclipse	( ) None / NA	( ) Beginner	(•) Intermediate	( ) Expert
SVN / TortoiseSVN	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
Abbot Java GUI Test Framework	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert
JUnit	(•) None / NA	( ) Beginner	( ) Intermediate	( ) Expert

APPENDIX C: Stages Log Reports

This appendix contains the study log reports from Stage 1 and Stage 2. However, to preserve the length of this thesis, we present a sampling of our log reports in this thesis. Only six out of twelve reports are kept for Stage 1 and six out of twelve reports are retained for Stage 2. The criterion of selection of the reports is the level of experience of the participants. Specifically, three less experienced and three more experienced participants had their log reports selected for Stage 1. The same participants also had their log reports selected for Stage 2.

The stages log reports presented in this thesis as well as the ones not shown are available online at the following addresses:

- Stage 1 Log Reports: https://drive.google.com/folderview?id=0BwkmElTjUf2qcHRLWTBlY25NYUU&usp=sharing
- Stage 2 Log Reports: https://drive.google.com/folderview?id=0BwkmElTjUf2qSWJfRXduTVdXQnc&usp=sharing

CSC 6110 Project Results Log

Student #4

Change Request#: 5

"Duplicate data when creating a new view" *Currently clicking View | New View will create a new view for the same data; which means that modification in one view will affect the other one. Add menu item New View&Buffer under menu View to allow the user to duplicate data for the current shown view only.*

1 Detailed Report

1.1 Code Analysis

The steps performed for implementing the change were:

1. *Before starting to work on the change, the JEdit tutorial was read on some of the basic features like View, Buffer and the relationship between them to understand their functionality to better work on the change.*
2. *Performed code search using the string "view menu" and retrieved the list of classes that were associated with it.*
3. *Analyzed the dependencies between the classes and the underlying methods to narrow down the classes that needs changes.*
4. *Ran the JEdit.java class to see the initial output to analyze the various View menu options to better understand the change request.*
5. *Inspected the class jedit_gui.props on how a new menu has been added and after reviewing the same, added a new menu item new-view-buffer under the view menu.*
6. *Tested the code if the new menu is added under the view menu.*
7. *After the new menu was added, the action of creating a new buffer had to be assigned to the created menu. So searched for "action" in the search bar and found that actions.xml was the relevant file to create the action for the new menu item created.*

8. *In the new buffer that was created, some text was typed in and the new-view-buffer menu was selected to see if the change in text in one view is affecting the other. It was affecting, so started with the code search that deals with the buffering of the text area from one view to the other.*
9. *Searched for the string "buffer" and retrieved classes like Buffer.java, BufferHandler.java, BufferOptions.java, BufferChanging.java, BufferHistory.java, etc.*
10. *After visiting the mentioned classes and their dependent classes and methods, figured that Buffer.java is the class that has to be referred to make changes.*
11. *The class Buffer.java had a variable called 'dirty' which is set to true when the user had entered some input in the text area. If there is no input then the attribute 'dirty' is set to false. So the 'dirty' attribute was set to true when there was some user input in the text area and the content of the current buffer was copied onto the buffer of the second view.*
12. *Tested the functionality again and this time the change in text in one view did not affect the other thus implementing the change request.*

Please provide a detailed journal entry describing how you went about identifying and determining which source code files needed to be modified in order to support this change request.

Describe the steps performed, how you went about inspecting / investigating the source code files and where to make the necessary changes.

Code Files Visited

Code Files Visited / Inspected Only	Comments
13	1. JEdit.java 2. Actions.xml 3. jedit.props 4. jedit_gui.props 5. Buffer.java 6. TextArea.java 7. JEditTextArea.java 8. BufferHandler.java 9. View.java 10. ViewOptionPane.java 11. JEditBuffer.java 12. BufferOptions.java 13. BufferChanging.java

1.1.1 Code file 1 – Jedit.java

This being the main class file, ran the file to check the output of the JEdit editor and analyzed the various options under the view menu and also in-depth code inspection was done for the 'new-view' method to understand the functionality of the method.

1.1.2 Code file 2 – Actions.xml

The motivation behind visiting this file was to check how the menu items are given an action to perform. The search term provided was 'action' that retrieved a list of class files that had action as the string. After visiting all the classes, this xml file seemed the most relevant as there were functions that assign actions to the menu items.

1.1.3 Code file 3 –jedit.props

This file was visited in a thought that this would be the file to create the new menu item. When

the search ‘menu’ was given in the search bar this file was brought up by the search and when inspected this file, realized that this file was not useful.

1.1.4 Code file 4- jedit_gui.props

This was the next jedit file that showed up when the search ‘menu’ was given. The previous file jedit.props did not seem to be useful. So the next file in the result was investigated to find out if this could be used for creating menus. After analyzing the entire file, concluded that this is the right one for creating menus.

1.1.5 Code file 5- Buffer.java

The next thing to look for after creating menu and assigning an action was files related to buffer and view. To implement the change request, the understanding of the working of buffer and view was very important. The search term given was ‘buffer’ and that produced Buffer.java, BufferHandler.java, BufferOptions.java and BufferChanging.java. The files retrieved were analyzed line by line to understand how buffer is created and under what condition a buffer is created. Buffer.java had all the required information for buffer creation while the other class files did not turn out to be useful.

1.1.6 Code file 6 – TextArea.java

TextArea.java was visited to check if there was any functionality regarding the text input in the text area. There was no search term provided for the file. It was randomly selected for analysis as the name of the file seemed relevant. But this class file contained information about the font size, font and various formatting options for the text. So this class was not useful for the change.

1.1.7 Code file 7-JEditTextArea.java

This file was the next file that was picked for analysis when the previous TextArea.java did not seem useful. But this class file also did not have any useful information.

1.1.8 Code file 8- BufferHandler.java

This file showed up when the search for buffer was made but when this file was visited it

did not have any required information.

1.1.9 Code file 9 – View.java

The reason to view ‘view.java’ file was to analyze if the class has any methods for creating new view and check the functionality of the existing view. The search term given was ‘view’ using the eclipse search bar. This file was not useful for the change.

1.1.10 *Code file 10-* ViewOptionPane.java

The next file that was retrieved for the search ‘view’ was ViewOptionPane.java. This class also did not have any relevant information about view.

1.1.11 Code file 11-JEditBuffer.java

This file was retrieved for the search term ‘Buffer’. Visited the file to find out if it has any code fragment for buffer creation and concluded not useful.

1.1.12 Code file 12-BufferOptions.java

This also was one of the files that was brought out for the search term ‘Buffer’ and was not required for the request implementation.

1.1.13 Code file 13- BufferChanging.java

This was amongst one of the files that was retrieved for the search ‘Buffer’ and was not required for the change request.

1.2 Code Changes

Please provide a detailed journal entry describing how you went about performing the necessary coding changes for this change request.

Coding Change Summary

Code Files				
Visited	Changed	Added	Unchanged	Comments
13	3	0	10	Addition of statements

Modified Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
1.	jedit_gui.props	Created a menu item new-View-Buffer	2	0	2
2.	actions.xml	Created a new action for newViewBuffer	5	0	5
3.	JEdit.java	Added a new method newViewBuffer	13	0	13

1.2.1 Code file 1 - jedit_gui.props

Searched for the string “view menu” to retrieve the list of relevant classes. Inspected all the files that was pulled out on the search and found that this was the file where all the menu for the JEdit has been created. Accordingly, the menu item new-view-buffer was created under the View menu.

1.2.2 Code file 2 – actions.xml

After the new menu was added, its subsequent action had to be assigned to the menu item. This led to another search for classes that dealt with creating actions for the existing menu. The search string provided was ‘action’ that retrieved a list of classes that had dependencies with action. After visiting all the classes and their dependent methods, ended up in actions.xml that the search result was referring to. Analyzed the entire file as to how actions are created for each menu and deeply inspected the action created for “new-view” menu and a similar action was created for new-view-buffer.

1.2.3 Code file3 – Jedit.java

The change was to create a new view such that modification in one view does not affect the other when new-view-buffer menu is chosen. Currently the buffer changes in each view, so the current buffer is captured from the view and is assigned to the newly created view. When a user has typed in the TextArea, a new buffer is not created. A dirty bit is set to true to accomplish the same. Then the text from the first view is retrieved using the getText method and assigned to the new buffer using the setText method. This way the modification in one view does not affect the other.

1.3 Testing

Testing was done at 3 stages.

1. *Testing was done to check after the new-view-buffer was created.*
2. *Secondly, after the new menu was created, the action was assigned to it and testing was again performed to check if the action is assigned to the menu.*
3. *Thirdly, the testing was done to check if the modification in one view affects the other.*

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Stateme	Covere d	%		
1.	JEdit.java	5506	29	0.52%	0	0

1.4 Timing

For code analysis, no tool was used for calculating the time. The time shown below is an approximate manual calculation spent on analysis. Time spent for code change and testing

were captured from Rabbit.

Timing Totals

Phase Name	Time (hh:mm)
Code Analysis	08:00
Code Change	01:00
Testing	01:30
Total Time	10:30

1.5 Conclusions

The menu 'new-view-buffer' was created in a way that the content in one view does not affect the content in the other view while the content in view1 is copied onto view2.

CSC 6110 Project Results Log

Student #5

Change Request#: 3

Please provide a description of the change request / defect:

Currently, the user is not allowed to add a class that has a time confliction. This is immensely unusable. Instead the user should be able to add such a class, however, all classes that overlap at any time. Also they should be labeled as conflicting.

1 Detailed Report

1.1 Code Analysis

Please provide a detailed journal entry describing how you went about identifying and determining which source code files needed to be modified in order to support this change request.
Describe the steps performed, how you went about inspecting / investigating the source code files and where to make the necessary changes.

Code Files Visited

Code Files Visited / Inspected Only	Comments
# 2	TimeofDay.java Advisor.java

1.1.1 Code file 1

Please provide a detailed journal entry describing the reason / motivation for visiting / inspecting the file. Also please describe how code file inspection was performed (i.e. tools used, terms searched, etc…)

TimeofDay.java

Ans:- First, I executed the project and then in the project tried to add two classes with same time and in the console it showed error and through the use of EclEmma coverage I narrowed down the file TimeofDay.java and in that is isAConflict(). This is the reason for inspecting the file.

1.1.2 Code file 2

Please provide a detailed journal entry describing the reason / motivation for visiting / inspecting the file. Also please describe how code file inspection was performed (i.e. tools used, terms searched, etc…)

Advisor.java

Ans:- when I wanted to add two classes with same time in the initial phase of project ,eclipse console used to show error messages and I inspected it as I wanted System.*err*.println statement to show conflict message instead of error message in the console. This is the reason why I inspected this file.

1.2 Code Changes

Please provide a detailed journal entry describing how you went about performing the necessary coding changes for this change request.

Coding Change Summary

Code Files				
Visited	Changed	Added	Unchanged	Comments
# 2	# 1	#	# 1	TimeofDay.java - (CHANGED) Advisor.java - (UNCHANGED)

Modified Code Files

#	**Code File Name**	**Task**	**Lines of Code**		
			Added	**Deleted**	**Total**
1	TimeofDay.java	MODIFICATION	8	-	8

1.2.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

TimeofDay.java

Ans:- First , I executed the project and then in the project tried to add two classes with same time and in the console it showed error and then I saw TimeofDay.java file and modified the method isAConflict() with return type Boolean true to false and then I executed the projected which allowed to add two classes with same time and then I imported libraries java.awt.Color, java.util.*, javax.swing.JOptionPane , javax.swing.UIManager, javax.swing.JOptionPane and javax.swing.UIManager and then I created UI.put to add red color and JOptionPane.showMessageDialog to show conflict information in dialog box and then I added System.*err*.println statement to show conflict message instead of error message in the console.

1.2.2 Code file 2

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

1.3 Testing

Please provide a detailed journal entry describing how you went about performing testing for this change request.

I had created a testcase with start time 11 and end time 13.00 using assertequals, asserttrue, assertfalse and assertnotnull and in the output showed both errors and failures as zero in junittest on file timeofdaytest.java file. It also showed green color band instead of red color band which indicates there are no bugs.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	TimeofDayTest.java	16427	31	0.2	0	0

1.4 Timing

Please provide a detailed journal entry describing how any of the supporting tools aided with completing this change request.

Timing Totals

Phase Name	**Time (hh:mm)**
Code Analysis	14:00
Code Change	7:00
Testing	4:00
Total Time	25:00

1.5 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

Answer:- First I analyzed all codes to create a vision pattern of packages by looking for GUI(Graphical User Interface) package, planner package, scheduler package and adapter package for which I lost most of the time and then I executed the jadvisor project and narrowed down the file and method to modify with.

CSC 6110 Project Results Log

Student #6

Change Request 3: Schedule display

Please provide a description of the change request / defect:

Currently, the user is not allowed to add a class that has a time confliction. This is immensely unusable. Instead the user should be able to add such a class, however, all classes that overlap at any time should be shown as red instead of the default. Also they should be labeled as conflicting.

1 Detailed Report

1.1 Code Analysis

Please provide a detailed journal entry describing how you went about identifying and determining which source code files needed to be modified in order to support this change request.

Describe the steps performed, how you went about inspecting / investigating the source code files and where to make the necessary changes.

All the source code files were first downloaded and the following steps were followed to analyze and point out the changes needed in the files:

1. Firstly, since the task was about modifying schedule display, source code files or folders having any name relation to ‘schedule’ were inspected as a first approach. This approach led to inspection of folder ‘jadvisor.scheduler’
2. In the jadvisor.scheduler, all the files were analyzed briefly to get an idea of their inputs,outputs and function files that they may refer.
3. The search command in Eclipse was utilized to find for words that matched the string ‘conflict’ in all the files under the entire jadvisor project folder. To maximize the search options, the search string was given as *conflict*.
4. There were 12 matches that were obtained from the previous step. A picture of the results is shown figure 1 below.

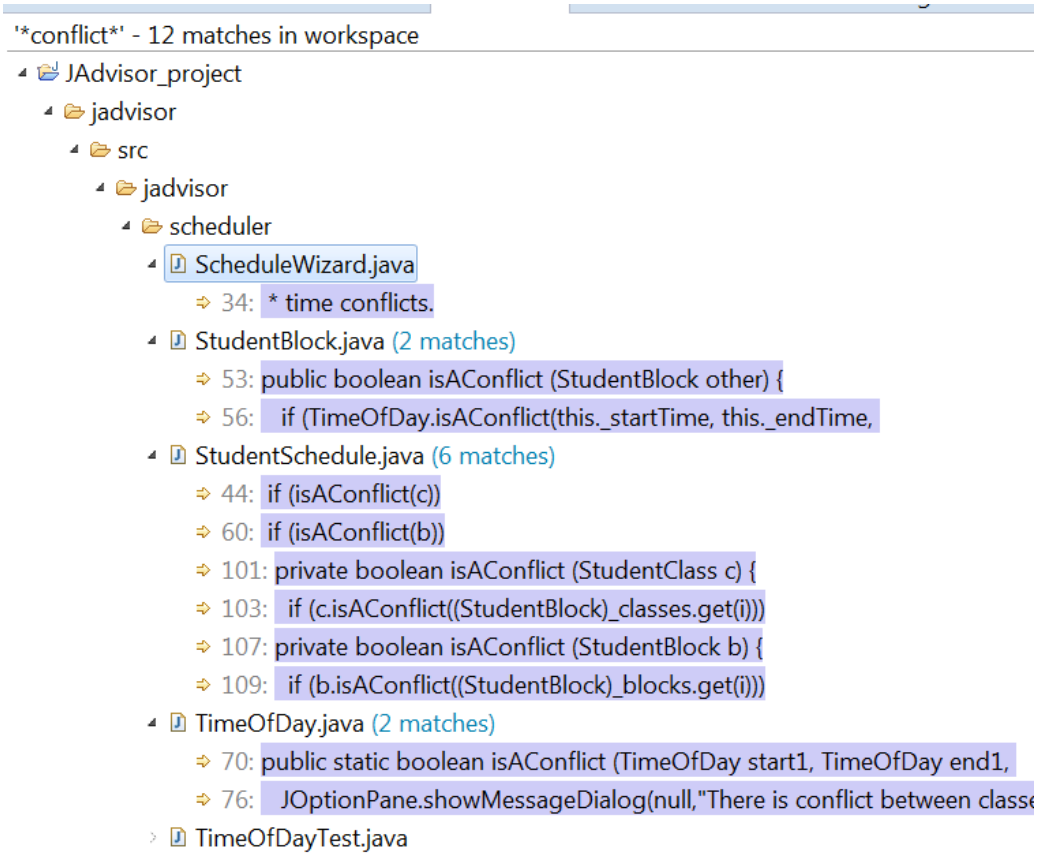


Figure 1. Results of the search command *conflict*

The relationship between the different files was studied to understand how they handle the input and outputs between each other. The lines of code shown in the search results were studied.

Code Files Visited

Code Files Visited / Inspected Only	Comments
4	The files that have the string 'conflict' were visited. Following are the files: ScheduleWizard.java | StudentBlock.java | StudentSchedule.java | TimeOfDay.java |

1.1.1 **Code file 1:** ScheduleWizard.java

Please provide a detailed journal entry describing the reason / motivation for visiting / inspecting the file. Also please describe how code file inspection was performed (i.e. tools used, terms searched, etc…)

This file had a matching string ‘conflict’ from the search command and was therefore visited for inspection. Using line 34 as a pointer from the search results, the specific line was studied in this file. It was found that this line was a comment section of the code and the file performed a task of ruling out classes that had time conflicts.

1.1.2 Code file 2: StudentBlock.java

Please provide a detailed journal entry describing the reason / motivation for visiting / inspecting the file. Also please describe how code file inspection was performed (i.e. tools used, terms searched, etc…)

Similar to the previous file, the reason to visit this file was that it had 2 matches for the string ‘conflict’. The relevant lines 53 and 56 were studied. This file takes the values of class time and dates and refers TimeOfDay.java file for inputs.

1.1.3 Code file 3: StudentSchedule.java

There were 6 matches of the string ‘conflict’ in this file. The study of this file revealed that if there is a conflict between the classes, then it displays an error message ‘Cannot Add’. All the lines related to conflict were studied.

1.1.4 Code file 4: TimeOfDay.java

There were 2 string matches in this file. Lines 70 and 76 from the search results were studied to find their contribution to file outputs.

1.2 Code Changes

Please provide a detailed journal entry describing how you went about performing the necessary coding changes for this change request.

Coding Change Summary

Code Files				
Visited	Changed	Added	Unchanged	Comments
4	1	-	3	TimeOfDay.java was edited.

Modified Code Files

#	Code File Name	Task	Lines of Code		
			Added	**Deleted**	**Total**
1	TimeOfDay.java	Allow the user to add the class in spite of a conflict.	7 lines added	None deleted	7 lines

1.2.1 Code file 1: TimeOfDay.java

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

The following were the changes performed in TimeOfDay.java file:

1. Added 4 new libraries. java.awt.Color library manages the change of color for the task that requires a notification in different color. java.util.*, javax.swing.UIManager and javax.swing.JOptionPane libraries are required to generate a pop up dialogue box.
2. Line 79 had a true value which would prevent adding classes in case of a conflict. The value was changed to false. With this change, the user is allowed to add the class irrespective of a conflict.
3. Line 74 to 76 were added. The code in these lines change the color to Red and generates a pop up message that reads “There is conflict between classes”.

1.3 Testing

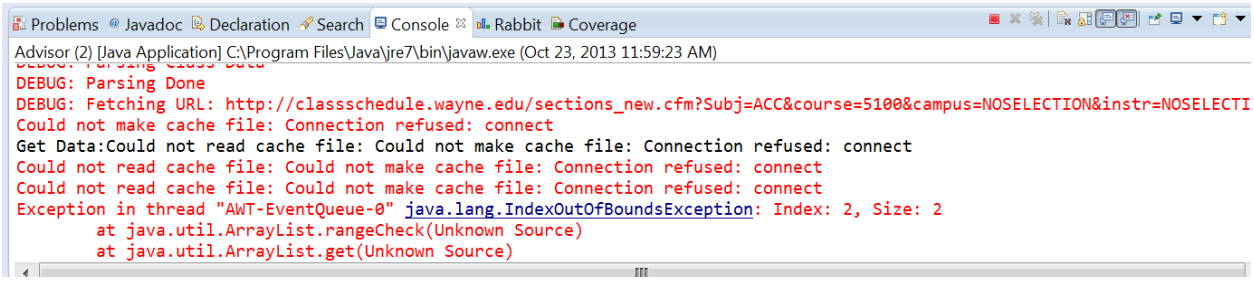
Please provide a detailed journal entry describing how you went about performing testing for this change request.

After modifying the TimeOfDay.java file, JUnit test case tool was implemented to run the ‘IsAConflict’ method for finding errors. A test case is shown in figure 2 below.

```
1  package jadvisor.scheduler;
2
3  import static org.junit.Assert.*;
6
7  public class TimeOfDayTest {
8
9      @Test
10     public void testIsAConflict() {
11         int num1 = 15;
12         int num2 = 15;
13         int num3 = 18;
14         int num4 = 18;
15         assertNotNull(num1);
16     assertNotNull(num2);
17     assertNotNull(num3);
18     assertNotNull(num4);
19
20         assertEquals(num1, num2);
21         assertTrue (num1< num3);
22         assertFalse(num2 > num4);
```

Figure 2. Test case using IsAConflict method

1) Test case was created to check for the correctness and the following errors were found and later debugged.



Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	TimeofDay.java	333	151	45.03%	1	4

1.4 Timing

Please provide a detailed journal entry describing how any of the supporting tools aided with completing this change request.

Eclipse: Eclipse is widely used and is the most developers start off. It supports several plug-ins like rabbit, Coverage, Eclemma and SVN which helped throughout the project in keeping track of time and check for the statements.

JUnit: Verifying the correctness of a program's behavior by inspecting the content of output statements using a manual testing, or more specifically, a visual process. Doing it manually was the tedious task. So JUnit helped in running the particular class or methods separately.

Timing Totals

Phase Name	**Time (hh:mm)**
Code Analysis	06.00
Code Change	03.00
Testing	01.00
Total Time	10.00

1.5 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

The Java tools Eclipse and JUnit were efficiently employed in this task to point out the areas to be studied and modified for completing the change request. The learning curve involved in understanding the software and the code was longer and led to greater amount of time spent on analyzing the code and shorter time for the actual modification of the code. The testing of code using JUnit yielded valuable insight into errors and further modifications of the code. The change required was successfully implemented as per the request. Overall, got a exposure to new environment.

CSC 6110 Project Results Log

Student #7

Change Request#: Zoom the text editor
Please provide a description of the change request / defect:

Under menu View, add two menu items “zoom+” and “zoom-” to scale the editors. At this stage, the scaling factors are not defined. The view should be able to be scaled multiple times.

Contents

1 Detailed Report

1.1 Code Analysis

Please provide a detailed journal entry describing how you went about identifying and determining which source code files needed to be modified in order to support this change request.
Describe the steps performed, how you went about inspecting / investigating the source code files and where to make the necessary changes.

1.1.1 Change strategy

The strategy for making this change was composed of the following steps:

1. Search the source code for adding the menu items “Zoom in” and “Zoom out”
2. Understand the existing code to know how the menu bar and their items work.
3. Run the program and manually test the change.
4. Change the program: add the menu items and make them functional (when they were clicked they showed a message in the program log).

5. Run the program and manually test the change.
6. Browser the code and understand the logic behind the editor's text area.
7. Change the code for making the zoom-in/zoom-out features work when the menu items were clicked.
8. Run the program and manually test the change.
9. Use abbot to make and run the GUI tests.

Steps 6, 7 and 8 where iteratively performed until the change worked. There were some additional activities after two or three iterations of these steps:

1. Find another text editor in Java which provides the zoom-in/zoom-out functionality. The other editor found was RText[1].
2. Understand how this feature works in RText.
3. Change the code of JEdit, based on the code of RText.

1.1.2 Change and Code Analysis

In general, for making this change these actions were performed (using eclipse features):

1. Searching text in the code.
2. Finding dependencies of classes, methods and attributes (clients and suppliers).
3. Debugging and running the program.

From the step 1 to 5 this is what was done to identify, understand and modify the code:

1. Search "menu" in all the source code.
2. Browse the search results. Only the results located in the org folder of jEdit were reviewed, as this folder contains the source code.
3. Visit the methods of the class GUIUtilities that deal with the menu bar loading.
4. Visit the class EnhancedMenu as this is instantiated in one of those methods.
5. Visit the class jEdit to figure it out where the menus' names are stored.
6. Inspect the file jedit_gui.props and change it to add the menu items in the View menu.
7. Run the program to check if the menu items effectively appear.
8. Search "Unknown action" in all the source code. When clicking one of the added menu items a message in the log appeared with the text "Unknown action: zoom-in".
9. Browse the results.
10. Visit the classes EditAction and DockableWindowFactory. The class DockableWindowFactory did not contain anything useful.
11. Visit the class ActionContext, since one of the methods of EditAction called the method ActionContext.getAction.
12. By browsing the dependencies of this class, the class jEdit was visited again.

13. Visit the file actions.xml. The file was changed by adding two actions called “zoom-in” and “zoom-out”
14. Two methods were added to the class EditPane: zoomIn and zoomOut. The methods contained only a line which displayed a message in the program’s log.
15. The program was run and some manual tests were performed.

For the rest this is what was done:

1. The class EditPane was inspected to understand how it worked. The class was inspected because some actions in actions.xml related to the view menu use this class.
2. These classes were inspected, based on dependencies analysis of the class EditPane:
    a. Buffer
    b. SyntaxStyle
    c. TextAreaPainter
    d. PaintText
    e. JEditTextArea
    f. TextArea
    g. ChunkCache
3. More than three runs and debugs of the program were performed to understand the code.
4. The methods zoomIn and zoomOut of the class EditPane were modified. Now, they changed the font of the classes TextArea and TextAreaPainter.
5. The program was run and some manual tests were performed.
6. The change partially worked: the caret and the dot in the text area changed their size but not the font.
7. These classes were inspected in detail: EditPane, TextArea and TextAreaPainter.
8. Some minor changes were performed. The program was run and debugged, but the changes didn’t work.
9. The inheritance of the class TextArea was changed to JTextArea, instead of JComponent, and some refactorings were necessary. This change was based on the RText’s code[2].
10. The program was run. This change didn’t work as the program logged some exceptions.
11. This last change was reverted.
12. The text “font” was searched in the package org.gjt.sp.jedit.textarea
13. Every result was read in the results window.
14. The following classes were inspected:
    a. ChunkCache
    b. FastRepaintManager
    c. Gutter
    d. TextArea
    e. TextAreaPainter
15. The method TextAreaPainter.setStyles and its clients were inspected. This method was reached because its comments had something related to the font.
16. The class SyntaxStyle was inspected.

[2] http://fifesoft.com/rtext/

17. The methods zoomIn and zoomOut of the class EditPane were modified. They now call the method TextAreaPainter.setStyles with new font styles. The size of the fonts was changed to the current size of the text area's font.
18. The program was run and some manual tests were performed. The change now works.

Code Files Visited

Code Files Visited / Inspected Only	Comments
20	In total, the following 20 code files were visited: 1. ActionContext.java 2. actions.xml 3. Buffer.java 4. ChunkCache.java 5. DockableWindowFactory.java 6. EditAction.java 7. EditPane.java 8. EnhancedMenu.java 9. FastRepaintManager.java 10. GUIUtilities.java 11. Gutter.java 12. JComponent.java 13. jEdit.java 14. jedit_gui.props 15. JEditTextArea.java 16. JTextArea.java 17. PaintText.java 18. SyntaxStyle.java 19. TextArea.java 20. TextAreaPainter.java

1.1.3 Code File: ActionContext.java

Motivation for inspection: one of the methods of EditAction called the method ActionContext.getAction.

Method used: dependencies browsing through reference searching (CTRL+G in Eclipse).

1.1.4 Code File: actions.xml

Motivation for inspection: this file contains the actions of the menu items. The file was changed by adding two actions called "zoom-in" and "zoom-out".

Method used: manual dependencies browsing.

1.1.5 Code File: Buffer.java

Motivation for inspection: this class represents a text buffer of a file. This class was inspected in order to understand its behavior, in relation with the change request.

Method used: dependencies browsing through reference searching (CTRL+G in Eclipse), from the class EditPane.

1.1.6 Code File: ChunkCache.java

Motivation for inspection: this class was inspected in order to understand its behavior, in relation with the change request. It was not relevant for the request. Also, the class was one of the results for the search "font" in the package org.gjt.sp.jedit.textarea.

Method used: text searching (CTRL+H in Eclipse) and dependencies browsing through reference searching (CTRL+G in Eclipse), from the class EditPane.

1.1.7 Code File: DockableWindowFactory.java

Motivation for inspection: this class was one of the results for the search "Unknown action". This class did not contain anything useful related to the change request.

Method used: text searching (CTRL+H in Eclipse).

1.1.8 Code File: EditAction.java

Motivation for inspection: this class was one of the results for the search "Unknown action".

Method used: text searching (CTRL+H in Eclipse).

1.1.9 Code File: EditPane.java

Motivation for inspection: this class represents the editor of a view in jEdit, including the text area. The class was inspected because some actions in actions.xml related to the view menu use this class. Two methods were added to this class: zoomIn and zoomOut.

1.1.10 Code File: EnhancedMenu.java

Motivation for inspection: the class is instantiated from one the methods inspected in the class GUIUtilities.

Method used: dependencies browsing through reference searching (CTRL+G in Eclipse), from the class GUIUtilities.

1.1.11 Code File: FastRepaintManager.java

Motivation for inspection: this class was one of the results for the search "font" in the package org.gjt.sp.jedit.textarea. This class is responsible for painting some specific elements in the text area.

Method used: text searching (CTRL+H in Eclipse).

1.1.12 Code File: GUIUtilities.java

Motivation for inspection: this class was one of the results for the search "menu". The methods that deal with the menu bar loading were reviewed.

Method used: text searching (CTRL+H in Eclipse).

1.1.13 Code File: Gutter.java

Motivation for inspection: this class was one of the results for the search "font" in the package org.gjt.sp.jedit.textarea. This class is the left side bar that displays the line numbers of the text area.

Method used: text searching (CTRL+H in Eclipse).

1.1.14 Code File: JComponent.java

Motivation for inspection: the inheritance of the class TextArea was changed to JTextArea, instead of JComponent. The class was inspected in order to understand its relationship with JTextArea.

Method used: text searching (CTRL+H in Eclipse) and dependencies browsing through inheritance hierarchy visualization (CTRL+G in Eclipse), from the class JComponent.

1.1.15 Code File: jEdit.java

Motivation for inspection: this class is the main class of jEdit and is responsible for loading properties configuration files of the application. This class was visited to figure it out where the menus' names and actions of menus were stored.

Method used: text searching (CTRL+H in Eclipse) and dependencies browsing through reference searching (CTRL+G in Eclipse), from the classes ActionContext and EnhancedMenu.

1.1.16 Code File: jedit_gui.props

Motivation for inspection: this file is used to store GUI label names, including menus and menu item. This file was changed to add the menu items in the View menu.

Method used: manual dependencies browsing.

1.1.17 Code File: JEditTextArea.java

Motivation for inspection: This class is a super class of a text area. This class was inspected in order to understand its behavior, in relation with the change request.

Method used: dependencies browsing through reference searching (CTRL+G in Eclipse), from the class EditPane.

1.1.18 Code File: JTextArea.java

Motivation for inspection: the inheritance of the class TextArea was changed to JTextArea, instead of JComponent. The class was inspected in order to understand its relationship with JComponent.

Method used: text searching (CTRL+H in Eclipse) and dependencies browsing through inheritance hierarchy visualization (CTRL+G in Eclipse), from the class JTextArea.

1.1.19 Code File: PaintText.java

Motivation for inspection: this class is responsible for painting some visual components of the text editor. This class was inspected in order to understand its behavior, in relation with the change request.

Method used: dependencies browsing through reference searching (CTRL+G in Eclipse), from the class EditPane.

1.1.20 Code File: SyntaxStyle.java

Motivation for inspection: a syntax style is basically a font with some visual attributes. This class was inspected in order to understand its behavior, in relation with the change request.

Method used: dependencies browsing through reference searching (CTRL+G in Eclipse), from the class TextAreaPainter.

1.1.21 Code File: TextArea.java

Motivation for inspection: this class is the text area of jEdit. This class was inspected in order to understand its behavior, in relation with the change request. Also, the class was one of the results for the search “font” in the package org.gjt.sp.jedit.textarea.

Method used: text searching (CTRL+H in Eclipse) and dependencies browsing through reference searching (CTRL+G in Eclipse), from the class EditPane.

1.1.22 Code File: TextAreaPainter.java

Motivation for inspection: this class paints all the elements in the text area. This class was inspected in order to understand its behavior, in relation with the change request. Also, the class was one of the results for the search “font” in the package org.gjt.sp.jedit.textarea.

Method used: text searching (CTRL+H in Eclipse) and dependencies browsing through reference searching (CTRL+G in Eclipse), from the class EditPane.

1.2 Code Changes

Please provide a detailed journal entry describing how you went about performing the necessary coding changes for this change request.

Coding Change Summary

Code Files				
Visited	**Changed**	**Added**	**Unchanged**	**Comments**
20	4	0	16	The process was described in the previous section.

Modified Code Files

#	**Code File Name**	**Task**	**Lines of Code**		
			Added	**Deleted**	**Total**
1	actions.xml	Addition of the menu item actions.	12	0	12
2	EditPane.java	Addition of the zoom-in/zoom-out logic.	45	0	45
3	jedit_gui.props	Addition of the menu items.	5	0	5
4	SyntaxStyle.java	Addition of a method.	7	0	7

1.2.1 Code file 1: actions.xml

The menu item actions zoom-in and zoom-out were added

1.2.2 Code file 2: EditPane.java

The methods zoomIn(), zoomOut() and updateFontSize(Font font) were added:

- zoomIn(): performs zoom-in of the current view of the text editor. The increment is 25 (point size). The upper size limit is 500.
- zoomOut(): performs zoom-out of the current view of the text editor. The decrement is 25 (point size). The lower size limit depends on the parameter “view.fontsize”.

The constant attributes fontSizeIncre (25) and MAX_SIZE (500) were added to control the bounds of the zooming.

1.2.3 Code file 3: jedit_gui.props

The actions zoom-in and zoom-out were added.

1.2.4 Code file 4: SyntaxStyle.java

The overridden method toString was added. This was added to visualize the style changing while testing.

1.3 Testing

Please provide a detailed journal entry describing how you went about performing testing for this change request.

Testing was performed in two ways:

1. Manually, by running the program and checking that the new functionality was working.
2. Automatically, by using JUnit. Unfortunately, although the plan was to use Abbot, the tool displayed some errors when running jEdit from the Abbot script editor (Figure 1), so it was not possible to use it. Thus, JUnit was used to implement and execute three test cases:
    a. Basic zoom-in/zoom-out: this test case consisted in emulating 3 zoom-in and 3 zoom-out operations, by directly calling the methods zoomIn and zoomOut of the class EditPane. At the end, the size of the font was the same than the size before the test was executed.
    b. Perform zoom-in until the upper size bound was reached: the expected behavior is that when the upper bound (500) is reached, the zoomIn method has no effect when executed.
    c. Perform zoom-out until the lower size bound was reached: the expected behavior is that the zoomOut method has no effect when the lower bound is reached.

```
[error] GUIUtilities 2013-10-20 10:11:55.496: Icon not found: new.gif
[error] GUIUtilities 2013-10-20 10:11:55.497: java.net.MalformedURLException: unknown protocol: jeditresource
[error] GUIUtilities 2013-10-20 10:11:55.497:  at java.net.URL.<init>(URL.java:592)
[error] GUIUtilities 2013-10-20 10:11:55.498:  at java.net.URL.<init>(URL.java:482)
[error] GUIUtilities 2013-10-20 10:11:55.499:  at java.net.URL.<init>(URL.java:431)
[error] GUIUtilities 2013-10-20 10:11:55.499:  at org.gjt.sp.jedit.GUIUtilities.loadIcon(GUIUtilities.java:173)
[error] GUIUtilities 2013-10-20 10:11:55.5:  at org.gjt.sp.jedit.GUIUtilities.init(GUIUtilities.java:1634)
[error] GUIUtilities 2013-10-20 10:11:55.501:  at org.gjt.sp.jedit.jEdit.main(jEdit.java:372)
[error] GUIUtilities 2013-10-20 10:11:55.501:  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

Figure 1. Error thrown by jEdit when was run from the Abbot script editor.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
	EditPane.java	40	40	100	0	0

1.4 Timing

Please provide a detailed journal entry describing how any of the supporting tools aided with completing this change request.

The timing was tracked manually and by using the eclipse plugin Rabbit.
Eclipse significantly supported searching of terms, dependency browsing, and coding.

Eclemma was used to analyze and calculate coverage of tests.
DiffStats was used to count the number of lines added in each modified file.
JUnit was used to perform automatic unit testing and regression testing.
Tortoise SVN was used to resolve conflict easily and fast.

All the tools helped to minimize development times.

Timing Totals

Phase Name	**Time (hh:mm)**
Code Analysis	06:20
Code Change	00:30
Testing	2:20
Total	**09:10**

1.5 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

The changed didn't work in the beginning because the method TextAreaPainter.setStyles was not checked and understood. For this reason, code analysis took more time than expected. In general, understanding the code is the most expensive task.

Regarding testing, as mentioned before, the usage of Abbot was unsuccessful due to some errors thrown by the tool. I tried to code the tests, instead of using the abbot script editor, but the tutorial followed was outdated, as some of the methods used in it were deprecated, and the main window of jEdit could not be displayed.

CSC 6110 Project Results Log

Student #8

Change Request : CREATION OF A SCHOOL ADAPTER

1 Detailed Report

1.1 Code Analysis

I have gone through the project source code. I found that in order to create an Adapter , there is need for creation of an object. So, I identified that the change can be done in ***Advisor.java*** *file and I have created a new object named* ***new WSUAdapter()*** *in the Advisor.java file. After the object creation, I studied the default as well as NCSU and UNC adapters . Then I have downloaded the Wayne state Class schedule and implemented the changes in the* ***WSUAdapter.java*** *file where the selection of courses, classes and look up of time table can be done.*

Code Files Visited

Code Files Visited / Inspected Only	Comments
Advisor.java	Created an object named WSUAdapter() .
SchoolAdapter.java	Studied the methods used in the project.
DefaultSchoolAdapter.java NCSU.java UNC.java	Inspected the files and studied about their implementation.
WSUAdapter.java	Downloaded the information about the Wayne State University and implemented the changes.
WSUAdapterTest.java	Assertions are written in the file and JUNIT testing is done.

1.1.1 Code file 1

Firstly, I wanted to create a School Adapter. So, I researched every file in the project to see where I can make change to attain it. I found ***Advisor.java****, wherein I have created an object.*

Then it has appeared on the interface namely WsuSchoolAdapter . Tortoise SVN has been used to commit the changes.

1.1.2 Code file 2

Secondly, I wanted to download the information about the wayne state school and made the changes in the **WSUAdapter.java** file . Before doing it I inspected the flow of code in the file and made necessary modifications.I used DiffStats to check the changes happened. TortoiseSVN has been used to add and commit the file.

1.2 Code Changes

Please provide a detailed journal entry describing how you went about performing the necessary coding changes for this change request.

Coding Change Summary

Code Files				
Visited	Changed	Added	Unchanged	Comments
2	2	0	0	WSUAdapter.java (changed) Advisor.java (changed)

Modified Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
	Advisor.java	MODIFICATION	1	0	1
	WSUadapter.java	MODIFICATION	14	0	14

1.2.1 Code file 1

Changes were made in Advisor.java file.I have created an object named WSUAdapter and modifications were seen on the user interface where WsuSchoolAdapter has been shown.

1.2.2 Code file 2

Changes were made in WSUAdapter.java file. Here I have downloaded the information about the school courses and schedule .Necessary code modifications have been done to show the working of the adapter.

1.3 Testing

I have created JUnit assertions to check the validation of the statements in the file. assertTrue(), assertFalse(), assertNotNull() and assertArrayEquals() methods have been used in the WSUAdapterTest.java file and checked for working of all the functionalities.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
	WSUAdaptertest.java	16465	76	0.5	0	0

1.4 Timing

Timing Totals

Phase Name	Time (hh:mm)
Code Analysis	12:00
Code Change	3:00
Testing	4:00
Total Time	19:00

1.5 Conclusions

Firstly, I looked into all the files and have studied the flow the project. After analysis of code, I have created an Object and changed the UI to show WSUSchoolAdapter implementation. After executing the JAdvisor project , WsuSchoolAdapter has been implemented .

CSC 6110 Project Results Log

Student #9

Change Request#: Group 4 - "Modify the splash window".

Please provide a description of the change request / defect:

Currently the splash window of jEdit is a static picture. Add the names and emails of your group members to it. And add moving text as the same effect shown in "About jEdit" dialog. Adjust the scrolling speed so that all text can be shown.

1 Detailed Report

1.1 Code Analysis

The change request was to add names and emails of the group members and roll the names on the SplashScreen as shown in **About jEdit**. The following procedure was followed to accomplish the change:

1. The first step was to search for "splashscreen" in *eclipse*. The search returned all the files containing Splashscreen in their files. On browsing the search result under org folder of jedit, *Splashscreen.java* was found. Only the result in the org folder has to be checked as it contains the source code.
2. Since the changes had to be done similar to the **About jEdit**, a search was given with "aboutjedit" as search key in eclipse. The search indicated, there were no files with aboutjedit in them.
    - Next a search was given on one of the names getting rolled in **About jEdit**. The search result returned the file *jedit_gui.props*. The names were assigned to about.text.
    - So the next search was for given on "about.text" and "Animation". The result lead to *AboutDialog.java*.
3. On analyzing the AboutDialog.java code, the rolling mechanism was understood.
4. Using *AboutDialog.java* as reference, changes were made to the *SplashScreen.java*
5. A method AnimationThread was added to *Splashscreen.java*, similar to the one present in *AboutDialog.java* to roll the names on the screen.
6. The change was implemented twice.

- The first time when the change was done, the names were placed in *jedit_gui.props* with splash.text as variable. It was called in splashscrren.java in the method splashscreen().
- The program was throwing error on executing it.
- Further analysis was done using the debugger in eclipse. The togglebreakpoints were placed in the main method (jEdit.java) , GUIUtilities.java and Splashscreen.java.
- It was discovered that the file jedit_gui.java is read after splash is called. So the names could not be added to jedit_gui.java. The names were then added to the SplashScreen.java file.

7. Changes to the method paintComponent was made to include the names of the team members and their Email IDs.
8. The method splashScreen() calls the animation thread to roll the names.

Code Files Visited

Code Files Visited / Inspected Only	Comments
5	The following files visited: 1. SplashScreen.java 2. AboutDialog.java 3. jedit_gui.props 4. jEdit.java 5. GUIUtilities.java

1.1.1 Code file: SplashScreen.java

1. The change request involved, changes in the splashscreen. So a search was given in the eclipse with the name “Splashscreen”. It returned all the files containing the name splashscreen in it and the file *Splashscreen.java*.
2. Further, an analysis was made so as to determine the task performed by the file.
   - The file displays the version, and the progress of the jEdit startup.
   - The flow of the methods paintcomponent, advance, advance(string), logAdvanceTime and splashscreen() was understood.

1.1.2 Code file: AboutDialog.java

1. Since the requested change was to roll the names as in **About jEdit**. A search was made in the eclipse with the name “aboutjedit”. On a search on one of the names rolling on the screen of **About jEdit**, it returned *jedit_gui.props.* Next the repository was searched with search key as “Animation”. On browsing the search result the file *AboutDialog.java* was obtained.
2. The file was analyzed for the animation to understand the workings of the method AnimationThread().

1.1.3 Code file: jedit_gui.props

The search on one of the names in the About Jedit lead to jedit_gui.props. This file contains all the names of the rolled in about jedit and contains java version.

1.1.4 Code file: jEdit.java

To determine the flow of the program jedit, a search on “main(String”. On browsing the search result, jEdit. Java is determined as to containing the main method. This function calls GUIUtilities.java.

1.1.5 Code file: GUIUtilities.java

From above, the file is analyzed. It is this file which calls the method splashscreen().

1.2 Code Changes

Coding Change Summary

Code Files				
Visited	Changed	Added	Unchanged	Comments
5	1	0	4	The file Splashscreen.java was edited

Modified Code Files

#	Code File Name	Task	Lines of Code		
			Added	**Deleted**	**Total**
1	SplashScreen.java	The names have to be displayed in the splashscreen giving a rolling effect	97	0	97

1.2.1 Code file 1: *Splashscreen.java*

Basically, the *Splashscreen.java* was changed keeping AboutDialog.java as reference. The following are the changes made to the *Splashscreen.java*

1. A method AnimationThread() is added to roll the names in the splashscreen – It rolls the names as in, it changes the position of the names.
2. A method addNotify() is added to start the thread. – start the thread
3. A method removeNotify() is added to kill the thread. – kills the thread
4. The names are added to paintComponent to display them – It displays the names of the team members.
5. The Animation thread is called from splashScreen method.

So the When the SplashScreen is called, the names are displayed giving a rolling effect.

1.3 Testing

There are two types of testing done:

- First, a manual testing is done. On executing the program the names are displayed on the splash screen with a rolling effect.
- Since the Splashscreen is neither a functionality nor a GUI, a simple test in done using JUnit wherein the methods in Splashscreen are called using the object type Splashscreen and checked if the methods are successfully executed.

```
public class SplashScreenTest {

@Test
public void testSplashScreen() {
      SplashScreen splash = new SplashScreen();
}


@Test
public void testDispose() {
      SplashScreen splash = new SplashScreen();
      splash.dispose();
}
}
```

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	SplashScreen.java	548	521	95.1	0	0

1.4 Timing

The supporting tool Rabbit was used to time the duration spent on each individual file. But it is difficult to provide a distinctive timeline for Code analysis and Code Change. Below is the timeline approximated for each of the Phase.

Timing Totals

Phase Name	Time (hh:mm)
Code Analysis	5:10
Code Change	2:00
Testing	00:05
Total Time	7:15

1.5 Conclusions

The first change done to the code didn't work because the names were added to the jedit_gui.props file and were called from splashscreen(). On further analysis and using debug functionality in eclipse, it came to light that jedit_gui.props is read after the splashscreen is called. So, it was concluded that the names have to be added to the splashscreen.java itself.

Regarding testing, since the change is neither a functionality nor a GUI application, a test is done to check if the methods in the splashscreen thrown an error or not.

Overall, it can be concluded that code analysis is the most expensive task followed by code change.

CSC 6110 Project Results Log

Student #4

Change Request#: 3

Please provide a description of the change request / defect:

In the feature "Autogenerate BibTeX keys" keys are generated in this format [author][year]. Make a change so that the BibTeX keys have the timestamp added to the format like this [author][year]_[hhmmss] (e.g.Brooks2010_083025).

1 Phase Report

1.1 Concept Location

Please provide a detailed journal entry describing how you went about performing concept location for this change request.

Concept Location Summary

Code Files			Comments
Visited	Propagating	Unchanged	
#	#	#	

Concept Location Code Files Visited

#	Code File Name	Tool Used	Located?	Comments

1.2 Impact Analysis

Please provide a detailed journal entry describing how you went about performing impact analysis for this change request.

Impact Analysis Summary

Code Files					**Comments**
Visited	Impacted	Propagating	Unchanged	Not Visited	
10	3	1	7	0	

Impact Analysis Code Files Visited

#	**Code File Name**	**Tool Used**	**Impacted?**	**Comments**
1	*BibtexParser.java*	*Eclipse search*	*Yes*	*Added a few LOC to change the format of the timestamp*
2	*JabRefPreferences.java*	*Eclipse Search*	*Yes*	*Changed the format of the timestamp*
3	*BibtexDatabase.java*	*Eclipse Search*	*No*	
4	*EntryEdit.java*	*Eclipse Search*	*No*	
5	*DatePickerButton.java*	*Eclipse Search*	*No*	
6	*LabelMaker.java*	*Dependency search*	*No*	
7	*DefaultLabelPatterns.java*	*Dependency Search*	*Yes*	Uncommented 10 LOC for the default pattern of the bibtex key

1.3 Prefactoring

Please provide a detailed journal entry describing how you went about performing prefactoring for this change request.

Prefactoring Summary

Code Files					
Visited	Changed	Added	Propagatin	Unchange	Added to Changed Set
#	#	#	#	#	#

Prefactoring Code Files

#	**Code File Name**	**Task**	**Lines of Code**		
			Added	**Deleted**	**Total**

Prefactoring was not required for this change.

1.3.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this files and its new / modified responsibilities

1.4 Actualization

Please provide a detailed journal entry describing how you went about performing actualization for this change request.

Actualization Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
10	3	0	0	8	2

Actualization Code Files

#	**Code File Name**	**Task**	**Lines of Code**		
1	BibtexParser.java	Addition of few lines	**Added**	**Deleted**	**Total**
2	JabRefPreferences.java	Modification	3	0	3
3	DefaultLabelPatterns.java	Modification		0	

1.4.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

BibtexParser.java:

In the BibtexParser.java file, code was added to append the time to the end of the bibtex key using the java.Util.Date package and constructor date which calculates the hours, minutes and seconds and displays in the format hhmmss using the methods getHours(),getMinutes() and getSeconds().

JabRefPreferences.java

Modified the format of the timestamp variable to yyyy.MM.dd_hhmmss.

DefaultLabelPatterns.java

Uncommented the lines of code to display the changed format for the display of the bibtex key to [author][year]_[timestamp].

1.5 Postfactoring

Please provide a detailed journal entry describing how you went about performing postfactoring for this change request.

Postfactoring Summary

Code Files					
Visited	Changed	Added	Propagatin	Unchange	Added to Changed Set
0	0	0	0	0	0

Postfactoring Code Files

#	**Code File Name**	**Task**	**Lines of Code**		
			Added	**Deleted**	**Total**

Postfactoring was not necessary as the change involved adding only a few lines of code.

1.6 Verification

Please provide a detailed journal entry describing how you went about performing verification for this change request.

Statement Verification

#	**Code File Name**	**Coverage of Application**			**Tests Failed**	**Bugs Found**
		Total Statemen	**Covered Statemen**	**%**		
1	Bibtexparser.java	3			0	0

Manual testing was done by importing a bib file and selecting an entry to autogenerate bibtex keys and the code worked perfectly and the time was appended to the end of the key.

1.7 Timing

Please provide a detailed journal entry describing how the supporting tools aided with completing this change request.

Timing Totals

Phase Name	Time (hh:mm)
Concept Location	04:00
Impact Analysis	01:00
Prefactoring	00:00
Actualization	02:00
Postfactoring	00:00
Verification	00:30

1.8 Conclusions

Please provide a detailed journal entry summarizing completing this change request.
The time format hhmmss was added to the bibtex key using the standard java util package with the date constructor and the methods to retrieve hours, minutes and seconds. The concept location was time consuming as there were a lot of classes that was associated with the date formatter and picking the right place for the change consumed time.

Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
3		10	5	3	0	3	0	3

CSC 6110 Project Results Log

Student #5

Change Request #: 1

Please provide a description of the change request / defect:

"Consolidating BibTeX files

Input: a folder, output: a .bib file

Scan recursively the input folder and its sub folders, find all BibTeX files, parse these files to BibTex Databases, merge these databases, remove conflicts if any and save the consolidated databases to a output file

Create GUI for this functionality

1 Phase Report

1.1 Concept Location

Please provide a detailed journal entry describing how you went about performing concept location for this change request.

I first started analysis and then I made the first jabref propagating and the I grep query search with word "" and words like "" it showed an results of both I narrowed my search to BasePanel.java then I visited it gave me the EOL marker and before it I called the method.

Concept Location Summary

Code Files			**Comments**
Visited	Propagating	Unchanged	
#2	#1	#0	It showed basepanel with EOL marker is disabled before running the file and when we open the database it enables with use EOL marker

Concept Location Code Files Visited

#	Code File Name	Tool Used	Located?	Comments
1	Jabref.java	Jripples		
2	BasePanel.java	Jripples – query	yes	

1.2 Impact Analysis

Please provide a detailed journal entry describing how you went about performing impact analysis for this change request

Impact Analysis Summary

Code Files					Comments
Visited	Impacted	Propagating	Unchanged	Not Visited	
#3	#	#	#3	#	

Impact Analysis Code Files Visited

#	Code File Name	Tool Used	Impacted?	Comments
1	Overlaypanel.java	Jripple-IA	No	
2	PreviewPanel	Jripple-IA	No	
3	ColorSetupPanel	Jripple-IA	No	

1.3 Prefactoring

Please provide a detailed journal entry describing how you went about performing prefactoring for this change request.

Prefactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#	#	#	#	#	#

Prefactoring Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total

1.3.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

Filehandling.java when I select the file it tells type of file and path.

1.4 Actualization

Please provide a detailed journal entry describing how you went about performing actualization for this change request.

Actualization Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#2	#1	#1	#	#	#1

Actualization Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
1	Filehandling .java	Gui for opening the file	1		

1.4.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

Here I made object call for basepanel.java to filehandling.java which handles gui for scanning files to consolidated file

1.5 Postfactoring

Please provide a detailed journal entry describing how you went about performing postfactoring for this change request.

Postfactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#	#	#	#	#	#

Postfactoring Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total

1.5.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this files and its new / modified responsibilities

1.6 Verification

Please provide a detailed journal entry describing how you went about performing verification for this change request.

Verification is I did manual by using cmd by testing each modified class calling from main test class.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		

1.7 Timing

Please provide a detailed journal entry describing how the supporting tools aided with completing this change request.

Timing Totals

Phase Name	Time (hh:mm)
Concept Location	2:00
Impact Analysis	:30
Prefactoring	-
Actualization	1:00
Postfactoring	-
Verification	1:00

1.8 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

The tricky situation was creating a button using windowbuilder in eclipse but I used the button from open database button for searching the file for consolidating file .Snapshot of the file and I also used plugin zotero-better-bibtex-master /combine.rb file to parse, merge databases and make consolidated file.

Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
	2	#2	#3	#	#	#2	#	#7

CSC 6110 Project Results Log

Student #6

Change Request 3 : Unicity of bibTeX key

Please provide a description of the change request / defect:

In the feature "Autogenerate BibTeX keys" keys are generated in this format [author][year].

Make a change so that the BibTeX keys have the timestamp added to the format like this [author][year]_[hhmmss] (e.g. Brooks2010_083025).

1 Phase Report

1.1 Concept Location

Please provide a detailed journal entry describing how you went about performing concept location for this change request.

Concept Location Summary

Code Files			Comments
Visited	Propagating	Unchanged	
06	-	02	

Concept Location Code Files Visited

#	Code File Name	Tool Used	Located?	Comments
1	BibtexEntry.java	Jripples	No	
2	Util.java	Jripples	Yes	
3	DefaultLabelPatterns.java	Jripples	Yes	
4	BasePanel.java	Jripples	No	
5	BibtexParser.java	Jripples	Yes	
6	JabRefPreferances	Jripples	yes	

1.2 Impact Analysis

Please provide a detailed journal entry describing how you went about performing impact analysis for this change request.

Impact Analysis Summary

Code Files					Comments
Visited	Impacted	Propagating	Unchanged	Not Visited	
03	03	-	-	-	

Impact Analysis Code Files Visited

#	Code File Name	Tool Used	Impacted?	Comments
1	JabRefPreferences.java	JRipples	Yes	
2	Util.java	Jripples	Yes	
3	BibtexParser.java	Jripples	Yes	

1.3 Prefactoring

Please provide a detailed journal entry describing how you went about performing prefactoring for this change request.

1) This change request has not gone through this stage.

Prefactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#	#	#	#	#	#

Prefactoring Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total

1.3.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this files and its new / modified responsibilities

1.4 Actualization

Please provide a detailed journal entry describing how you went about performing actualization for this change request.

Actualization Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
05	3	-	-	02	03

Actualization Code Files

#	Code File Name	Task	Lines of Code		
			Added	**Deleted**	**Total**
1	Util.java	Added LOC	5	-	5
2	JabRefPreferences.java	Modified	1	-	1
3	DefaultLabelPatterns.java	Added	1	-	1

1.4.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

1) *Util.java : Few lines of code were added to display Time in hh:mm:ss format.*
2) JabRefPreferences.java: The default timestamp format was modified to YYYY:MM:DD_hh:mm:ss.
3) DefaultLabelPatterns : The new line was added to display timestamp in the given format.

1.5 Postfactoring

Please provide a detailed journal entry describing how you went about performing postfactoring for this change request.

1) This change request has not gone through this stage.

Postfactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#	#	#	#	#	#

Postfactoring Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total

1.5.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

1.6 Verification

Please provide a detailed journal entry describing how you went about performing verification for this change request.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	Util.java	578	289	35%	2	-
2	DefaultLabelPatterns.java	346	230	25%	-	-
3	JabRefPrefrences.java	489	290	30%	-	-

1.7 Timing

Please provide a detailed journal entry describing how the supporting tools aided with completing this change request.

Timing Totals

Phase Name	Time (hh:mm)
Concept Location	06.30
Impact Analysis	00.30
Prefactoring	-
Actualization	00.45
Postfactoring	-
Verification	00.25

1.8 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

Initially it took more time find the concept location, even though JRipples tool shows up the classes that have to be visited but checking each and every file was time consuming. But it helped in finding the classes that were going to be impacted by other classes. Once the class was found that is going to be changed then it was easy to track which other classes are going to change. The code is added to display time along with date.

Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
		06	03	03	-	03	-	3

CSC 6110 Project Results Log

Student #7

Change Request#: Auto-update timestamp on edit

Please provide a description of the change request / defect:

The current format of the timestamp when adding an entry is [year].[month].[day] (e.g. 2013.11.18). Make a change so that the timestamp has the format [year][month][day].[hh][mm][ss] (e.g. 20131118.083025) and it is auto updated when the button auto is clicked.

Table of Contents

1 Phase Report

1.1 Concept Location

Please provide a detailed journal entry describing how you went about performing concept location for this change request.

Concept Location was done using Eclipse searching tools and the features of JRipples, including the Grep searching feature.

The first step for performing concept location was to understand the current functionality related to the change request. For this, I ran the program; I used it and made some tests. Some doubts came out about the change request which were clarified by the teacher assistant.

After understanding the functionality, I started JRipples and marked the JabRef class as propagated. Instead of looking and checking every class as Next I searched for "timestamp". Most of search results marked as next were reviewed, according to the number of matches; the classes with more matches were inspected first. In concrete, the following classes were inspected (in the following order):

1. Util: it contains three methods called setAutomaticFields which modify the timestamp field.
2. JabRefPreferences: it contains all the preferences of the application, including the ones that define the default owner and timestamp. This class was modified.
3. BibtextFields: it models the fields of a bibtex entry.
4. Globals: it manages global application features.
5. BibtexEntry: it is the class that models a bibtext entry.

Before inspecting the class JabRefPreferences, I ran the application and I checked the preferences window of the application. In the general tab of the preferences, the user can change the timestamp format so I changed it to "yyyyMMdd.hhmmss". After restarting the application, I added a new entry which had the timestamp with the format specified. At this point, I knew the change had to do with the application preferences. To confirm this, I changed the format in the constructor of JabRefPreferences to check if the timestamp changed, but it didn't. After making debugging and testing, I realized that when the preferences are changed in the Jabref's preference dialog, those are stored in the Windows registry, in HKEY_CURRENT_USER\Software\JavaSoft\Prefs\net\sf\jabref, and the format defined in the application

preferences is the one taken by the application; if there is no format in the registry then the default format is taken. So, what I did was to remove the JabRef registry entry and the change worked. Finally, the class JabRefPreferences was marked as Located in JRipples.

For the second part of the change request, I first searched for *owner*, using the JRipples GREP search. The following classes were inspected with no success in finding the change location: GeneralTab, PrefsDialog, BasePanel, BibtexFields and JabRefFrame. These classes where inspected because they were marked as next. After this, I searched for *auto*, and the classes AbstractAutoCompleter and JabRefFrame were inspected with no success. Then I searched for *owner* again, but this time using the Eclipse searching feature. In this case, the first class inspected was ImportInspectionDialog and finally I found the location: the method getExtra of class EntryEditor. This class was not found using JRipples because it was not marked as Next and I just focused on Next classes.

Concept Location Summary

Code Files			**Comments**
Visited	Propagating	Unchanged	
13	2	0	

Concept Location Code Files Visited

#	Code File Name	Tool Used	Located?	Comments
1	Util.java	JRipples	No	
2	JabRefPreferences.java	JRipples	Yes	
3	BibtextFields.java	JRipples	No	
4	Globals.java	JRipples	No	
5	BibtexEntry.java	JRipples	No	
6	GeneralTab.java	JRipples	No	
7	PrefsDialog.java	JRipples	No	
8	BasePanel.java	JRipples	No	

9	BibtexFields.java	JRipples	No	
10	JabRefFrame.java	JRipples	No	
11	AbstractAutoCompleter.java	JRipples	No	
12	ImportInspectionDialog.java	Eclipse search	No	
13	EntryEditor.java	Eclipse search	Yes	

1.2 Impact Analysis

Please provide a detailed journal entry describing how you went about performing impact analysis for this change request.

Impact Analysis was performed manually because the amount of classes marked as Next in JRipples was 174 and the changes seemed not to impact a lot of functionality. The analysis resulted in no classes impacted.

Impact Analysis Summary

Code Files					**Comments**
Visited	Impacted	Propagating	Unchanged	Not Visited	
2	0	0	0	173	

Impact Analysis Code Files Visited

#	Code File Name	Tool Used	Impacted?	Comments
1	EntryEditorTab.java	Eclipse	No	
2	Utils.java	Eclipse	No	

1.3 Prefactoring

Please provide a detailed journal entry describing how you went about performing prefactoring for this change request.

The change didn't require prefactoring.

1.4 Actualization

Please provide a detailed journal entry describing how you went about performing actualization for this change request.

The actualization was divided in two steps:

1. Change the format of the timestamp field
2. Add the new functionality: when the Auto button is pressed the timestamp field should be updated to the current timestamp.

Actualization Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
2	2	0	0	0	0

Actualization Code Files

#	Code File Name	Task	Lines of Code			
			Added	Modified	Deleted	Total
1	JabRefPreferences.java	The timestamp format was changed.	0	1	0	1
2	EntryEditor.java	The behavior of the Auto button was changed	8	0	0	8

1.4.1 Code file 1: JabRefPreferences.java

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

The change of this file was done in the constructor. The format of the “timeStampFormat” property was changed to "yyyyMMdd.HHmmss".

1.4.2 Code file 1: EntryEditor.java

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

An attribute that stores the timestamp text field was created. In addition, the method actionPerfomed of the Auto button was modified so the timestamp field is updated together with the owner field.

1.5 Postfactoring

Please provide a detailed journal entry describing how you went about performing postfactoring for this change request.

The postfactoring was performed for testing purposes (see the subsection Verification). Two attributes their setters and getters were created in EntryEditor.java.

Postfactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
1	1	0	0	0	0

Postfactoring Code Files

#	Code File Name	Task	Lines of Code			
			Added	Modified	Deleted	Total
1	EntryEditor.java	Creation of two additional attributes	30	4	0	34

1.5.1 Code file 1: EntryEditor.java

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

The following attributes, together with their getters and setters were created: onwerField

and ownerAutoBtn. The method EntryEditor.getExtra was modified to set those new attributes.

1.6 Verification

Please provide a detailed journal entry describing how you went about performing verification for this change request.

The test case developed was simple: just assert the timestamp and owner fields were empty before the user clicked the auto button and those fields were not after the user clicked the button. In addition, the timestamp text was asserted to be correct, according to the format "yyyyMMdd.HHmmss".

The tests case was implemented using JUnit, and was based on already implemented test: AutoCompleterTest The following coverage resulted only by the execution of the implemented test case.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
1	JabRefPreferences.java	3761	2714	72.2	0	0
2	EntryEditor.java	3612	1506	41.7	0	0

1.7 Timing

Please provide a detailed journal entry describing how the supporting tools aided with completing this change request.

The timing was tracked manually.
Eclipse significantly supported searching of terms, dependency browsing, and coding.
JRipples was used to track some of the files visited.
Eclemma was used to analyze and calculate coverage of tests.
DiffStats was used to count the number of lines added in each modified
file. JUnit was used to perform automatic unit testing and regression

testing.

Tortoise SVN was used to resolve conflicts easily and fast.

Timing Totals

Phase Name	Time (hh:mm)
Concept Location	01:31
Impact Analysis	00:16
Prefactoring	00:00
Actualization	00:05
Postfactoring	00:15
Verification	00:51
Total	02:58

1.8 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

Although the change was simple, concept location was a bit hard. But I realized that it was my mistake, because I was paying attention only to Next classes in JRipples. In addition, it seemed that verification was going to be tough with Abbot, but at the end I realized I didn't need to use Abbot.

Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
1	Auto- update timestamp on edit	13	2	2	0	2	1	2

CSC 6110 Project Results Log

Student #8

Change Request#: 4

Auto-update timestamp on edit

The current format of the timestamp when adding an entry is [year].[month].[day] (e.g. 2013.11.18). Make a change so that the timestamp has the format [year][month][day].[hh][mm][ss] (e.g. 20131118.083025) and it is auto updated when the button auto is clicked.

1 Phase Report

1.1 Concept Location

Please provide a detailed journal entry describing how you went about performing concept location for this change request.

I Used eclipse search approach to find the java files containing the word "timestamp", in that matches found Util.java was the file where format for the timestamp was found .

```
dateFormatter = new SimpleDateFormat(format);
```

Here is the location and I needed to change the timestamp format.

Concept Location Summary

Code Files			Comments
Visited	Propagating	Unchanged	
2	#	1	It showed the code for format of the timestamp. It was initially in yyyyMMdd format. Concept was found in Util.java

Concept Location Code Files Visited

#	Code File Name	Tool Used	Located?	Comments
	Util.java	Eclipse search	Located	Format for timestamp is located in the file.
	FileListEditor.java	Eclipse search	inspected	Auto button implementation found here

1.2 Impact Analysis

Please provide a detailed journal entry describing how you went about performing impact analysis for this change request.

Actually I have found the Change location by using Eclipse search approach. So, impact analysis was not much looked up.

1.3 Prefactoring

Please provide a detailed journal entry describing how you went about performing prefactoring for this change request.

This change request has not gone through this stage as the change has to be done within the existing code.

1.4 Actualization

Please provide a detailed journal entry describing how you went about performing actualization for this change request.

Actualization Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
2	1	0	#	1	1

Actualization Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
	Util.java	modification	1	0	1
	FileListEditor.java	visited	0	0	0

1.4.1 Code file 1

Please provide a detailed journal entry describing the changes performed on this file and its new / modified responsibilities

Util.java: Timestamp format is found. Changed it to `dateFormatter = new SimpleDateFormat("yyyyMMdd.HHmmss");`

FileListEditor.java : auto button functionality is inspected.

1.5 Postfactoring

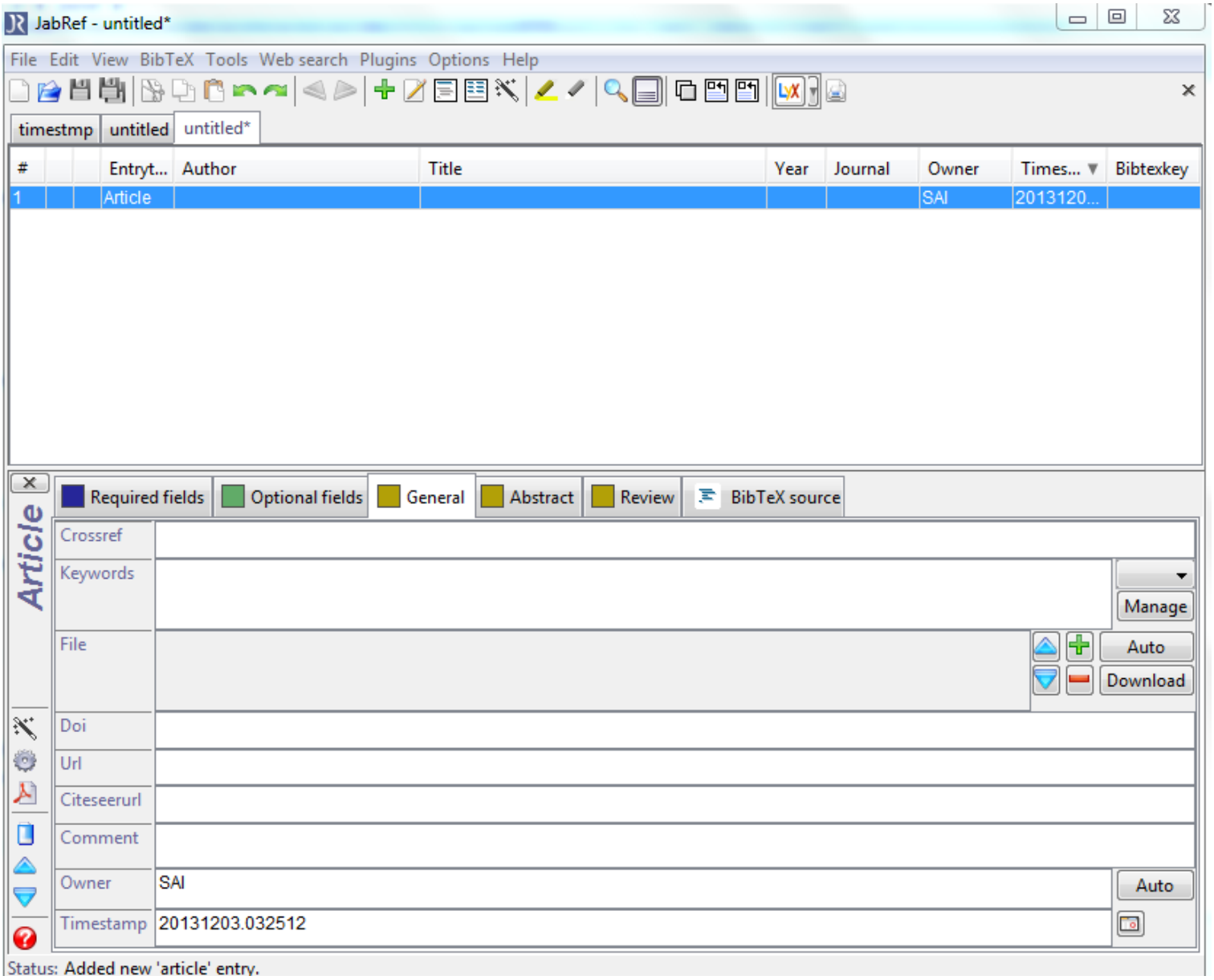
Please provide a detailed journal entry describing how you went about performing postfactoring for this change request.

There was no further clean up necessary when the changes were implemented.

1.6 Verification

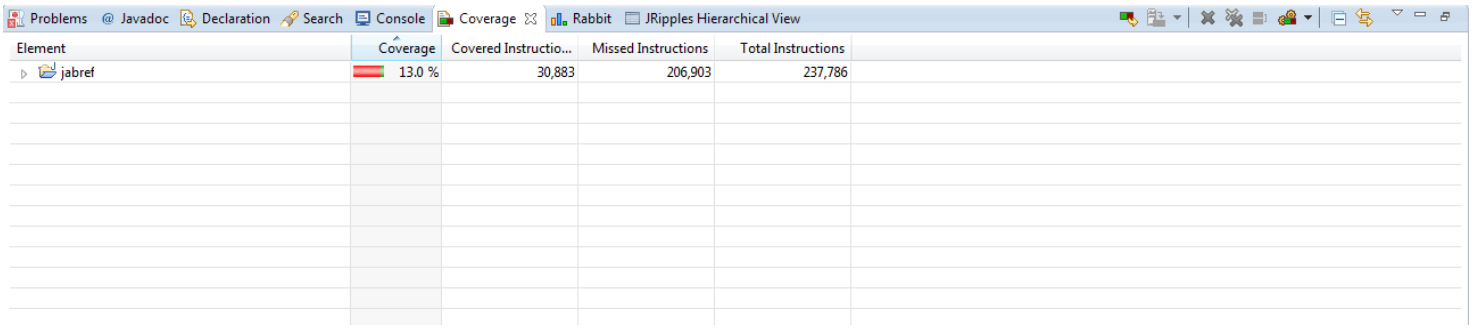
Please provide a detailed journal entry describing how you went about performing verification for this change request.

The code was manually checked for the correctness and the following errors were found and later debugged. Every file was tested manually to find the errors and it was debugged. The change request was implemented.



Screenshot showing the changed timestamp format

Coverage:



1.7 Timing

Please provide a detailed journal entry describing how the supporting tools aided with completing this change request.

Eclipse search approach helped me to locate the file where is needed to be done. I have changed the code and completed my change request.

Timing Totals

Phase Name	**Time (hh:mm)**
Concept Location	00:20
Impact Analysis	00:00
Prefactoring	00:00
Actualization	00:35
Postfactoring	00:00
Verification	00:15

1.8 Conclusions

Please provide a detailed journal entry summarizing completing this change request.

With the help of Eclipse search tool, I took less time in finding the file where the change has to be made. I used the keyword “Timestamp” in the search process. Then I found Util.java, where format for timestamp is present. It was in yyyyMMdd format. I have changed it to yyyyMMdd.hhmmss format. Thus I have implemented the change request.

CSC 6110 Project Results Log

Student #9

Change Request#: 2

Please provide a description of the change request / defect:

Shrinking BibTeX files

Input: a .bib file, a folder containing .tex files, **output**: a new .bib file.

Create GUI for this functionality.

1 Phase Report

1.1 Concept Location

The request was to add a new gui for this functionality to compare tex files and remove any redundant items from the files and merge them. The concept location is identified using JRipples. The main search identifies JabRef containing the main class. The classes marked as Next are scanned through. The file JabRefFrame.java is selected. The file is located as concept location where the option of comparing files is provided. Similar analysis provided JabRefFrame.java and JabRefPreference.java.

A new file OpenTex.java is added which takes .tex files as input from the user and parses it.

Concept Location Summary

Code Files			Comments
Visited	Propagating	Unchanged	
3	1	2	JabRefPreference.java JabRefFrame.java JabRef.java

Concept Location Code Files Visited

#	Code File Name	Tool Used	Located?	Comments
1	JabRefFrame.java	JRipples		
2	JabRefFrame.java	JRipples		
3	JabRef.java	JRipples		

1.2 Impact Analysis

The file impacted by the change is OpenTex.java. JRipples was used to identify the dependencies.

Impact Analysis Summary

Code Files					Comments
Visited	Impacted	Propagating	Unchanged	Not Visited	
1	1	0		#	

Impact Analysis Code Files Visited

#	Code File Name	Tool Used	Impacted?	Comments
	-	-	-	-

1.3 Prefactoring

The software didn't have to be reorganized to make the actualization.

Prefactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#	#	#	#	#	#

Prefactoring Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
	-	-	-	-	-

1.4 Actualization

Please provide a detailed journal entry describing how you went about performing actualization for this change request.

Actualization Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
4	3	1	1	0	1

Actualization Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
1	BasePanel.java	Added actions	3	0	3
2	JabRefFrame.java	Added an option in tool menu	3	0	3
3	JabRefPreference.java	Binding the option provided in the JabRefFrame	1	0	1
4	OpenTex.java	Takes files from the users and parses it	123	0	123

1.4.1 BasePanel.Java

Added an action to open the class OpenTex.

1.4.2 JabRefFrame.java

It added an option in the tool menu to compare files.

1.4.3 JabRefPreference.java

The file was modified to add a binding to the compare file option in the tool menu.

1.4.4 OpenTex.Java

The file was created to take .tex files from the users and compare them and delete redundant key and merge the files in .bib format.

1.5 Postfactoring

The software didn't need any postfactoring

Postfactoring Summary

Code Files					
Visited	Changed	Added	Propagating	Unchanged	Added to Changed Set
#	#	#	#	#	#

Postfactoring Code Files

#	Code File Name	Task	Lines of Code		
			Added	Deleted	Total
-	-	-	-	-	-

1.6 Verification

The verification was done by two methods. First testing was done manually asked the users to provide the files from the system. Second the test was done using JUnit. It accepts the files from the user.

Statement Verification

#	Code File Name	Coverage of Application			Tests Failed	Bugs Found
		Total Statements	Covered Statements	%		
	OpenTexTest.java	123	91	73	0	0

1.7 Timing

To locate concept location took time as the code had to be analyzed in detail.

Timing Totals

Phase Name	Time (hh:mm)
Concept Location	2:00
Impact Analysis	00:45
Prefactoring	00
Actualization	00:45
Postfactoring	00
Verification	30:00

1.8 Conclusions

The change request was complicated as the file had to take from the users compare the keys in the files and merge the files and compile the .tex file to get .bib file. I took time to analyze the code as it was complicated for. JRipples helped in identifying the concept location faster.

Since the change included creation of a new file, the major coding was done in the new generated file.

Code File Summary

#	Change	Number in Code Files						
		Visited Concept Location	Estimated Impact Set	Changed Set	Added during			Total Project
					Pre	Act	Post	
		#	#	#	#	#	#	#
	changed	3	0	0	0	3	0	4
	added	1	1	1	0	1	0	

APPENDIX D: Post – Experiment Questionnaires

This appendix contains the post - experiment questionnaires of the study. Similarly to appendix B and appendix C, three less experienced and three more experienced participants have their post - experiment questionnaires displayed in this thesis.

All the post - experiment questionnaires are available online via the following link: https://drive.google.com/folderview?id=0BwkmElTjUf2qSmROWW5OMFZDSGs&usp=sharing.

CSC 6110 Student Post-Experiment Questionnaire

Student # 4

Based on your recent participation / experience in the CSC 6110 Research project please answer the following questions to the best of your ability.

1. Did performing the change request by following the “Phased Software Change Model” (PSCM) as a part of Stage 2 of the course assignment work better for you than not following a specific process like you did during your Stage 1 change request?

◯ Yes ◉ No

Additional comments

2. Please check / mark which box best describe whether the PSMC approach help or hinder your performance during stage 2 change requests compared conducting the stage 1 change request.

◯ very unproductive
◉ somewhat / slightly unproductive
◯ Neutral
◯ somewhat / slightly productive
◯ very productive

Additional comments

3. Do you feel that following the PSCM approach save you time? If so where specifically?

◯ Yes ◉ No

If Yes, which phases?

Concept Location

◯ Yes ◯ No

Impact Analysis

◯ Yes ◯ No

Refactoring (Pre / Post)

◯ Yes ◯ No

Actualization

◯ Yes ◯ No

Verification

◯ Yes ◯ No

Additional comments

4. Please rate the difficulty of performing the change request in the two stages

	1	2	3	4	5
Stage 1	◯	◉	◯	◯	◯
Stage 2	◯	◯	◉	◯	◯

Any additional comments

CSC 6110 Student Post-Experiment Questionnaire

Student # 5

Based on your recent participation / experience in the CSC 6110 Research project please answer the following questions to the best of your ability.

1. Did performing the change request by following the “Phased Software Change Model” (PSCM) as a part of Stage 2 of the course assignment work better for you than not following a specific process like you did during your Stage 1 change request?

◉ Yes ○ No

Additional comments

2. Please check / mark which box best describe whether the PSMC approach help or hinder your performance during stage 2 change requests compared conducting the stage 1 change request.

○ very unproductive
○ somewhat / slightly unproductive
◉ Neutral
○ somewhat / slightly productive
○ very productive

Additional comments

3. Do you feel that following the PSCM approach save you time? If so where specifically?

◉ Yes ○ No

If Yes, which phases?

Concept Location

◉ Yes ○ No

Impact Analysis

○ Yes ◉ No

Refactoring (Pre / Post)

○ Yes ○ No

Actualization

◉ Yes ○ No

Verification

◯ Yes ◯ No

Additional comments

4. Please rate the difficulty of performing the change request in the two stages

	1	2	3	4	5
Stage 1	◯	◯	◉	◯	◯
Stage 2	◯	◯	◯	◯	◉

Any additional comments

CSC 6110 Student Post-Experiment Questionnaire

Student # 6

Based on your recent participation / experience in the CSC 6110 Research project please answer the following questions to the best of your ability.

1. Did performing the change request by following the “Phased Software Change Model” (PSCM) as a part of Stage 2 of the course assignment work better for you than not following a specific process like you did during your Stage 1 change request?

◉ Yes ○ No

Additional comments

2. Please check / mark which box best describe whether the PSMC approach help or hinder your performance during stage 2 change requests compared conducting the stage 1 change request.

○ very unproductive
○ somewhat / slightly unproductive
○ Neutral
○ somewhat / slightly productive
◉ very productive

Additional comments

3. Do you feel that following the PSCM approach save you time? If so where specifically?

◉ Yes ○ No

If Yes, which phases?

Concept Location

◉ Yes ○ No

Impact Analysis

◉ Yes ○ No

Refactoring (Pre / Post)

○ Yes ◉ No

Actualization

○ Yes ◉ No

Verification

◯ Yes ◉ No

Additional comments

4. Please rate the difficulty of performing the change request in the two stages

	1	2	3	4	5
Stage 1	◯	◯	◯	◉	◯
Stage 2	◯	◯	◯	◯	◉

Any additional comments

CSC 6110 Student Post-Experiment Questionnaire

Student # 7

Based on your recent participation / experience in the CSC 6110 Research project please answer the following questions to the best of your ability.

1. Did performing the change request by following the “Phased Software Change Model” (PSCM) as a part of Stage 2 of the course assignment work better for you than not following a specific process like you did during your Stage 1 change request?

(●) Yes ( ) No

Additional comments

2. Please check / mark which box best describe whether the PSMC approach help or hinder your performance during stage 2 change requests compared conducting the stage 1 change request.

( ) very unproductive
( ) somewhat / slightly unproductive
( ) Neutral
(●) somewhat / slightly productive
( ) very productive

Additional comments

In the first stage, I used a slight variation of PMSC.

3. Do you feel that following the PSCM approach save you time? If so where specifically?

(●) Yes ( ) No

If Yes, which phases?

Concept Location

(●) Yes ( ) No

Impact Analysis

( ) Yes (●) No

Refactoring (Pre / Post)

( ) Yes (●) No

Actualization

( ) Yes (●) No

Verification

◯ Yes ◉ No

Additional comments

I think PSCM doesn’t necessarily saves you time in impact analysis, but maybe the accuracy of finding the change set is greater than with other approaches.

4. Please rate the difficulty of performing the change request in the two stages

	1	2	3	4	5
Stage 1	◯	◯	◉	◯	◯
Stage 2	◯	◉	◯	◯	◯

Any additional comments

In general, the change requests were easy. For me, the tasks that consumed most of the time were concept location and verification. The former because you need to understand the code and this takes time and the second one because you need make sure the program is behaving as expected; for this you need program the tests and in some of the cases there were no tests at all.

CSC 6110 Student Post-Experiment Questionnaire

Student # 8

Based on your recent participation / experience in the CSC 6110 Research project please answer the following questions to the best of your ability.

1. Did performing the change request by following the “Phased Software Change Model” (PSCM) as a part of Stage 2 of the course assignment work better for you than not following a specific process like you did during your Stage 1 change request?

◉ Yes ○ No

Additional comments

Performing change request in stage 2 was better because sometimes we used dependency analysis ,which helped in saving time.

2. Please check / mark which box best describe whether the PSMC approach help or hinder your performance during stage 2 change requests compared conducting the stage 1 change request.

○ very unproductive
○ somewhat / slightly unproductive
○ Neutral
○ somewhat / slightly productive
◉ very productive

Additional comments

3. Do you feel that following the PSCM approach save you time? If so where specifically?

◉ Yes ○ No

If Yes, which phases?

Concept Location

◉ Yes ○ No

Impact Analysis

○ Yes ◉ No

Refactoring (Pre / Post)

◉ Yes ○ No

Actualization

◉ Yes ○ No

Verification

◉ Yes ○ No

Additional comments

4. Please rate the difficulty of performing the change request in the two stages

	1	2	3	4	5
Stage 1	◉	○	○	○	○
Stage 2	○	○	◉	○	○

Any additional comments

Change request in stage 2 was little bit tough to implement.

CSC 6110 Student Post-Experiment Questionnaire

Student # 9

Based on your recent participation / experience in the CSC 6110 Research project please answer the following questions to the best of your ability.

1. Did performing the change request by following the “Phased Software Change Model” (PSCM) as a part of Stage 2 of the course assignment work better for you than not following a specific process like you did during your Stage 1 change request?

◉ Yes ○ No

Additional comments

2. Please check / mark which box best describe whether the PSMC approach help or hinder your performance during stage 2 change requests compared conducting the stage 1 change request.

○ very unproductive
○ somewhat / slightly unproductive
◉ Neutral
○ somewhat / slightly productive
○ very productive

Additional comments

3. Do you feel that following the PSCM approach save you time? If so where specifically?

◉ Yes ○ No

If Yes, which phases?

Concept Location

◉ Yes ○ No

Impact Analysis

○ Yes ◉ No

Refactoring (Pre / Post)

○ Yes ◉ No

Actualization

○ Yes ◉ No

Verification

◉ Yes ○ No

Additional comments

4. Please rate the difficulty of performing the change request in the two stages

	1	2	3	4	5
Stage 1	○	◉	○	○	○
Stage 2	○	○	○	○	◉

Any additional comments

The difficulty level for change requests varied, while some got multiple requirements in one change request others got very simple request. With my experience in Java, the change requests I got were tough for stage 2. My only issue was with variation in the difficulty level.
With that said I also learned many stuffs which will be useful in the future.

References

[1] apache.org. "Apache™ Subversion®," 1/24/2014; http://subversion.apache.org/.

[2] K. Beck, *Extreme Programming Explained*, Reading, MA: Addison Wesley, 2000.

[3] K. Beck, "JUnit," 2011.

[4] J. Buckner *et al.*, "JRipples: A Tool for Program Comprehension during Incremental Change." pp. 149-152.

[5] M. H. Claes Wohlin, Kennet Henningsson, “Empirical Research Methods in Software Engineering,” *Lecture Notes in Computer Science*, pp. 7-23, 2003.

[6] T. A. Corbi, “Program Understanding: Challenge for the 1990s,” *IBM Systems Journal,* vol. 28, no. 2, pp. 294-306, 1989.

[7] B. Dit *et al.*, “Feature location in source code: a taxonomy and survey,” *Journal of Software: Evolution and Process,* vol. 25, no. 1, pp. 53-95, 2013.

[8] C. Dorman, “An Experience Report Of The Solo Iterative Process,” Computer Science, Wayne State University, Detroit, MI, 2011.

[9] C. Dorman, "DiffStats," 2011.

[10] C. Dorman, and V. Rajlich, "Software Change in the Solo Iterative Process: An Experience Report." pp. 22-30.

[11] D. V. Duine, “Personal Software Process & Team Software Process: An overview,” in Seattle Eastside Area SPIN, 2006.

[12] eclemma.org. "Eclemma - Java Code Coverage for Eclipse "; http://www.eclemma.org/.

[13] eclipse.org. "Rabbit 1.2.1," http://marketplace.eclipse.org/content/rabbit.

[14] extremeprogramming.org. "The Rules of Extreme Programming," 1/24/2014; http://www.extremeprogramming.org/rules.html.

[15] extremeprogramming.org. "Extreme Programming: A gentle introduction " 1/24/2014; http://www.extremeprogramming.org/rules.html.

[16] P. Grubb, and A. A. Takang, *Software Maintenance: Concepts and Practice*, Second ed.: World Scientific, 2003.

[17] I. N. H. Takeuchi, “The New New Product Development Game,” *Harvard Business Review*, pp. 137-146, 1986.

[18] D. E. Harter, M. S. Krishnan, and S. A. Slaughter, “Effects of Process Maturity on Quality, Cycle Time, and Effort in Software Product Development,” *Manage. Sci.,* vol. 46, no. 4, pp. 451-466, 2000.

[19] W. S. Humphrey. "The Team Software Process (TSP)," http://www.sei.cmu.edu/reports/00tr023.pdf.

[20] ISO/IEC, "Software Engineering — Software Life Cycle Processes — Maintenance," 2006.

[21] S. Jarzabek, *Effective Software Mantenance and Evolution: A Reuse-based Approach*: CRC Press, 2007.

[22] jEdit.org. "jEdit," 1/28/14; http://www.jedit.org/FAQ/general.html.

[23] C. Jones, *Software Engineering Best Practices*, p.^pp. 11: McGraw-Hill, 2009.

[24] B. A. Kitchenham, Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El-Emam, K., and Rosenberg, J., “Preliminary Guidelines for Empirical Research in Software Engineering,” 2001.

[25] S. Lehnert, “A Taxomony for Software Change Impact Analysis,” *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pp. 41-50, 2011.

[26] M. Höst, B. Regnell, and C. Wohlin, “Using Students as Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment,” *Empirical Software Engineering: An International Journal,* vol. 5, no. 3, pp. 201-214, 2000.

[27] M. Petrenko *et al.*, “Teaching Software Evolution in Open Source,” *IEEE Computer,* vol. 40, no. 11, pp. 25-31, 2007.

[28] A. A. R. E. Jeffries, & C. Hendrickson, *Extreme programming installed*, Boston, MA: Addison-Wesley Longman, 2000.

[29] V. Rajlich, and P. Gosavi, “Incremental change in object-oriented programming,” *Software, IEEE,* vol. 21, no. 4, pp. 62-69, 2004.

[30] V. Rajlich, *Software Engineering: The Current Practice*, Boca Raton, FL: CRC Press, 2012.

[31] Sarah Boslaugh, and P. A. Watters, *Statistics in a Nutshell*: O'Reilly Media, 2008.

[32] H. B. Shah, C. Gorg, and M. J. Harrold, “Understanding exception handling: Viewpoints of novices and experts,” *Software Engineering, IEEE Transactions on,* vol. 36, no. 2, pp. 150-161, 2010.

[33] J. S. Sutherland, K., "The Scrum Guide™," scrum.org, ed., 2013.

[34] The TortoiseSVN Team, "TortoiseSVN," 2011.

[35] tortoisesvn.net. "TortoiseSVN," 1/24/2014; http://tortoisesvn.net/about.html.

[36] T. Wall, "Abbot Java GUI Test Framework," 2008.

[37] T. Wall. "Getting Started with the Abbot Java GUI Test Framework," 1/24/2014; http://abbot.sourceforge.net.

[38] J. Wang *et al.*, “An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions,” in 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 213 - 222.

ABSTRACT

CASE STUDY OF PHASED MODEL FOR SOFTWARE CHANGE IN A MULTIPLE-PROGRAMMER ENVIRONMENT

by

YOANN SENIN

August 2014

Advisor: Dr. Václav Rajlich

Major: Computer Science

Degree: Master of Science

The aim of this thesis is to perform an empirical study comparing programmers completing software changes assisted by the recently published software process Phased Model for Software Change (PMSC) to those completing software changes without any assistance. There have been numerous researches on software change, but most of them focused more on individual phases of the software change process in lieu of the software change process as a whole. For that reason, this thesis explores the impact of the PMSC process on programmers' performance. The subjects of this study are graduate students with different level of experience.

The results of the experiment show that following the PMSC process improves the performance of both less experienced and more experienced programmers by reducing the amount of time spent to complete software changes by about half. This improvement is noticeable in both code analysis and code implementation activities. We also talk about ways to refine PMSC.

AUTOBIOGRAPHICAL STATEMENT

YOANN SENIN

Yoann Senin received a Master of Science in Computer Science at University Félix-Houphouët-Boigny, former University of Cocody (Ivory Coast), in 2010. After working as a Web Developer at Afrinal, he moved to Michigan to pursuing his education. His is currently completing another Master of Science in Computer Science at Wayne State University with a concentration on software engineering. During his curriculum at Wayne State University, Senin worked as a Web Developer at the Center for Urban Studies and also as a Graduate Teaching Assistant at the Department of Computer Science of the same institution. His versatile experience both in web development and desktop application development make him proficient in PHP, JSP, Java, MySQL, Oracle, HTML, CSS, XML technologies, and several other tools related to the listed technologies.