ANALYSIS ON A RELEASE HISTORY DATABASE TO ASSIST MANAGEMENT OF THE SOFTWARE MAINTENANCE

Mohammad Saeed Bohlooli

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirement
For the Degree of Applied Science in Software Engineering
Concordia University
Montreal, Quebec, Canada
April 2013

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Mohammad Saeed Bohlooli

Entitled: Analysis on a Release History Database to Assist Management of the Software Maintenance

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Software Engineering

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. R. Jayakumar

_____ Examiner
Dr. Yuhong Yan

_____ Examiner
Dr. Nikolaos Tsantails

_____ Supervisor
Dr. Constantinos Constantinides

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

Abstract

Analysis on a Release History Database to Assist Management of the Software Maintenance

Mohammad Saeed Bohlooli

Software maintenance is the most time consuming activity in the life cycle of software. Software maintenance suffers from missed deadlines and from being over budget. Managers usually pay more attention to development than to maintenance: for example, they prefer to assign senior developers to the development phase tasks and neglect maintenance ones. Managers have difficulty identifying problems, and their causes, in maintenance.

This thesis presents techniques for analysis on the proposed release history database to provide metrics for improvement of the maintenance phase. The proposed release history database is enriched by valuable data that comes from an issue tracking system, code repository, and time entry system. The proposed release history database and the analysis of the data contained there provides metrics which allow maintainers to find risky and time-consuming codes, recommending maintenance team and maintenance location and a suggestions for the future of the maintenance. Automation is also provided as a proof of concept through a prototypical tool.

Acknowledgments

This thesis would not have been possible without the help, support and patience of my supervisor, Dr. Constantinos Constantinides whose advice, encouragement and unsurpassed knowledge contributes to my graduate work and experience. I would have been lost without him.

I also would like to thank the members of Software Maintenance and Evolution Research Group, Parisa Mirshams, Michel Parisien and my speical thanks for Zohreh Sharafi providing valuable comments and their friendly help for my research work.

I owe my loving thanks to the most influential people in my life: my wife, Sahar Tamadon, and my parents for unconditional support and encouragement to pursue my interest.

Contents

List of Figures

Chapter 1

Introduction

As software systems evolve over time and keep changing, the quality of the software system declines. Therefore, software maintenance activities are required to preserve the quality of the software. Software maintenance is the modification of the software product after delivery to correct faults, improve performance, and to adapt the product to a new environment [ANSI/IEEE standard 1219-1998]. It is reported that 50% to 70% of the overall cost of a software is dedicated to maintenance [36]. Initiating, evaluating, and controlling changes to the software system are the main activities of software maintenance. We need to manage the evolution of the software and track any change in it [33]. Software configuration management(SCM) started in the late 1960s and is a process for controlling changes in the life cycle of software development as well as the maintenance life cycle [55].

1.1 Objective and goals of this dissertation

We know that software maintenance suffers from missed deadlines and exceeded budgets. Our objective is to improve management in software maintenance by analyzing a custom release history database in order to help managers to meet deadlines and stay on budget. To meet this objective, first we import data from different resources into our proposed release history database, including release histories, the development environment, and the maintenance environment. In the next step, we show how analysis and queries on our proposed release history database will help maintainers and managers to improve software maintenance.

1.2 Organization of the dissertation

The remainder of this dissertation is organized as follows: In chapter 2, we provide the necessary background for the thesis. In Chapter 3, we discuss the problem and motivation behind our research. In Chapter 4, we discuss our proposal and in Chapter 5 we describe how we can build our proposed *release history database* and then in Chapter 6, we discuss our methodology for achieving our goals. In Chapter 7, we describe a case study to demonstrate how our proposed approach can be applied to a software which is in maintenance. In Chapter 8, we discuss related works to our work. In Chapter 9, we describe the automation and tool support. We list our conclusion and provide recommendations for future research work in Chapter 10. Finally, in Appendix A, we present the survey details we have performed for the thesis.

Chapter 2

Background

In this chapter, we provide some necessary theoretical background on software maintenance, management in software maintenance, issue tracking systems, and release history databases.

2.1 Software maintenance

Software maintenance is defined as the modification of a software product performed after delivery in order to [37]:

Adaptive maintenance: Adapt to a changing/new environment.

Perfective maintenance: Prevent latent faults from becoming failures or to improve software attributes.

Corrective maintenance: Repair known problems.

Preventive maintenance: Prevent latent faults from becoming operational faults.

Every day the number of software systems moving into their maintenance phase grows. This is a costly phase of the software life-cycle [35]. Studies in historic data in legacy software systems show that the reasons for changes are divided into three categories: adding new features, fixing bugs, and refactoring codes to accommodate future changes [42].

2.2 Software maintenance: time and cost

Two of the major concerns in the software maintenance phase are how to estimate the cost of maintenance releases in software systems [4] and how to stay within the estimated cost. Many software companies see the software maintenance phase as the most resource and time consuming [1]. Studies show most of the time and cost in software development is spent on the maintenance of application. The maintenance phase involves fixing *bugs* which are signaled by customers or the Quality Assurance (QA) team, and new *change requests* which come from stackholders. Software maintenance is the most difficult phase because of its cost and its error-prone nature.

Records show that 50-80 percent of the total budget of a software department or software company is spent on maintenance; as a matter of fact, maintenance activities are their most time and cost consuming activity[35]. Software organizations are always interested in improving the maintenance phase in order to reducing costs. This is one of the reasons why several organizations are interested in outsourcing

their maintenance activities. In most cases, software maintenance is executed across the world as Global Software Development and Maintenance (GSDM) to benefit form time and cost reductions [38].

2.3 Management in software maintenance

There are a couple of challenges in the maintenance phase. Software tool support for maintenance achieves more productivity and improves the quality of software maintenance, but most tools are specialized for code analysis rather than for improving management and process of maintenance [35]. Analyses of software maintenance states show that insufficient management is the major problem in the maintenance process. This causes low productivity; thus, the maintenance phase will not be on *schedule* or on *cost*.

We know that existing information in large software systems is valuable data for adjusting the future software process [19].

2.4 Software configuration management

Software configuration management is a type of discipline that shapes control, status accounting and control audits to a given software product [8]. In the following sections we describe different components/activities of software configuration management in software maintenance.

2.4.1 The issue tracking system

The *issue tracking system* in the software development process is a system which tracks issues, bugs and change requests from the starting point until the process ends with a release. Most software development companies have issue tracking systems in place. Today, software developments are distributed all over the world and most software companies use web based issue tracking systems which are accessible around the world.

The data in this system is the most important available management information source for the improvement of software process decisions. The information in the *issue tracking system* shows the full history of bugs, issues, and change requests.

Moreover, open source and even commercial software companies use open bug repositories which let both end users and developers to follow how the software is moving forward [26].

Some papers call the *issue tracking system* the *defect tracking system*, and it is also sometimes referred to as the *bug tracking system*, but we prefer to call it the *issue tracking system* as it is more general and covers bugs, issues, as well as change requests.

Software development teams usually have their own workflow in an *issue tracking system* based on their relation with customer, the team size, and the team's distribution around the world. Figure 1 presents a minimized workflow for bugs, issues, and change requests during the maintenance. This workflow's states are the most common states in different workflows that we have studies in non-commercial and

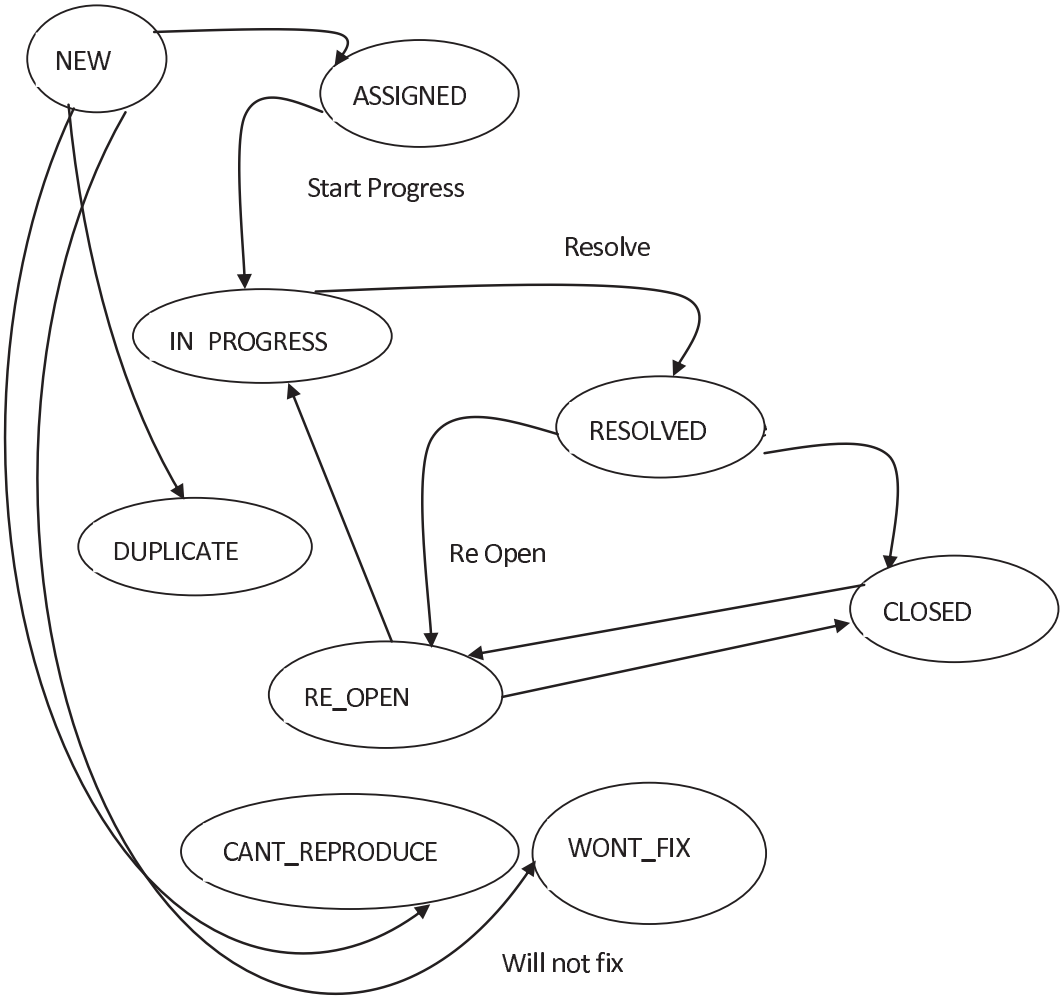

Figure 1: A typical workflow of an issue tracking system.

commercial tools.

As illustrated in Figure 1, when bugs are discovered by testers or customers, the status of the bug is indicated as NEW state. Then, the team leader or project manager will make a decision about what the next state of bug or issue should be. If the team leader decides to resolve this issue in the next release, then he/she will move it to the next release and will assign it to a developer; the status will be changed to ASSIGNED. In case it was not a bug, it will be changed to the WONT_FIX state, or in case they couldn't reproduce it, it will be moved to the CANT_REPRODUCE state. If it is already on the bug list, the team leader will move it to the DUPLICATE state.

Developers query the *issue tracking* system on a daily basis to find the tasks that are assigned to them [62]. A developer will start working on a bug, a task, or an issue and will change its status to IN_PROGRESS. A developer will change its status to RESOLVED when it is done. In the next step, testers will try to verify the bug fix, and the state will be switch to IN_REVIEW. In case the issue is verified by the tester, it will be moved to the CLOSED status. If it is not resolved, the tester will put the bug into the RE_OPEN status , and it will be seen by the team leader or the project manager. That is a typical workfow; teams develop workflow schemas based on the nature of the project, the team size and the Service Level Agreement (SLA).

2.4.2 Code repository

While developing software and during the maintenance phase, the code always changes. Therefore, we need to control these changes effectively. We control changes in code through *Revision Control*, *Version Control* and *Source Control* management of changes in the source code. Managing *Version Control* is a part of *Software Configuration Management*. In software development, the term *code repository* refers to codes which are stored in the *version control* system. The *Code repository* has all of the software's code.

Version control systems distinguish files by numbers which are called *revision numbers*. These indicate the version of the file. Each new version of a file is stored in the code repository gets a unique revision number. We have *release numbers* which indicates the release of the software. Every *release* represents a snapshot of the latest version of the codes in the code repository. We are using the Subversion (SVN) [59] in our case study. After releasing a new version of a software, a symbolic number as representive of the software release will be assigned to the *revision number* of the current files [20].

Any software in the repository is managed in its trunk and branches. Figure 2 illustrates the trunk, branches, and release numbers. Different companies choose different strategies for the management of trunks and branches. The most popular one is shown in Figure 2. Most of the times, the trunk points to the ongoing stream of codes, and for every release, the release management team usually makes a branch or patch. As shown in 2, there is one branch for 1.0 which would be merged with

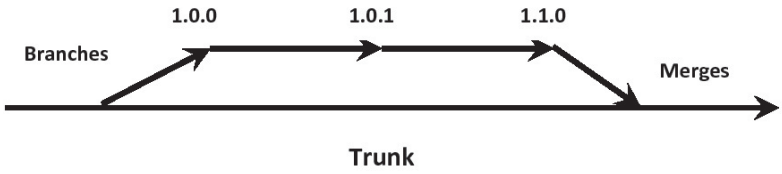

Figure 2: Trunk and branches in version control

the trunk in the future. Every release has its own *branch*. Every release has its own information which is hidden in the version control. Code repositories allows us have differences between two different branches and then we will be able to store their differences. We can compare the newly released software against the last release. The output of differences between the two releases is in text plain format. The output file shows any code added or removed code from a file. In addition, a reference to the *issue number* exists in every difference between a file in two different releases.

Generally speaking, a *code repository* keeps a record of every change of codes which were ever checked in to the repository. We can therefore explore the history of any of the resources which exist in the repository.

There are two types of version control systems. The first type is the *centralized version control system* (CVCS) and the second type is *distributed version control system* or *decentralized version control system* (DCVS).

In the *centralized version control system*, all the version control functions are centralized on a server and there is one instance repository that is placed in the central server. The most commonly used centralized version control systems are CVS

and Subversion (SVN) [13].

In terms of comparing centralized and decentralized version control systems, in a *centralized version control system*, backupping data and maintaining servers is easier because everything is in one place and because the system contains multiple repositories for multiple projects. *Centralized version control systems* allow developers to have available and consistent systems. However, when we consider scalability and distribution, the *centralized version control systems* are not good options.

*Distributed version control system*s work almost like *centralized version control systems*, except that instead of one central repository, there are servers of repositories. In *distributed version control system*s, we have multiple instances of the repository. A new repository instance will be created using the *clone* operation and most operations interact with the local repository, not with a network repository. Then, developers could synchronize two repository instances using *push* command. This operation sends a copy of some of the *change sets* into a remote instance. Two instances probably will not be identical after running the *push* command. In order to fully synchronize, developers should *pull* everything from the remote instance, and then *push* everything to the remote instance. *Distributed version control systems* are more popular in the open-source community.

In 2000, CollabNet Inc. decided to start developing the Subversion (SVN) project. Subversion was released as open-source software. SVN is a good option for companies with few locations. In SVN, like other centralized version control systems, every project is stored in a central repository. The repository has all files of the project

with all changes and history information. When using SVN, the first step is to use the *import* command in order to add an existing project to an existing repository which will add existing files to the repository. The result will be a directory which is under version control, called *working copy*. In the *working copy*, files can be modified locally. Other developers can create *working copy* using the *checkout* command. Then, after making changes, the *checkin* command sends the changes to the repository. Using the *update* command, developers can bring recent changes from the repository into a working copy. There are few other useful commands that developers need to know about. A *diff* command creates a difference between a file in the repository and the working copy. A *conflict* happens when two developers are trying to change a file and the system is unable to resolve the change. Usually, conflicts are resolved by choosing one of the versions by the developers or by integrating changes from both into the repository.

2.4.3 The time entry system

The *time entry system* is used to log the amount of time which every team member has spent on a specific task during the day. A task during the maintenance could definitely be a bug, issue, or change request. Everyone in the team needs to log the time they have spent on tasks. Some software organizations pay teams based on the reports of the time entry system; this is done in order to incite them to be more careful about recording their work. Today, software development is becoming more globalized, and organizations will use more web-based *time entry system*s.

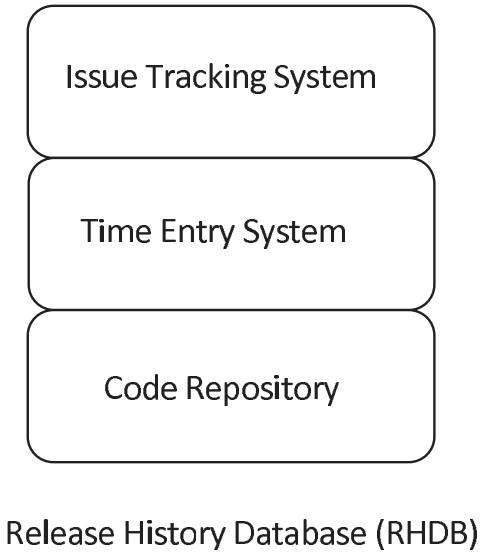

Figure 3: Release history database boundary

2.4.4 Release History Database(RHDB)

A software release includes a set of software changs that includes new functionalities, changed functionalities or fixes on bugs that will be available in the production environment. We can monitor and control software changes in the *release history database*, which points to the information available in the software development environment from one release until the next. In general, the *release history database* (RHDB) points to any raw data which can be retrieved and aggregated from any source during the software development or maintenance phases.

In this thesis we will define a *release history database* for information which is in the *issue tracking* system, *code repository*, and *time entry* system. Figure 3 illustrates the boundary and information sources of the release history database. The release history database contains a wealth of hidden information which could create improvements and innovations in the field of the software maintenance.

Chapter 3

Problem and motivation

In this section, we discuss the problem and the motivation behind this research which constitutes the scope of this dissertation. The primary motivation behind this thesis is to show that analysis on a well bundled software history improves in software maintenance.

We mentioned in Section 2 that the maintenance phase is the most time consuming and expensive phase in the software industry. Inefficient management in software maintenance is one of the reasons for low productivity in the maintenance phase [35]. Inefficient management in software maintenance is also the root cause of missed deadlines and surpassed budgets [35]. Senior managers usually give more priority to software development as compared to the software maintenance; for example, junior software developers are usually assigned to fix bugs and apply new changes in software maintenance. However, assigning junior software developers may increase costs in the long term and cause the organization to have less quality software in place. Moreover,

the maintenance phase is the first candidate for outsourcing to reduce costs. However, outsourcing software could also bring more errors and bugs into the system.

We believe that a lack of efficient management in maintenance is one of the reasons for the software crisis. On the other hand, the majority of studies in software maintenance have been done around code analysis and feature [50] location, not on improving software maintenance management. There have not been enough studies about improving and innovating new approaches for management in software maintenance and most of the tools and studies for the maintenance have focus on the code analysis. [35]. Management in software maintenance does not utilize thoroughly enough advantage of information which is placed in software histories.

There are some commercial and non-commercial tools which contain a subset of information we need in the *release history database* before analysis. *DrProject* [14] is a project management tool that has *revision control*, *issue tracking* and *mailing lists* integrated. Any project in *DrProject* supports wiki, subversion, ticket, mailinglist, roadmap and tags. *DrProject* has two sets of the sources of information we need for analysis, but the time parameter is not available for analysis. The other issue is that the data from different sources in the *DrProject* are not related together. This does not allow to perform analysis on the information which is stored. Jira [28] is also another *issue tracking system* tool developed by *Atlassian* [2]. Jira allows developers and maintainers to log their daily activities for tasks and bugs. In our case study, we have used Jira as an *issue tracking system* and we have integrated Jira with a *version control system* and *time entry system*. Hackystat [23] is another

open source framework for analysis, visualization and interpretation of data collected during the software development process. It uses small software plugins as sensors which collect data from the software development tools and subsequently sensors send raw data to a repository in order for analysis. *Sonar* is another open source framework which helps in highlighting complex areas of code that are insufficiently covered by test cases that can be sources of future bugs. This automatic detection of bugs allows teams to fix them before deploying the software in the production environment. *Sonar* also increases the maintainability of the software by detecting and reducing duplication, complexity, and potential bugs. *Sonar* helps managers with bug prediction and software quality improvement in alternative way we use the release history database. *IBM Rational Team Concert* [10] is a collaboration tool that has task tracking, source control and build management, thus allowing teams to track all aspects of their work. The *IBM Rational Team Concert* has two of the three resources we need to integrate before analysis. Therefore, if we decide to use the *IBM Rational Team Concert*, then we need to set up a *time entry system* in the software maintenance environment; and in the next step we need to integrate these three sources of information together. Our approach is to analysis the data that exists in the release history database. The source of the data for release history databases could be any issue tracking system, version control system or time entry system.

Our objective is to use existing software data which can be found in the release history (See Section 2 for definition) in order to improve management in maintenance. These improvements include reducing number of bugs from one release to the next

release, finding *risky objects* (See Section 6 for definition) and assisting managers in choosing maintenance team and location.

One of the main issues for the utilization of release histories is that information is scattered thorough different places in the software development/maintenance environment. Therefore, the data in these resources are not valuable unless links are made between the data. We analyze more amounts of the data which would lead to more detailed, helpful results which managers can use to enhance software maintenance.

Several works have proposed using release history data to improve software maintenance and these works used release history data to improve maintenance by predicting bugs [34] or to determine the person who is the best choice for fixing bugs. Furthermore, there are some studies about helping maintainers with code comprehension and code analysis [16] [17], but they do not directly help managers to improve management in software maintenance. Managers are always worried about cost and time in maintenance, and senior managers in the maintenance phase are always concerned with choosing the maintenance geographical location and maintenance team.

Chapter 4

Proposal

As a solution to the short comings in management during maintenance, we build a *release history database* (RHDB) and propose analysis on this database to help maintainers and senior managers to help them improve this process. Our proposed analysis on our suggested *release history database* helps managers not only assess the current situation, but also provide a set of goals to shape the future behavior of the software maintenance process. In this thesis, we will also provide a set of new metrics for software maintenance that will help managers to control what is happening in the maintenance process.

Our proposed approach is focused on analysis of an integrated *release history database* from the three different resources presented as follows:

1. An issue tracking system which has historical information about bugs, issues, and changes.

2. A code repository which has all source codes of the software and all code histories.

3. A time entry system which has all the time spent on tasks, bugs, issues and any other task in the maintenance phase.

Previous works have mostly focused on the analysis of a limited set of sources. Most of them used *issue tracking systems* and *code repositories* as major sources of information. They have considered the *time* parameter in their studies implicitly. Therefore, the metrics which are defined in previous works do not have time related parameters. Moreover, in the literature, several works related to the mining of software repositories fall into code analysis and bug prediction. Furthermore, information in the release histories is not widely used to improve management in software maintenance.

Therefore, in this thesis, we present how to analyze and run queries against our proposed *release history database* and complement the previous works by adding the notion of time explicitly. By adding a time entry system, we have a more explicit time parameter in our study which allows us to have a better understanding of what happened in previous releases from the performance point of view.

To retrieve information from *release history database*, we need to select a time period. In this work, we consider either the time period from one major Release to the next major Release or the time period from one minor Release to the next minor Release. For example, we can assume that a development team has released Release 1.1 to customers. The next minor release would be 1.2 which may include bugs'

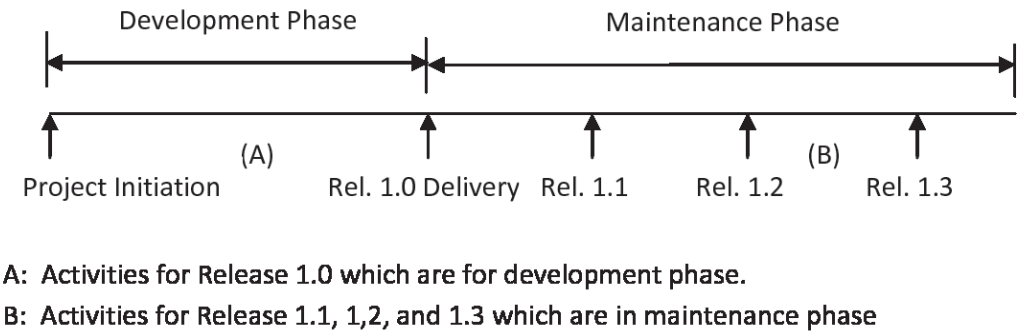

Figure 4: Releases in development and maintenance phase.

fixations and change requests. Any activity after Release 1.1 to Release 1.2 would be considered as a maintenance activity. The work-flow for issues is presented in Figure 1.

After Release 1.1, customers would raise bugs, issues, or asking for change requests using issue tracking system. Maintainer would fix bugs and develop changes requests definitely after the change impact analysis. Finally, as shown in Figure 4, the team will deliver Release 1.2 which includes all fixed bugs and changes.

In this thesis, we integrate an issue tracking system, a code repository and a time entry system as the main three resources to build a release history database to provide more practical information about the software system. Each of these resources has valuable information that is not valuable individually.

The expected contribution of our thesis is to improve software maintenance phase by analyzing the integrated *release history database* that consists of three different systems. Potential beneficiaries of this approach include maintainers and project managers who can be more productive and on time. Our goal is to help managers to

prepare achievable plans in order to meet cost estimates and scheduling commitment.

In the following chapters, we discuss our methodology for proposing *release history database* can be used to retrieve useful information from Release histories. In addition, we explain how we extract a set of metrics from the proposed *release history database* to evaluate the source code.

Chapter 5

Building a release history database

In this thesis, we propose an approach to build a *release history database* and analyze its information in order to improve efficiency of software maintenance. To analyze the *software repositories*, we need to prepare a suitable environment for retrieving release history information which comes from different resources. This chapter presents our approach to build a release history database.

The reminder of this chapter is organized as follows: In Section 5.1, we present our three major sources of information namely an *issue tracking* system, a *code repository*, and a *time entry* system while explaining how to integrate them so as to build an integrated *release history database*. We also present the schema of our proposed *release history database*. Section 5.2 presents how we set up a query system to retrieve information from three sources of information.

5.1 Building an integrated release history database

In our proposed solution, we aggregate distinct data to build our integrated *release history database*. As we addressed in Section 4, these valuable data are scattered in different resources. We choose three major sources of information during software maintenance to retrieve necessary information including an *issue tracking* system, a *code repository*, and a *time entry* system.

Figure 5 shows an example of how in our proposed *release history database* the data from the *issue tracking* system, the *code repository*, and the *time entry* system are integrated and associated.

In our code repository every issue, bug, or change request is associated with a revision number. As we have described in Section 2, the code repository assigns a number to every check-in as a *revision number*.

Figure 5 also shows how an issue, bug, or change request is associated with at least one piece of source code. Any entry in the issue tracking system is associated with one or multiple records in the time entry system. For example, as seen in Figure 5, BUG-4591, which is "New User Doesn't Get Confirmation Email After Registration", has been raised by a customer and a developer has fixed it. By having association between *issue tracking system* and *code repository*, we realize two classes (*Register.Java* and *RegisterAction.java*) and one JSP page (*register.jsp*) have been modified during the bug fix. Furthermore, we establish an association between the issue tracking system and the time entry system which allows us to know that Jason has spent 2:00 hours

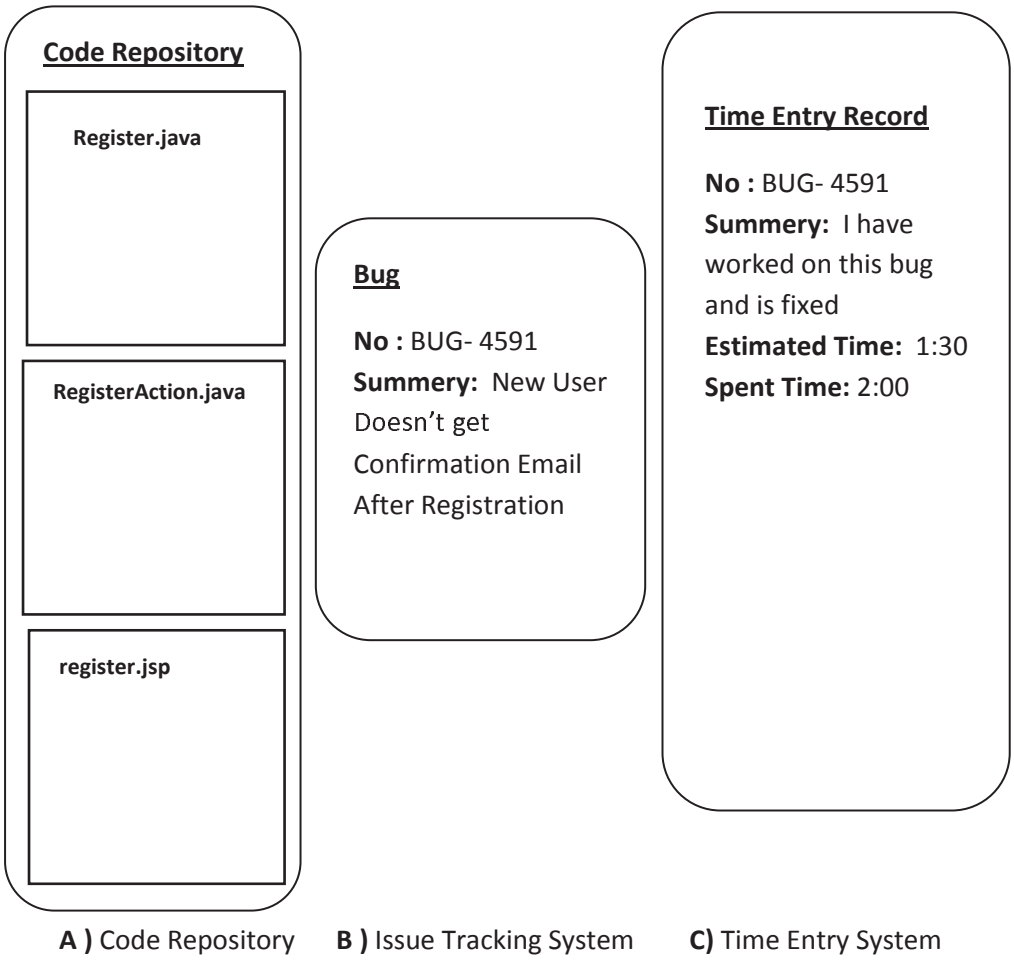

Figure 5: An example to show how different data from *Code repository*, *Issue tracking*, and *Time entry* systems are integrated and associated

fixing them.

First, we need to set up an environment to aggregate and integrate the *issue tracking system*, the *time entry system*, and the *release history database*. The *issue tracking system* and the *time entry system* have their own database while the *version controls* keep all information in the plain text files. Therefore, we need to pull required data from those two databases and the text files from our own *release history database*. These resources have large amounts of data that would need a lot of disk space. We do not need to acquire all of their data, so we pull out only the data which we need in the analysis. The data we need for analysis includes: 1) The information that is related to the issues 2) the information that is related the code 3) the information that is related the time entry system. Figure 6 shows all the fields which we require in order to analysis.

In order to store information coming from three different resources, we need to have a *Relational Database Management System* (RDBMS): so the *Relational Database Management System* will contain all data from the three mentioned resources. We import the required information from these three applications and keep it in one integrated *release history database*. Figure 7 describes the methodology we use to build a *release history database*. Our methodology contains the following steps:

1. We build the release history database.

2. We retrieve and import data from the *issue tracking system*.

3. We retrieve and import data from the *version controls*.

4. We retrieve and import data from the *time entry system*.

We propose a schema for our *release history database* which is shown in Figure 6. This schema contains 5 tables named *revision history*, *time entry*, *issue*, *project* and *user*. As shown in Figure 6, on one hand, every *issue* corresponds with one or more revision history entities. On the other hand, every *issue* record is associated with one or more records in the *timeentry* table. Furthermore, the *user* table has a one-to-many relationship with *issue* and *timeentry* tables. One of the significant differences between our *release history database* and the other approaches is that our data are strongly associated together. As discussed before, we need to the maintenance team follow some rules for code check-in and booking thier time that are determined before the maintenance phase. These powerful associations let us dig into the data and analyze of it so that we can assist managers in the maintenance phase.

In order to establish associations between the three different resources, we need maintainers to follow some rules. These rules need to be determined in the beginning of the maintenance phase by team leaders; these rules will help to build association between the different sources. Maintainers should be informed about the rules at the beginning of the maintenance. In every *issue tracking system*, every issue or bug has a *unique identifier*. Tools usually use a combination of an abbreviation of the project name concatenated with a sequence number. For example, if the project is *online banking*, the *unique identifier*s of issues are ONLINBNK-1, ONLINBNK-2, ... ONLINBNK-823. The primary rule that we need to team follow is mentioning the issue's unique identifier in every code check-in. The second rule that the team

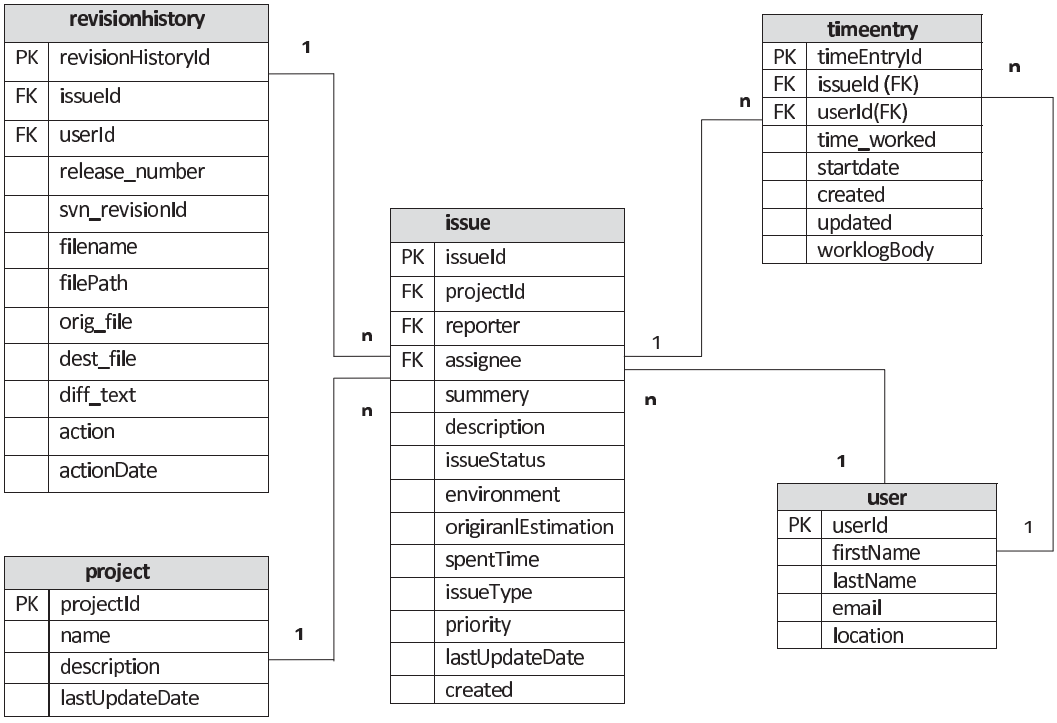

Figure 6: Release history database schema

needs to follow is that every developer needs to mention the issue's unique identifier in the *time entry system*. The processor which will be described in this section tries to find the issue unique identifier in any check-in message or any time entry record. If the processor does not find the unique identifier, then a report will be generated to inform the orphan records. The recommended solution is having some validation scripts in order to verify that every code check-in and every time entry record has a valid reference to an *issue identifier*. In case the maintainer forgets to put the reference in the check-in message or time entry record, validation scripts will alert the maintainer with a proper message. Having validation processes before any check-in or booking any time entry record, will guarantee that all required information has been provided for association building. Following the given rules will guarantee that all required associations between *issue*, *code revision* and *time entry* records will be in place.

5.2 Retrieving Required data from Three Resources

We finish building our proposed *release history database* by retrieving data from three resources: our *issue tracking system*, our *code repository*, and our *time entry system*.

5.2.1 Retrieving information from the issue tracking system

The first step is retrieving data from the *issue tracking system* into a relational database. As we mentioned before, *issue tracking systems* are being used by customers, developers, testers and managers in different geographical locations and different time zones, so most issue tracking system applications are web-based applications. These applications store information in *Relational Database Management Systems* (RDBMS). Moreover, issue tracking systems usually provide *API*s or *web services* for developing plug-ins or doing queries. Thus, we would have two options to retrieve the corresponding part of data into our *release history database*: 1) using *API*s or *Web Services* to get the required data or 2) directly accessing to issue tracking RDBMS. We preferred to choose direct access to a Relational Database Management System because it was faster than using APIs to retrieve required data and the process for retrieving data was running on the same host.

After any release and before starting any activity for the next release, we need to import data from the issue tracking system to our release history database. We import data from the *issue tracking system* to the *issue* table in our schema as shown in Figure 6. The most important data (fields) coming from the *issue tracking system* are as follows:

issueId: This field is a unique number in the *issue tracking system* and usually is a combination of a *project abbreviation* and a *sequence number*.

projectId: This field is a foreign key to the *project* table which holds information

about all projects.

reporter: This field points to the person who has reported the bug, issue, or change request.

assignee: This field shows the person to whom the issue is assigned.

summary: A summary of a task, bug, issue, or change requests.

description: This field has full descriptions and full details about issue, task, bug, and change request.

issueType: This field indicates whether the issue is a *task*, *bug*, *change request*, or *issue*.

issueStatus: This field indicated the status of the issue in the workflow which is described in Figure 1.

original_estmaition: Points to the original time estimation of the issue as specified by managers, customers, or the team leader.

time_spent: This field points to the actual time spent by a developer on an issue, bug or change request.

environment: This field points to the environment where the bug or change request has happened. For example, *environment* could be *web*, *desktop*, or *iPhone*.

priority: This field refers to the priority of the bug or task which is being specified in *work flow* described in Figure 1.

lastUpdateDate: This refers to the date of the latest change on record.

created: This field refers to creation date of the issue, task, or change request.

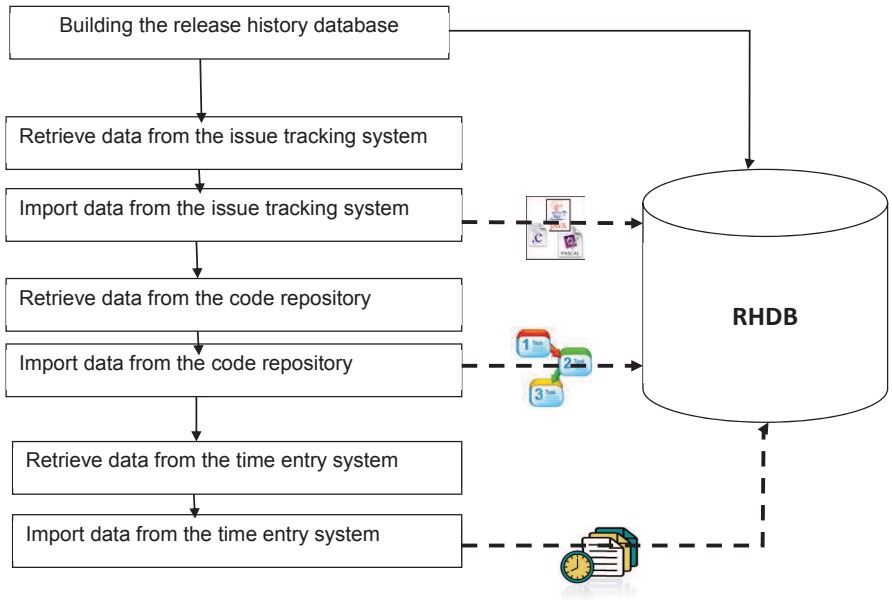

Figure 7: Activity diagram for building release history database

These are the general fields in the *issues* table that are common in all *issue tracking systems*.

5.2.2 Retrieving data from the code repository

The second step in building the release history database is retrieving data from the *code repository*. As we mentioned, teams make *branches* after releasing software. For example once the team has released version 1.3, the team would make a branch and snapshot for the Release 1.3 codes. Therefore, after Release 1.3, the team has a snapshot code from Release 1.3, as well as snapshots for Release 1.1 and Release 1.2. Moreover, having snapshots of all major and minor releases lets us have details of changes between releases. We have the names of files which are changed, actions,

action dates, and the person in charge of the change. In this step, we import all required information and raw data for each release from the *code repository* to our Relational Database Management System. We import only the data which relates to maintenance. Our approach needs to have all data which indicates the association between any code *check in* and *issues*, so we need to import all required information from the *code repository* and any other associated data in other parts of the software maintenance environment. In order to achieve the association between the code repository and other resources in our proposed schema, we keep the relationship of *code check in* and *issue* as a one-to-one or one-to-many relationship.

We developed a component to have an automated import process. First of all, it makes an automatic comparison of two code branches between the two latest releases and then sends the outputs to text files. In the next step, the *difference text processor* processes output files and populates data to a table in our Relational Database Management System. The data outputs to the *revisionhistory* table which you can see in Figure 6. The *revisonhistory* table has the following fields:

revisonHistoryId: This field is the primary key to the table.

issueId: This field is a foreign key (FK) to the *issues* table.

userId:: We use this foreign key (FK) field to indicate the user who checked in codes. User information is stored in the *user* table.

revision_id: As we mentioned before, the *revision_number* is a unique sequential number pointing to the code changes that are checked into the code repository.

release_number: This field points to the release number of software.

filename: This field indicates the name of the class, page, or any resource.

filePath: This is the file's path in the current release

orig_file: It points to the full path of the original file in the previous release.

dest_file: File's path in the current release

diff_text: This field indicates the differences between the two files in the two releases.

action: It points to the type of the *code versioning* activities.

5.2.3 Retrieving information from the time entry system

Our third step in building our *release history database* is retrieving information from the *time entry system* and populating data to our *release history database*. Software companies usually use a time entry system to log a team's spending time, and teams are paid based on the records in the time entry system. Moreover, a *time entry system* helps managers have the cost for each project. In the third step, we import data from the time entry system to our RDBMS. Any record in the time entry system has a one-to-many relationship with a bug, issue, or change request. We import information to the *timeentry* table which is shown in Figure 6. This table keeps the following information:

timeEntryId: This field is the primary key (PK) of the table

userId: A foreign key (FK) to the user table is stored here.

time_worked: This field shows the time spent spent on bug, task, or change request.

startdate: The task's start date is being stored in this field.

created: A field which shows when the record is created.

updated: This points when the record was last updated.

worklogbody: The developer or maintainer gives some details about the task in this field.

There are two other tables in our schema shown in Figure 6: *user* and *project*. The followings are fields that are in the *user* table.

userId: This field is the primary key(PK) of the table.

firstName: This field shows the developer or maintainer's first name.

lastName: This field shows developer or maintainer's last name.

email: This is the user's email address. It is also plays the role of username in order to login into to system.

location: This field specifies the user's location.

We now have our *release history database*, and in the next section we are going to *analyze* information which exists in the *release history database* in order to help the software maintenance team have more efficient software maintenance. In the following chapters, we present a set of queries to extract data from our proposed *release history database* for specific management tasks which help manager in software maintenance activities.

Chapter 6

Analyzing release history database

6.1 Introduction

The number of software systems rolling into the maintenance phase is rapidly growing [35]. Support and maintenance is the most time consuming phase of the software life cycle. Releases usually suffer from delays in delivery time and from being over budget [35]. There is a difference between the planned and actual progress of a software project [61], while managers usually do not have enough information to find the reason for delays.

Our proposed approach is to analyze data in software *release history database* in order to prepare reports for senior managers. Then, there will be a chance for managers to improve things from one release to the next one. Several studies investigated the roots of delays in software deliveries and reported different results. Herbsleb *et al.* showed that distance and global software development is one of the reasons for

delays. In contrast, Nguyen *et al.* [47] reported that distance does not have a strong effect on task completion times.

The aim of our study is to add the existing knowledge of the reasons of delays and overruns in software development, specially in the software maintenance. In our study, we analyze software repositories to find the subsystems, features, classes, methods, or even code snippets which are root of delays in the software progress. Our approach allows managers to have a report on pieces of codes, classes and even methods for which the time spent for fixing bugs or changing code was higher than the average time the team usually spends [40]. In general, there are two policies in software maintenance: work-based time policy, and time-based policy [51]. In the work-based policy, the team needs to do a fixed amount of work in an open time frame. In the time-based policy, a fixed amount of time will be spent to maintenance. The total cost of these two polices is almost equivalent [51], but both experience delays in delivering software on time.

The remainder of this chapter is organized as follows: in Section 6.3, we introduce *risky objects* which are obtained form analysis of the *release history database*. In Section 6.4, we present our methodology by introducing a metric called *Average Time to Change a Line of Code in Release* (ATCLiR) to measure the updating time for each line of code. This metric helps managers to identify classes, methods, or subsystems for which maintenance is more time-consuming than for previous releases. In Section 6.5, we introduce two new metrics and a profile for each developer which will help managers to choose the maintenance team. In Section 6.6, we explain how we

combine our proposed metrics into a new set of metrics by assigning different weights to them in order to help managers identify the location for software maintenance. In Section 6.7 we will help managers to make right decision by analyzing on the *release history database* for future of the software.

6.2 Metrics

We are following certain activities for every metric. For metric definition, we use the GQM (Goal-Question-Metric) method [6]. This method defines the metric based on the goal we want to achieve from the measurement [11]. Empirical validation is required to demonstrate the usefulness of a metric in a commercial application [7], [18].

6.2.1 Metric Definition

Analyse issue tracking system, code version control and time entry system

For the purpose of helping management in the software maintenance

With respect to finding risky objects, finding time consuming code, recommending maintenance team and recommending maintenance location

From the point of view of Software maintainers and managers in maintenance

In the context of Master thesis in Software Engineering at Concordia University

6.2.2 Planning

After the definition, we do planning for experiment. The planning is some preparation activities for how the experiment is conducted and has following activities:

Context selection. The context of the experiment was maintainers and managers working in the maintenance phase of a commercial application.

Selection of subjects. The subjects were chosen for convenience. For example we choose maintainers and managers working on a single project.

Variable selection. The independent variables are the metrics we are calculating. The dependent variables are lines of code that are changed during bug fixing, number of different type of bugs and time spent on bug fixing.

Instrumentation. The objects used in the experiment were different releases of a commercial software.

Hypothesis formulation. We planned to test the following hypotheses:

First Hypothesis:

Null hypothesis: A code with more changes compared with previous releases is not expected to have more bug reports in the next releases.

Alternative hypothesis: A code with more changes compared with previous releases is expected to have more bug reports in the next releases.

Second Hypothesis:

Null hypothesis: Maintainers and locations with lower *average of new introduced bugs in 100 lines of code per release* are not suitable for maintenance.

Alternative hypothesis: Maintainers and locations with lower *average of new introduced bugs in 100 lines of code per release* are more suitable for maintenance.

Third Hypothesis:

Null hypothesis: Maintainers and locations with lower *average of reopened bugs in 100 lines of code per release* are not suitable for maintenance.

Alternative hypothesis: Maintainers and locations with lower *average of reopened bugs in 100 lines of code per release* are more suitable for maintenance.

Instrumentation. The objects used in the experiment were java classes of a commercial project are described in Section 7.

6.2.3 Operation

Preparation: We chose releases from software that were already released and there were no activities on those releases. We only needed the software owner to give *read access* to the software maintenance environment.

Execution: We had *read access* to the maintenance environment and we were able to integrate required information from different resources. For *threshold*, described in Section 6.4, a questionnaire is used. The questionnaire is described in the Appendix A

Data validation: We verify the information integrated from the three resources in the *release history database*. It is important we do not have any *code revision* and *time entry* record which is unrelated to at least one *issue* or *bug*. In our experiment, we set up a pre-commit script for any code check-in which does not allow to code check-in to the repository without having a valid *issue id* in the comment of code check in.

6.2.4 Analysis and Interpretation

We analyzed the result in the detail in the Section 6.3, 6.4, 6.5.1 and 6.5.2.

6.3 Risky objects

We introduce *risky objects* as the objects in the software which may introduce new bugs and defects to the software. *Risky object*s are the pieces of code bundled as subsyetem, feature, class, or method which we suspect that may introduce new bugs to the software.

There are a lot of studies about bug predictions in code and it was one of favorite domains in mining software repositories [12]. They have used different techniques for bug prediction in terms of complexity and type of techniques. Some bug prediction techniques rely on various types of information like code metrics [5, 46] include static

code analysis, mining code repositories in order to predict bugs; some researchers have worked on software process metrics like number of change requests and recent activities [45, 9], while other researchers have worked on defect prediction based on previous bugs [31, 24]. In general, previous researchers have covered two aspects of defect predictions: the first one is relationship between software defects and the second one is code metrics. Other studies have used other techniques. For example, in [52], they have tried to predict number of defects based on the *import relations*.

There are studies which use software repositories to specify the objects which may introduce new bugs to the software. We have extended their techniques to use more resources from software repositories to get more accurate and specific result.

Jacek Silwerski *et. al* [54] showed that a significant percentage of bugs appear after fixing bugs and applying changes. Their study [54] tries to locate bug fixes or code changes that are inducing bug fixes. In another study, Backer *et. al* [3] have called this concept *fix-on-fix changes*.

As our study shows, a code which is subject to huge changes due to multiple bugs is also likely to indicate new bugs into the new release of the software during maintenance. The time developers have spent on the *changing code* highlights that there were challenging issues in those codes. The more time spent on a piece of code, the greater the chance of introducing new bugs.

The maintenance team fixes raised bugs in each release, and then the QA team verifies bug fixes. While the software passes QA controls and is close to release, our software repository analysis reveals *risky object*s (features, subsystems, classes,

methods, or code snippets). Our analysis determines *risky objects* from the previous release to current release.

We have focused on software maintenance. The quality assurance (QA) teams test software before releasing it, but software always has some bugs. Our knowledge and studies show that bug fixing usually does not need huge changes in the original code unless we have a change request or refactoring in the code. From a quality point of view, making huge change in the code for a bug fix is a flag that the bug was not a normal bug and points us to a critical problem in that piece of code or subsystem. In other words, we say that if we have changes in the code higher than the threshold % change in a class, or in a method from one release to the next release, that is a flag for having a *error prone code*, which we call a *risky object* in our thesis. We call this threshold percentage as τ %. We take advantage of the study conducted in [52] and we learn from their techniques.

6.3.1 Threshold for risky objects

We need to specify τ as a threshold for having *risky objects*. Code with changes more than τ % change would be candidate to be flagged as *risky object*. We have decided to choose the threshold based on a questionnaire from experienced developers and maintainers. We did a survey asking five development team leaders, five QA team leaders, and five delivery managers to determine threshold range for code changes to identify *risky* codes. In Appendix A, the survey questions, survey audience and their answers are described. Our survey showed that the average threshold, τ, is 24%.

When we address the threshold is 24, means that for τ more than 24% changes in the code (because of fixing a bug in a class or pages), the class is a candidate for being one of the classes or pages which could introduce new bugs to the software. Having a list of classes and pages with more than 24% changes during a release (because of a bug) is a flag for managers that those pieces of code and corresponding subsystems may introduce new bugs to the next release. We call *risky* classes, pages, or subsystems the ones that have code changes more than our threshold.

The τ in the query for analyzing software repositories can be modified based on managers' experience, project timing schedule, costs or other parameters during software maintenance. However, a higher percentage threshold leads to increasing risks of having more bugs in the next release.

Our objective was to help team leaders and managers in the maintenance phase. The integrated *release history database* allows us to highlight *risky objects* before releasing software. Therefore, project managers, delivery managers and risk officers will have a chance to investigate and take appropriate actions once they are informed about *risky objects* in the software release. For example, managers could ask senior software developers to review code changes in *risky objects*. This notice will let the team prevent the introduction of new bugs into the software.

To retrieve *risky object*s, we analyze all code changes which are from the latest release upto now by mining in software repositories. In our analysis we will find out the bugs and issues which code changes are related to. Furthermore, we will find out the time spent on the code by querying in our proposed *release history database*. Any

Table 1: A high level quert for risky objects

```
SELECT
required fields

FROM
timeentry, issue, revisionhistory

WHERE
conditions for joining tables

AND
filter data for release = ?

AND
percentage of changes on objects is > threshold
```

class or page may have been changed multiple times in each release for different bugs or change requests. For example: *Register.java* may have been modified and then checked in to the repository for two different bugs in a release. In same example, our query shows that *Register.java* has been changed for two bugs and three new features which are raised in Release 1.2. Table 1 displays a high level query for retrieving objects, including classes, and pages, which are candidates to introduce new bug into the software.

6.4 Analysis for time consuming codes

In our approach, we introduce a metric called *average time to change a line of code in release* (ATCLiR) which is measured during the progress of software maintenance

starting from the first major release, Release 1.0.0 or 1.0. We measure the average time for updating each line of code performing maintenance tasks, including fixing a bug, or adding a new feature.

Table 2: Average time to change a line of code in release(ATCLiR).

Release	**ATCLiR** (min)
Release *r1*	*t1*
Release *r2*	*t2*
Candidate for Release *r3*	*t3*

Tang *et. al* [35] have introduced a metric called *average time turnover of a MRF(Maintenance Request Form)* as a metric to help senior managers to control situation.

They have calculated *average time turnover of a MRF* as follows:

$$Average\ time\ turnover\ of\ a\ MRF = \frac{\sum_{i=1}^{n} T_i}{n} \tag{1}$$

where we have T as the time for development on MRF, and n is the number of MRFs in the release.

Then we calculate ATCLiR as follows:

$$ATCLiR_r = \frac{\sum_{i=1}^{n} T_i}{\sum_{i=1}^{n} LC_i} \tag{2}$$

where T is the time spent for fixing any bug issues in each release, and LC indicates the number of the *Line of Code* which is modified for the bug i or change request i in the Release r.

Our proposed metric, ATCLiR, calculates the average time based on the lines of code that have been changed. Therefore, the ACTLiR metric helps managers:

1. Compare and find the time consuming parts of code (the ATCLiR for the previous releases are smaller than the ATCLiR for the current one).

2. Decide and determine the next actions in order to deal with the time-consuming codes. This method help managers to decide whether it is worth (time and budget) continuing maintenance, or wether they should apply refactoring.

For example, if we assume that a team has started fixing bugs for Release *r3*, we know that the ATCLiR (for an insert, update, or delete) for bug fixing in Release *r1* was *t1* minutes per line and that the ATCLiR for Release *r2* was *t2* minutes per line which is shown in Table 2. While the team is maintaining and fixing bugs for Release *r3* and the QA team is testing software, there is a chance for managers to identify classes, methods, or subsystems where their ATCLiR in Release *r3* is more than the ATCLiR of Release *r1* and the ATCLiR of release *r2*.

We will illustrate a case in Chapter 7 to show how finding time consuming code helps determining the snippet code, method, class or subsystem that may introduce new bugs into the software.

The reason why we could retrieve this information from the release history database is that we elaborate a powerful relation between the *code repository*, *issue tracking* and *time entry* systems. The *time* factor for each bug or change request is available for us because we have integrated the *time entry system* and *issue tracking system* together. Furthermore, we have the number of the modified lines of code for each bug or change request as we have integrated the *code repository* to the *issue tracking*

system. In Table 3, we present a high-level query to extract all the information from the *code repository*, *issue tracking* and *time entry* in order to calculate the ATCLiR metric.

Table 3: A high level query for time consuming code

```
SELECT
required fields

FROM
timeentry, issue, revisionhistory

WHERE
condition for joining tables

AND
filter data for release = ?

AND
filter in revisionhistory for release = ?

AND
spent time on unit of code > than average = ?
```

6.5 Recommending maintenance team

Software development is a construction process; in contrast, debugging and bug fixing is a search process on codes, runs, and even on project history [62]. Maintainers cannot trust the first developers' assumption. Maintenance is therefore more challenging than software development.

In our approach, we analyze the integrated release history database in order to help managers choose developers with the best possible performance for the maintenance phase. Based on the history of the maintainer's activity in the past releases, the type of activities in which they have participated, and their performance, we recommend maintenance team for the next release. Some programmers are good in developing new features while others are good in code comprehension, refactoring, and bug fixing. A *release history database* has valuable information to let us know which developer is the best choice for maintenance team. In this Section, we propose four different metrics to help managers choose the maintainers for maintenance team.

ATCLiR calculates the average time to change a line of code in specific release. It is clear that a higher amount of *code change* does not necessarily point to better performance. Sometimes a developer writes a lot of code but introduces more new bugs into the software. Therefore, we need to extend the ATCLiR metric by adding information about the maintainers' performance during maintenance. In other words, ACTLiR is not enough for making decisions and we need to provide support using new metrics. In this section, we also also introduce two new metrics in Section 6.5.1 and Section 6.5.2 which will help managers to arrange their team and choose the maintenance location.

So far, we have *average time to change a line of code in release*(ATCLiR) for a team during the different releases and we calculate their average.

There are some works which recommend organizing teams by applying heuristics against data collected from the software development area in order to determine who

is an expert in what area of the system [39]. For example *Expertise Browser* [41] is mining version controls to determine expertises in different areas. Then, researchers come up to add more parameters to data driven from software repositories in order to determine development teams; for example, Griba *et al.* [22] used the *number of line of code* that each developer has modified. They have used *number of line of code* as one of metrics to choose thier team. We have used existing techniques that mine software repositories in order to determine the software development team. We will discuss our approach in details.

In our approach, we combined *average time to change a line of code in release (ATCLiR)* and two new metrics in order to help senior managers to select maintainers for maintenance.

As we discussed in Section 6.4 Tang Li *et. al* [35] introduce the *Average Time Turnover of a MRF(Maintenance Request Form)* to help senior managers to control situations. They claimed that because the MRF is a core of maintenance activity, monitoring the state of the MRF is a very critical task that needs to be performed.

Having the *average time turnover of a MRF* [35] does not lead us to any action on the team to improve efficiency in the maintenance. We use these metrics in conjunction with more resources from software repositories. We analyze our proposed *release history database* to calculate the *average time turnover of a MRF* for maintainers in different releases.

In addition, we have valuable data from our release history database which shows

from which developer's code *reopen* and *new bugs* come. Using these data, we introduce two new metrics here which will help managers to choose maintenance team based on the *average number of introduced bugs* and *average number of reopened bugs*. These two metrics, namely *average number of new introduced bug per 100 lines of code in each release* and *average number of reopen bugs per 100 lines of code in each release*, are explained in the following section.

6.5.1 Average of new introduced bugs in 100 lines of code per release

Maintainers may introduce new bugs to software while they are fixing bugs or applying new changes. We call these bugs as *new introduced bugs in fixes (NIBiF)*. Every maintenance team uses its workfolw for the *issue tracking system*. A typical workflow is shown in Figure 1. We need team leaders or maintainers to specify that the bug is a *bug from origin software* or *new introduced bugs in fixes* or according to their knowledge of the application. The root of *new introduced bug in fixes* could be one of the following items:

1) Maintainers do not have enough knowledge about the business domain of application, so fixing a bug or applying a change breaks a feature somewhere else. Studies show that software knowledge usually is not documented very well, so maintainers does not have enough knowledge about software. The *NIBiF* bugs are new bugs, but their root cause is fixing of reported bugs in maintenance.

2) As shown in [54], a significant percentage of bugs are appears in after bug fixing

(fix-on-fix changes).

We need to consider *new introduced bugs in fixes* in releases by maintainers while choosing a team for maintenance. Maintainers introduce these type of bugs while fixing bugs or applying new changes. We calculate the *average of new introduced bugs in 100 lines of code per release*, which in turn indicates the average number of NIBiF which have happened in the total lines of code that were changed in the release. We calculate *average of new introduced bugs in 100 lines of code per release* as follows:

$$Average\ of\ new\ introduced\ bugs\ in\ 100\ lines\ of\ code = \frac{\sum_{i=1}^{n} New\ Introduced\ Bugs_i}{\frac{\sum_{j=1}^{n} LC_j}{100}} \tag{3}$$

where n is the number of bugs in the release and LC points to the number of *Lines of Code* for any change or bug fix.

We need to specify *new introduced bugs in fixes* among the other type of bugs. In our approach, team leaders and maintainers specify the type of the bug before or while fixing the bugs or issues. They are allowed to change the type of the bug while the maintenance team is working on the next release.

6.5.2 Average of reopened bugs in 100 lines of code per release

Fixing bugs is the most important responsibility of maintainers. We call a bug a *reopened bug* if the bug appears again in the software after fixing the bug and releasing software as a patch, minor, or major release. In the other words, if the bug is reopened, this means that the bug is not fixed completely. The maintainer probably did not spend enough time fixing the bug, or did not spend enough time reproducing all failure cases in order to fix them, or the maintainer did not have enough knowledge in the software's business domain. All these cases usually happens when the maintainers are different from the developers. Any reopened bug will give a bad reputation to the software organization and will increase the cost. Therefore, it is important for managers to choose a team in order to reduce the number of *reopened* bugs.

We analyze software repositories in order to retrieve the number of reopened bugs introduced by a maintainer in any given release. For example, John has fixed 4 bugs for Release 1.1.3. However, after the release, customers are starting to raise bugs and they reopen 3 of 4 fixed bugs. This means that 75% of bugs are reopened.

By mining our proposed *release history database* software repositories, we introduce a new metric as *average of reopened bugs in 100 lines of code in release*. We have the number of reopened bugs in each release. Furthermore, we have the code change which is associated for each bug. Therefore, we calculate the *average of reopened bugs in 100 lines of code per release* from the associated software repositories which

are bundled in our proposed *release history database*. Meanwhile we can calculate *reopened bug percentage* for any release. We calculate *average of reopened bugs in 100 lines of code per release* as follows:

$$Average\ of\ reopened\ bugs\ in\ 100\ lines\ of\ code = \frac{\sum_{i=1}^{n} Reopened\ Bugs_i}{\frac{\sum_{j=1}^{n} LC_j}{100}} \quad (4)$$

where n is the number of reopened bugs in the release and LC points to the number of *Lines of Code* for any change or bug fix.

After introducing these metrics, we can use these metrics to build our maintenance team. We determine values of metrics for maintainers during past releases, and then we build the team based on the every maintainer's performance.

In most cases of the software maintenance, the knowledge of the domain is an important key for building the team; then our proposed metrics can be used as support for techniques based on using knowledge and expertise as a key factor for building teams. For example, Mocks *et. al* [41] are adding filters to the result of expertise browser [41], which mines version controls to determine level of expertise in different areas. In other words, we are using existing techniques in conjunction with our proposed metrics (obtained from mining software repositories) in order to build our software maintenance team.

	Release Number	Release Number	Release Number
Status			
Average time turnover of a MRF			
ACTLiR			
Average of new introduced bugs in 100 lines of code per release			
Average of reopen bugs in 100 lines of code per release			

Figure 8: Developers' profile.

6.5.3 Developer's Profile

Moreover, we can profile each of our maintainers by mining software repositories, specially in our proposed *related history database*. After starting maintenance for Release 1.0, we start profiling (See Figure 8) the *average time turnover of a MRF* and *average time to change a line of code in release (ATCLiR)*, the *average of new introduced bugs in 100 lines of code per release*, and the *average of reopened bugs in 100 lines of code per release*. We keep and save this information as a part of the developer's profile, and managers can make the best decision for the maintenance team. For example, we can monitor these metrics for each maintainer in Release 1.1, 1.2 and 1.3; if we have a significant increase or decrease for a developer, it would be significant flag for senior managers to pay attention to that maintainer. We also profile each developer as *junior* or *senior* developers.

6.5.4 Building team

As explained in the previous sections, we propose three metrics that can be calculated from our proposed *release history database*. There metrics are the *average time to change a line of code in release (ATCLiR)*, the *average of new introduced bugs in 100 lines of code per release*, and the *average of reopened bugs in 100 lines of code per release*. We elaborate on these metrics in order to choose maintainers for maintenance team. In order to compare these metrics with each other, we need to normalize the data for past consecutive releases. We use the Formula 5 in order to normalize a number, x in the Formula, into a scale between a and b, such that the minimum and maximum of existing numbers are A and B respectively.

$$y = a + ((x - A) * (b - a))/(B - A) \tag{5}$$

We propose maintenance team based on our proposed analysis in the software repositories bundled in the *release history database* for the next release. Then, comparing the team's activity and our predicted model, we show that our proposed team would be more efficient.

Therfore, managers should normalize the three metrics of maintainers using Formula 5. Managers can keep results in a table like Table 4 which has normalized numbers of metrics for maintainers. In the next step, team leaders and managers need to specify how important are metrics for choosing maintainers in order to build a team. Table 5 shows typical ranks for choosing maintainers. Thus, maintainer should determine a *rate* for every metric. The *rate* can be a number between 0 to 10

Table 4: Normalized metrics for maintainers

Maintainers	**M1**	**M2**	**M3**
Maintainer1	$m1_{maintainer1}$	$m2_{maintainer1}$	$m3_{maintainer1}$
Maintainer2	$m1_{maintainer2}$	$m2_{maintainer2}$	$m3_{maintainer2}$
Maintainer3	$m1_{maintainer3}$	$m2_{maintainer3}$	$m3_{maintainer3}$
Maintainer4	$m1_{maintainer4}$	$m2_{maintainer4}$	$m3_{maintainer4}$
Maintainer5	$m1_{maintainer5}$	$m2_{maintainer5}$	$m3_{maintainer5}$

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)

Table 5: Importance rank of metrics to recommend maintenance team

Importance rate	**M1**	**M2**	**M3**
Rank	$rank_{m1}$	$rank_{m2}$	$rank_{m3}$

M1: Average time to change a line of code in release
M2: Average of new introduced bugs in 100 lines of code per release
M3: Average of reopened bugs in 100 lines of code per release

or 0 to 100. The important thing for *rate* is that managers can give weight to metrics in order to compare maintainers using the metrics. The next step is calculating maintainer's rank based on the Formula 6.

$$Rank_{maintainer} = \frac{\sum_{i=1}^{n} rank_{mi} \ * \ mi_{maintainer}}{\sum_{i=1}^{n} rank_{mi}} \tag{6}$$

Therefore, managers will have a ranking for maintainers. Managers will have a number for the team like Table 6. The numbers in the table helps managers to choose the maintenance team. The higher number(rank) for the maintainer means that the maintainer is a better candidate for the maintenance team.

Our analysis for recommending maintenance team is based on one site (location) and we did not consider the maintainer's wage in our proposal for recommending

maintenance team. We would consider these two parameters in the next two sections in order to recommend a location for maintenance.

Having more efficient teams will help us meet deadlines and stay on budget. In this section, we showed how using our analysis techniques allows able to provide this information to senior managers as our *release history database* powerfully because of combining three resources from software repositories during the maintenance phase. Choosing maintainers for a team is not possible with only one or even two resources. We can only retrieve this information if we integrate and highlight these three resources. Table 7 presents a high-level query which we use for recommending maintenance team.

Table 6: Comparing maintainers for recommending maintenance team

Maintainers	**Maintainer's rank**
maintainer1	$\text{rank}_{maintainer1}$
maintainer2	$\text{rank}_{maintainer2}$
maintainer3	$\text{rank}_{maintainer3}$
maintainer4	$\text{rank}_{maintainer4}$
maintainer5	$\text{rank}_{maintainer5}$

6.6 Recommending software maintenance location

Software organizations are interested in distributing their software development and maintenance activity around the world in order to reduce the cost and also to have software support in different time zones. The maintenance phase is always the first candidate for outsourcing. There are several reasons that software organizations

Table 7: Query for recommending maintenance team

```
SELECT
required fields

FROM
timeentry, issues, revisionhistory, users

WHERE
condition for joining tables

AND
filter data for release = ?

AND
filter in revisionhistory for release = ?

AND
group for different developers
```

prefer to have development as an in-house activity and maintenance as an off-shore activity. Here, we present two reasons for keeping software development in house and outsourcing maintenance:

1) The first reason is that during the software development phase, developers need to meet in person with customers, especially in the *Agile software process* where there is a role player called the *product owner* who plays the customer role for the development team. This is one of the reasons that software departments prefer to keep development activities as in house activity.

2) If the developer is familiar with the business in his/her life, it would speed up

the development of software. For example, in a software for home mortgage administration, a software developer who lives in North America or developed countries would be faster and more efficient in developing an amortization feature in a piece of banking software compared to someone who lives a country where people buy their home in cash.

Software companies have different choices for outsourcing maintenance. Table 8 shows the average cost per hour for software development in five typical cities around the world.

Table 8: Average maintenance rate in five different cities

Maintenance Site Location	**Maintenance Rate(Hourly basis)**
city1	$\text{cost}_{city1} USD$
city2	$\text{cost}_{city2} USD$
city3	$\text{cost}_{city3} USD$
city4	$\text{cost}_{city4} USD$
city5	$\text{cost}_{city5} USD$

At first glance, a city with lowest rate is the best candidate to host maintenance, but it maybe is not right place for the maintenance. Analysis in software repositories in the *release history database* will help managers to choose the right place for the maintenance, which may be not the less expensive one. Furthermore, in-house maintenance sometimes is more cost effective compared to outsourcing *software maintenance* activities. Table 12 displays a high-level query for recommending maintenance location.

How we can choose maintenance location? So far, we have metrics obtained from

mining software repositories. The value of metrics are in different ranges and are not comparable. Therefore, we need to normalize the numbers to have them in one range based on Formula 5. Table 9 shows the normalized average of the metrics that are obtained from our analysis for latest releases for different locations. We want to use these metrics to recommend maintainers, considering wage as a factor as well.

Table 9: Normalized metrics for different locations

Site Location	**M1**	**M2**	**M3**	**M4**
city1	$m1_{city1}$	$m2_{city1}$	$m3_{city1}$	$m4_{city1}$
city2	$m1_{city2}$	$m2_{city2}$	$m3_{city2}$	$m4_{city2}$
city3	$m1_{city3}$	$m2_{city3}$	$m3_{city3}$	$m4_{city3}$
city4	$m1_{city4}$	$m2_{city4}$	$m3_{city4}$	$m4_{city4}$
city5	$m1_{city5}$	$m2_{city5}$	$m3_{city5}$	$m4_{city5}$

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance rate (Normalized)

We now have all metrics comparable together. In the next step, we need to rate the importance of the metrics. Table 10 shows the typical ranking of importance of the metrics. In other words, managers are trying to give them weight. Therefore, senior managers can give more significant rates to the metrics which are more important for them. In the next step we use Formula 7 in order to calculate location's rank. The result thus will be a list of locations which are ordered based on the metrics we have retrieved from our proposed software repositories like Table 11.

$$Rank_{city} = \frac{\sum_{i=1}^{n} rank_{mi} \ * \ mi_{rank}}{\sum_{i=1}^{n} rate_{mi}} \tag{7}$$

We only recommend maintainers to the senior managers. They may consider other factors, i.e.: soft skills, to choose the maintenance location for a release. We involve the metrics which are retrieved from the software repository analysis.

Table 10: Importance rate of metrics to choose the location

Importance rate	**M1**	**M2**	**M3**	**M4**
Rank	rank_{m1}	rank_{m2}	rank_{m3}	rank_{m4}

M1: Average time to change a line of code in release
M2: Average of new introduced bugs in 100 lines of code per release
M3: Average of reopened bugs in 100 lines of code per release
M4: Maintenance rate per hour in USD

We analyze information in the *release history database* to recommend a location for software maintenance. We would not be able have this unless we had all three resources integrated in our *release history database*.

Table 11: Comparing locations for recommending maintenance location

Site Location	**Location's Rank**
city1	rank_{city1}
city2	rank_{city2}
city3	rank_{city3}
city4	rank_{city4}
city5	rank_{city5}

Table 12: Query for recommending maintenance location

SELECT
`required fields`

FROM
`timeentry, issue, revisionhistory, user`

WHERE
`condition for joining tables`

AND
`filter data for release in (?  , ?  , ?  ...?  )`

AND
`filter in revisionhistory for release = ?`

AND
`group data for locations`

6.7 Bug fixing vs refactoring vs developing from scratch?

Software has its own age [48] and software dies gradually, moreover bug fixing by junior developers during the maintenance phase sometimes makes code messy. Software sometimes suffers from a weak design or a poor architecture. In both cases, teams know that bugs will always present and customers will send bug reports after each release. Senior managers expect that software will move to a stable state, and they want to see that the rate of reported bugs eventually converges to zero. There is always an open question for managers when bugs are exponentially increasing: should

we stop *bug fixing* and start *refactoring* or should we *start* rewriting the software (developing new software from scratch)? Technologies and frameworks are being changed all the time. Sometimes developing software with new technologies and frameworks is more reasonable (considering price and quality) compared to fixing bugs, adding new features to old software, or even refactoring. That is a hard decision for senior managers to make as they have to use time and cost as major parameters for their decision. Refactoring or starting the development of new software has its own difficulties. Customers and end users often resist changes and they are usually conservative. The new software may not have the same functionalities as before.

The data that we have in a *release history database* and its analysis could help senior managers to make the right decisions. Our analysis report could be one of the parameters which could help senior managers to take a better approach. In any release, we categorize all bug fixes activities into three different categories:

1. **Bugs from origin software:** These bugs are reported by customers or end users and their root *is not* the bug fixes or change requests which were in the *latest major release*. The cause of this type of bug traces back to the development phase of the software. They are not related to the maintenance. Senior managers are expecting that the number of this type of bug converges toward zero after two or three patches or minor releases.

2. **Reopened bugs from previous releases:** These bugs which have been fixed once, but they have been reappeared in the software.

3. **New introduced bugs on fixes:** This type of bug arises from fixes on bugs or from adding new features. We have described this type of bug in Section 6.5.1.

We propose the following strategy after analyzing software repositories to help senior managers to decide whether continuing maintenance, apply refactoring, or design from scratch.

If 1) the number of *bugs from origin software* starts moving toward zero after major releases, and 2) The number of *reopened bugs from previous releases* is not increasing after minor releases or patches after major releases, we suggest letting software remain in the maintenance phase. Decreasing *bugs from origin software* and *reopened bugs from previous releases* means that the software is in a stable state and is not suffering from bad design. New bugs are not introduced after each release and we suggest keeping software in the maintenance state.

If 1) the number of *bugs from origin software* is not moving toward zero after major releases, and 2) The number of *reopened bugs from previous releases* starts growing fast; then we suggest refactoring or developing software from scratch. We expect that the rate of bugs from latest development decrease over time if we have a well developed software and well designed architecture.

Our approach in analyzing software repositories allows managers to make decisions about their strategy for software maintenance. Table 13 displays high level query to run against *release history database* to help managers to identify future path of the software maintenance.

Table 13: Query for bug fixing vs developing from scratch

SELECT
`required fields`

FROM
`worklogs, issues, revisionhistory, users`

WHERE
`condition for joining tables`

AND
`filter data for release in (?  , ?  , ?  ...?  )`

AND
`filter in revisionhistory for relase=?`

AND
`group on number of bugs and spent time for`
`different categories of bugs`

6.8 Accurate estimation

Accurate estimation of software development effort is one of the most critical tasks in software engineering [57]. We know that estimation from experts is the most commonly used approach for the estimation of software development effort [43] since at least the 1960s. *Expert estimation* is more frequently used in estimation because no evidence exists suggesting that formal estimation models lead to more accurate estimations. Phan *et. al* [49] showed that the cause of 44% of overruns is optimistic planning from managers. They showed that over-optimistic estimates and user changes were the most important reasons of overruns.

The difference between *estimated* work time for a release and the *actual spent time* is a challenge through the software life cycle. That difference is a big concern in software maintenance and can lead to the project going over budget. Sometimes, releases are more than 50% over budget. The difference between *estimation* and *actual* work in a release may happen when we have delays for each task in the release. In other words, the difference between *estimation* and *actual* time in *tasks* leads to delays in delivery and being over budget.

Table 14: Accurate estimation query

SELECT
`required fields`

FROM
`worklogs, issues, revisionhistory, users`

WHERE
`condition for joining tables`

AND
`filter data for release in (?  , ?  , ?  ...?  )`

AND
`filter in revisionhistory for release=?`

AND
`group data in order to compare estimation time vs actual time`

Our approach towards analysis of software repositories provides us reports which help managers in the software maintenance phase to detect the reasons for missed

deadlines and exceeded budgets by using data which we have in the *release history database*. We analyze integrated software repositories to retrieve the error rate for each maintainer.

On the other hand, making accurate estimations on the tasks and bugs is a key skill for managers and team leaders. We analyze software repositories bounded in the *release history database*. We present reports to senior managers so that they know how much team leaders' estimations are real and close to actual spent time.

Chapter 7

Case Study

In this chapter, we illustrate our approach by using one case study. We have applied our approach to an industrial project which was in the maintenance phase. This experience illustrates how our analysis of the proposed *release history database* can help managers in software maintenance to apply more efficient management practices.

Verification of our results in the case study can be confirmed by the incoming bug reports, change requests, direct inspection of code, or change requests.

7.1 Choosing the case study

We have chosen a commercial application as our case study. This application is a platform and core for *mobile banking* from a company based in Toronto, Canada. It was first developed in Canada since 2007; then, the maintenance phase activities were distributed around the world. It was developed in a web and mobile environment,

and recently extended to iOS and Android. It has around 141,000 lines of code. The author was involved in the development and some releases in the maintenance of the application. The company gave read access of the *issue tracking system*, *code revision system* and *time entry system* to the author.

We had constraints when it comes to the selection of a suitable case study because we had to choose a software which met the following criteria:

1) The software had to be in the maintenance phase. Our focus is on improving management and efficiency in the maintenance; so we had to choose a software which had been developed completely and was in the maintenance phase.

2) Our potential case study needed to have all our required information in the software repositories. The information which we are looking to analyze exists in the different softoware repositories. Therefore, it was important that the maintained application had all required information in software repositories, even scattered in different places.

3) The maintenance team for our potential case study had to be well disciplined and organized in order to follow rules about producing enough logs and information as input for our analysis process. For instance, maintainers had to log all daily activities based on the issues existing in the *issue tracking system*s.

Figure 9 shows our case study's architecture. It is a J2EE application developed using the most popular open source framework and technologies. It is developed in Java [27] using Spring [56], Hibernate [25], Sitemesh [53], JSP [29], and MySql [44] as a Relational Database Management System.

Our case study software has been released in version 3.9.0 recently, but we chose

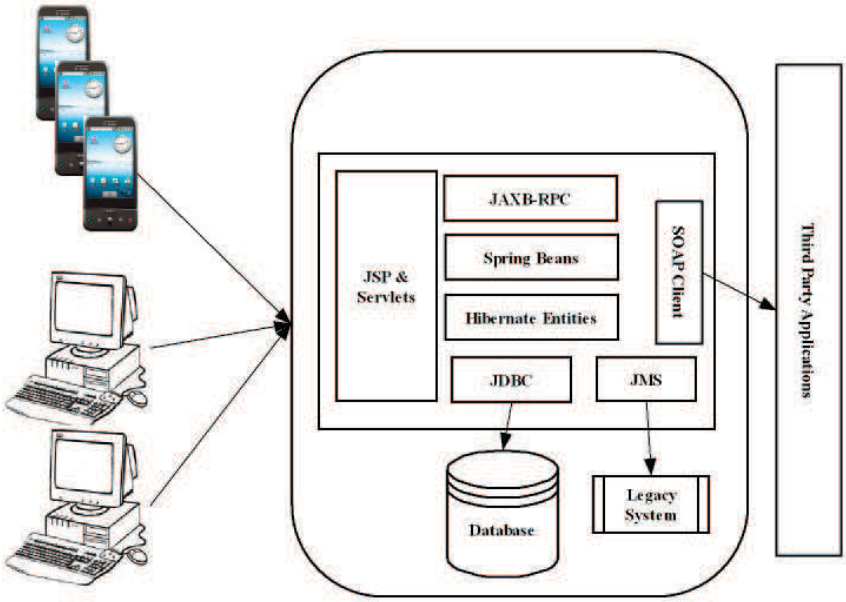

Figure 9: Architecture of a J2EE application that we used as a case study

the ten minor releases: 2.9.x, 2.10.x, 2.11.x, 2.12.x, 2.13.x, 3.4.x, 3.5.x, 3.6.x, 3.7.x and 3.8.x and their patch releases to examine our approach to it. In order to have analysis on software repositories, we need to build *release history database* with importing data from different sources. The .x in 2.9.x and other releases means that it incudes all patches/revisions (for example: 2.9.1, 2.9.2, 2.9.2 ...).

7.2 Building the release history database

In the first step, we built a proper release history database from the data which accumulated during these ten releases (2.9.x, 2.10.x, 2.11.x, 2.12.x, 2.13.x, 3.4.x, 3.5.x, 3.6.x, 3.7.x and 3.8.x) and their patch releases (i.e: 3.6.1,3.6.2, 3.6.3, ...). We ran the following three steps to build our *release history database*. We have used the tool described in Chapter 9 to build the *release history database*. In the following sections, we will show how we will build our *release history database* from the software repositories.

7.2.1 Importing data from the issue tracking system

The first step for building the *release history database* is to import data from the *issue tracking system*. Jira [28] is a commercial tool for task, bug, and issue management. Jira is a web based application which can run with different Relational Database Management Systems(RDBMS). MySql [44] is the RDBMS in our case study.

Jira keeps all information from the first day of the project, and we need to take the

information for the releases we need. We developed a component to read all existing data from Jira and import only the data for the releases we need to our *release history database*. We have described the table's schema in Figure 6.

7.2.2 Importing data from the code repository

In this step, we imported data from the code repository to our *release history database*. The Subversion [58] is being used as the *code repository* in our case study. As we mentioned before, the Subversion keeps the change history of codes and documents for all releases from the starting day of software. However, we only needed to have information and change logs for the releases we need, so we have imported only required data to our *release history database*.

7.2.3 Importing data from the time entry system

In our case study, software maintainers were using their own *time entry system* which they have called *TIES*. It is a web based application in which developers logged their daily activities. TIES keeps all information on its RDBMS. We developed a component to get only the required data associated with the releases. The developed component gets all the required information from *TIES* and populates it to our release history database's database. We need all time entry records for ten releases we chose. We now have all required information placed in our *release history database* in our database.

7.3 Analysis of release history database

By this time, we have built our *release history database* and it is ready to use for analyzing data to improve software maintenance management. In our case study, we were also aware that the maintenance team is located in different cities.

7.3.1 Retrieving risky objects

We have all the required data for Releases 2.9.x, 2.10.x, 2.11.x, 2.12.x, 2.13.x, 3.4.x, 3.5.x, 3.6.x, 3.7.x and 3.8.x. As we discussed before, code changes over a certain threshold could be an alert that new bugs could have been introduced into the software.

We ran queries against the *release history database* for code changes in Release 2.9.x, 2.10.x, 2.11.x, 2.12.x, 2.13.x, 3.4.x, 3.5.x, 3.6.x, 3.7.x and 3.8.x and the result is displayed in Table 15, 16, 17, 18, 19, 20, 21, 22, 23 and 24.

Based on our approach, we expected that these classes (classes with changes above the threshold) should have introduced new bugs for the next Releases. We are trying to verify it as follows: We have bug reports after each ten release; therefore, we will verify our approach with the analysis of the *release history database* by considering classes which are modified for bug fixes in each of ten releases and bug reports after each release. Our analysis, as shown in Table 25 shows how many percentages of bugs after each release are in the classes which were highlighted because those classes have changes above the threshold. As shown in Table 25 at least 51.29% of bugs in ten

Table 15: Some of classes or pages in Release 2.9.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
SpecialPhoneCFilter.java	Security	46.24%
PendingReferals.java	Schedules	38.32%
PendingOutboundMessages.java	Schedules	35.29%
ExhcnageRateService	Services	29.43%
LicenseeCommissionRateService.java	Services	25.32%
viewMyEarnings.jsp	Dashboard	24.55%

Table 16: Some of classes or pages in Release 2.10.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
RewardPointService.java	Reward Subsystem	51.20%
LifeCeycleListiner.java	Base Framework	50.10%
TransactionLimitService.java	Transaction Service	41.55%
TransactionActivityType.java	Transaction Service	38.97%
CashoutBeanService.java	Transaction Service	29.43%
InstalmentService.java	Services	26.83%

Table 17: Some of classes or pages in Release 2.11.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
InstallmentTransactions.java	Transaction Service	32.54%
LifeCeycleListiner.java	Base Framework	22.42%
PostRedirectGetListener.java	Base Framework	23.31%
referNewClient.jsp	Client Area	35.32%
PendingTransactions.java	Transaction Management	32.91%
agentPanel.java	Agent Area	29.32%

Table 18: Some of classes or pages in Release 2.12.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
AccessDecissionManager.java	Payment Subsystem	33.29%
SpeicalCashFilter.java	Base Framework	29.43%
SystemLedgerService.java	Base Framework	53.11%
VoucherService.java	Services	39.32%
refundPayment.jsp	Cleint Area	24.19%
PostRedirectListner.java	Base Framework	42.32%

Table 19: Some of classes or pages in Release 2.13.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
ServicePrevilageVoter.java	Payment Subsystem	45.32%
TransactionComisionTimer.java	Scheduler	29.56%
PayeeHandler.java	Payment Framework	33.17%
PayLimitHandler.jsp	Client Area	28.54%
RejectInvitionService.java	Referral Service	33.19%
TransactionHandler.java	Transaction Management	43.22%

Table 20: Some of classes or pages in Release 3.4.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
LostReportService.java	Reporting Subsystem	23.54%
PayeeManager.java	Payee Subsystem	37.41%
CommisionRateLogService.java	Base Framework	33.19%
SpeicalUserPermission.jsp	Security	45.32%
QuickPayService.java	Payee Subsystem	33.28%
BatchReferalService.java	Batch Management	25.31%

Table 21: Some of classes or pages in Release 3.5.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
BatchInstalmentsBean.java	Batch Management	38.29%
RefundPaymentService.java	Payment Subsystem	39.21%
ChangeCorrectionReportService.java	Base Framework	34.29%
referNewClient.jsp	Client Area	35.29%
PendingToTransactionTimer.java	Transaction Management	29.19%
SpecialPrivilageVoter.java	Security	43.22%

Table 22: Some of classes or pages in Release 3.6.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
InstallmentTransactions.java	Payment Subsystem	28.32%
LifeCeycleListiner.java	Base Framework	27.14%
PostRedirectGetListener.java	Base Framework	29.19%
referNewClient.jsp	Client Area	31.54%
PendingTransactions.java	Transaction Management	34.19%
agentPanel.java	Agent Area	34.75%

Table 23: Some of classes or pages in Release 3.7.0 with changes above a certain threshold.

Class or Page	Subsytem or feature	Change
HttpSender.java	Utils	33.29%
AgentCommissionRateDAO.java	Payee Subsystem	37.31%
OutboundMessageService.java	Base Framework	51.33%
referNewClient.jsp	Client Area	37.32%
PrivacyHandler.java	Security	29.32%
POSHandler.java	Agent Area	41.23%

experimented releases are from the code which is highlighted as *risky objects*. Therefore, querying for classes and pages for changes above the threshold will introduce *risky object*s into the next release.

7.3.2 Finding time consuming codes

We mentioned in Chapter 6 that our approach helps to identify time consuming codes, subsystems, or features in maintenance. Finding the most time consuming code in each release gives a chance for managers to revisit those codes or make more aggressive tests on those parts of the software, features or subsystems. Having more tests on time consuming codes will prevent the introduction of new bugs into the software in the next release.

In our case study, we analyzed the *release history database* for our case study. In the first step, we calculated the *average time to change a line of code in release* (ATCLiR) for releases from beginning of the project until the release candidate. In the next step, we analyzed the *release history database* for release 2.9.0, 2.10.0, 2.11.0, 2.12.0, 2.13.0, 3.4.0, 3.5.0, 3.6.0, 3.7.0 and 3.8.0 in order to identify classes, features and subsystems for which the times spent by the team was more than the average of *ATCLiR*s.

We are verifying our approach to show that the time consuming code may introduce bugs into the next release. For verification, we analyzed bugs that are raised in 2.9.0, 2.10.0, 2.11.0, 2.12.0, 2.13.0, 3.4.0, 3.5.0, 3.6.0, 3.7.0 and 3.8.0.

Table 26 shows how many percentages of bugs in each release comes from the

Table 24: Some of classes or pages in Release 3.8.0 with changes above a certain threshold.

Class or Page	**Subsytem or feature**	**Change**
SystemCommissionRatesHandle.java	Fees Subsystem	43.21%
GatewayLogServer.java	Base Framework	56.21%
CashoutService.java	Base Framework	25.32%
ProfitHandler.java	Client Area	29.32%
VouchersPrepaidsBean.java	Transaction Management	41.33%
SearchResultWrapper.java	Agent Area	29.92%

Table 25: Percentage of bugs coming from *risky objects* of releases

Release	**Percentage of bugs from risky objects**
Release 2.9.0	58.42 %
Release 2.10.0	63.60 %
Release 2.11.0	56.84 %
Release 2.12.0	73.12 %
Release 2.13.0	51.29 %
Release 3.4.0	63.29 %
Release 3.5.0	54.31 %
Release 3.6.0	63.81 %
Release 3.7.0	56.32 %
Release 3.8.0	63.26 %

classes which maintainer spent time more than the *ATCLiR* for previous releases. It shows at least 43.21% of the bugs came from the code for which the team has spent more than an average amount of time.

7.3.3 Recommending maintenance team

Our approach helps senior managers to choose, and coach, the maintenance team for each release. As we have mentioned in Chapter 6, developing software and maintaining software requires two different skill sets. Our approach makes suggestions to help senior managers and human resource (HR) managers choose the best developers for maintenance.

We analyzed the *release history database* to calculate introduced metrics: *average number of introduced bugs* and *average number of reopen bugs*. Table 27, 28, 29, 30, 31, 32, 33, 34, 35 and 36 shows our analysis result for fixes on releases 2.9.0, 2.10.0, 2.11.0, 2.12.0, 2.13.0, 3.4.0, 3.5.0, 3.6.0, 3.7.0 and 3.8.0.

Each of those metrics has its own weight; in our case we specified their *rank* based on the senior managers' point of view. They gave to the three metrics equal rank to the *average time to change a line of code in release*, *average number of introduced bugs* and *average number of reopen bugs*. The *rank* column is calcualted based on Formula 5. The maintainers with higher ranks are better choice for the mainteance team. Table 37 shows the bugs from the maintainers whose their rank are greater than five in each release. It shows maximum 45.31% of bugs are coming from maintainers that their rank's are at least five from ten.

Table 26: Bugs from *time consuming* code in different releases

Release	**Percentage of bugs from** *time consuming*
Release 2.9.0	53.72 %
Release 2.10.0	58.96 %
Release 2.11.0	56.60 %
Release 2.12.0	43.27 %
Release 2.13.0	74.51 %
Release 3.4.0	64.32 %
Release 3.5.0	53.34 %
Release 3.6.0	48.92 %
Release 3.7.0	54.32 %
Release 3.8.0	64.31 %

Table 27: Normalized metrics for maintainers for Release 2.9.0

Maintainers	**M1**	**M2**	**M3**	**Rank**
Maintainer 1	5.77	5.23	8.15	6.39
Maintainer 2	4.76	6.01	5.84	5.54
Maintainer 3	7.59	4.42	7.01	6.34
Maintainer 4	6.64	5.30	7.15	6.43
Maintainer 5	4.02	5.26	5.84	5.04

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 28: Normalized metrics for maintainers in Release 2.10.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	4.54	7.26	5.87	5.89
Maintainer 2	4.87	6.43	6.98	6.09
Maintainer 3	5.23	6.34	4.76	5.44
Maintainer 4	6.22	8.64	8.08	7.64
Maintainer 5	7.75	8.97	8.66	8.46

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 29: Normalized metrics for maintainers in Release 2.11.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	8.06	6.14	4.53	6.24
Maintainer 2	6.39	5.22	8.42	6.68
Maintainer 3	8.03	3.21	4.81	5.35
Maintainer 4	6.00	4.68	3.22	4.63
Maintainer 5	7.54	7.91	4.33	6.59

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 30: Normalized metrics for maintainers in Release 2.12.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	6.99	4.50	7.59	6.36
Maintainer 2	2.21	4.06	6.46	4.24
Maintainer 3	5.75	5.49	3.42	4.89
Maintainer 4	5.49	5.10	6.49	5.69
Maintainer 5	6.79	4.00	4.62	5.14

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 31: Normalized metrics for maintainers in Release 2.13.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	3.82	8.27	6.88	6.32
Maintainer 2	6.05	4.10	4.76	4.97
Maintainer 3	4.62	8.27	5.77	6.22
Maintainer 4	5.20	3.54	6.10	4.99
Maintainer 5	4.65	8.33	7.46	6.81

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 32: Normalized metrics for maintainers in Release 3.4.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	5.41	7.31	7.44	6.72
Maintainer 2	5.67	6.26	5.69	5.96
Maintainer 3	4.17	6.39	5.24	5.27
Maintainer 4	5.68	5.36	4.53	5.19
Maintainer 5	5.36	8.38	7.93	7.22

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 33: Normalized metrics for maintainers in Release 3.5.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	5.18	8.22	8.03	7.14
Maintainer 2	6.55	4.67	7.86	6.36
Maintainer 3	4.32	8.11	4.28	5.57
Maintainer 4	6.17	7.38	4.49	6.01
Maintainer 5	5.39	6.63	3.69	5.24

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 34: Normalized metrics for maintainers in Release 3.6.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	2.58	4.43	3.78	3.59
Maintainer 2	2.79	6.87	5.10	4.92
Maintainer 3	6.78	6.09	4.62	5.83
Maintainer 4	3.48	4.09	3.90	3.82
Maintainer 5	5.48	5.43	3.78	4.90

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 35: Normalized metrics for maintainers in Release 3.7.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	5.60	5.29	5.08	5.32
Maintainer 2	5.69	7.21	6.80	6.57
Maintainer 3	2.87	3.20	7.09	4.39
Maintainer 4	3.81	4.41	5.20	4.48
Maintainer 5	2.84	3.76	6.36	4.32

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)
M3:Average of reopened bugs in 100 lines of code per release (Normalized)

Table 36: Normalized metrics for maintainers in Release 3.8.0

Maintainers	M1	M2	M3	Rank
Maintainer 1	2.29	7.29	4.51	4.70
Maintainer 2	4.90	5.36	7.25	5.84
Maintainer 3	6.08	6.25	5.76	6.00
Maintainer 4	3.28	7.75	5.18	5.41
Maintainer 5	3.45	7.23	7.86	6.18

M1:Average time to change a line of code in release (Normalized)
M2:Average of new introduced bugs in 100 lines of code per release (Normalized)

7.3.4 Recommending maintenance location

As we discussed in Chapter 6, our approach helps senior managers to choose a location for maintenance. We have verified our approach for releases 2.9.0, 2.10.0, 2.11.0, 2.12.0, 2.13.0, 3.4.0, 3.5.0, 3.6.0, 3.7.0 and 3.8.0 of our case study. Maintenance activities for these releases were distributed in different cities around the world.

Our approach suggests locations to the managers to host the maintenance phase. We will try to recommend the location by considering the *average time to change a line of code in release*, *average of new introduced bugs in 100 lines of code per release* and *average of reopen bugs in 100 lines of code per release*. Table 38, 39, 40, 41, 42, 43, 44, 45, 46 and 47 shows normalized metrics for different locations based on the data in the *release history database* captures for releases 2.9.0, 2.10.0, 2.11.0, 2.12.0, 2.13.0, 3.4.0, 3.5.0, 3.6.0, 3.7.0 and 3.8.0. We gave equal rank to the calculated metrics and *rate per hour*. The *rank* column in these tables for each release is calculated based on Formula 7. The locations with higher ranks are better choice for the mainteance location.

In the next step, we verify our approach for recommending the *location*. We have analyzed *release history database* for reported bugs after the releases grouped by different locations and Table 48 shows bug reports after each release. It shows maximum 45.31% of bugs are comming from the locations which are recommended by our approach.

Table 37: Bugs from recommmended maintainers in different releases

Release	**Percentage of bugs**
Release 2.9.0	29.32 %
Release 2.10.0	45.31 %
Release 2.11.0	29.31 %
Release 2.12.0	31.66 %
Release 2.13.0	25.91 %
Release 3.4.0	39.62 %
Release 3.5.0	35.31 %
Release 3.6.0	18.85 %
Release 3.7.0	14.17 %
Release 3.8.0	34.31%

Table 38: Recommending location for Release 2.9.0

Location	**M1**	**M2**	**M3**	**M4**	**Rank**
New York	3.52	7.38	8.33	1.00	5.06
Paris	3.31	7.34	5.82	4.60	5.27
Toronto	3.38	4.03	7.54	6.40	6.38
Montreal	3.13	7.84	3.68	6.76	5.35
Hidarabad	5.05	7.18	7.74	10	6.15

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 39: Recommending location for Release 2.10.0

Location	M1	M2	M3	M4	Rank
New York	6.01	4.25	7.86	1.00	4.78
Paris	6.55	7.07	7.53	4.60	6.44
Toronto	6.35	5.12	3.62	6.40	5.37
Montreal	3.81	3.91	6.21	6.76	5.17
Hidarabad	6.34	6.11	8.44	10	7.72

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 40: Recommending location for Release 2.11.0

Location	M1	M2	M3	M4	Rank
New York	5.34	7.30	8.32	1.00	5.49
Paris	2.26	6.13	5.67	4.60	4.67
Toronto	6.86	8.00	5.56	6.40	6.71
Montreal	5.46	7.31	6.32	6.76	6.46
Hidarabad	3.07	4.53	6.30	10	5.97

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 41: Recommending location for Release 2.12.0

Location	M1	M2	M3	M4	Rank
New York	5.25	6.35	8.32	1.00	5.49
Paris	4.67	4.21	6.04	4.60	4.67
Toronto	6.46	5.02	5.32	6.40	6.71
Montreal	4.64	5.73	8.30	6.76	6.46
Hidarabad	2.34	5.51	5.43	10	5.82

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 42: Recommending location for Release 2.13.0

Location	M1	M2	M3	M4	Rank
New York	6.30	7.30	7.52	1.00	5.53
Paris	2.97	6.25	5.34	4.60	4.79
Toronto	3.38	4.69	7.01	6.40	5.37
Montreal	6.89	6.66	4.44	6.76	6.22
Hidarabad	2.31	5.32	4.56	10	5.55

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 43: Recommending location for Release 3.4.0

Location	M1	M2	M3	M4	Rank
New York	4.06	4.91	3.75	1.00	3.43
Paris	4.72	6.93	6.38	4.60	5.66
Toronto	2.74	3.72	5.62	6.40	4.62
Montreal	4.88	7.31	6.31	6.76	6.31
Hidarabad	3.00	3.55	6.99	10	5.89

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 44: Recommending location for Release 3.5.0

Location	M1	M2	M3	M4	Rank
New York	7.98	6.43	8.01	1.00	5.86
Paris	8.02	8.09	4.91	4.60	6.41
Toronto	8.08	5.08	4.45	6.40	6.00
Montreal	2.04	7.95	7.05	6.76	5.95
Hidarabad	4.56	5.17	3.45	10	5.80

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 45: Recommending location for Release 3.6.0

Location	M1	M2	M3	M4	Rank
New York	5.40	6.77	8.90	1.00	5.52
Paris	4.63	8.43	5.73	4.60	5.85
Toronto	6.13	3.58	7.16	6.40	5.82
Montreal	4.67	5.26	7.79	6.76	6.10
Hidarabad	2.17	4.96	3.53	10	5.17

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 46: Recommending location for Release 3.7.0

Location	M1	M2	M3	M4	Rank
New York	6.32	5.97	7.46	1.00	5.19
Paris	4.54	6.11	4.16	4.60	4.85
Toronto	6.16	4.01	3.54	6.40	5.03
Montreal	6.39	4.32	4.27	6.76	5.43
Hidarabad	4.97	5.27	5.55	10	6.45

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

Table 47: Recommending location for Release 3.8.0

Location	M1	M2	M3	M4	Rank
New York	7.43	8.21	8.32	1.00	6.24
Paris	5.87	6.52	4.04	4.60	5.26
Toronto	3.43	7.04	6.45	6.40	5.83
Montreal	3.16	8.43	7.48	6.76	6.46
Hidarabad	3.42	5.32	4.53	10	5.82

M1: Average time to change a line of code in release (Normalized)
M2: Average of new introduced bugs in 100 lines of code per release (Normalized)
M3: Average of reopened bugs in 100 lines of code per release (Normalized)
M4: Maintenance Rate (Normalized)

7.3.5 Fixing bugs, or refactoring, or developing from scratch?

As we discussed on Chapter 6, making the right decision for the future of the software in maintenance is one of the hardest decisions in the maintenance phase.

In our case study, we chose release 2.9.0, 2.10.0, 2.11.0, 2.12.0, 2.13.0, 3.4.0, 3.5.0, 3.6.0, 3.7.0 and 3.8.0. We analyzed the *release history database* to give a suggestion for future strategy of the software. We analyzed *release history database* to calculate number of different type of bugs which we have introduced in Chapter 6 including *bugs from origin software*, *reopened bugs from previous releases*, and *new introduced bugs on fixes*. Table 49 shows our proposed recommendation considering the three different type of bugs *bugs from origin software*, *reopened bugs from previous releases* and *new introduced bugs on fixes*.

7.3.6 Accurate estimation

We try to use our analysis in order to veify how good are managers in estimating of maintenance tasks and how good are maintainers in fixing bugs on time. Table 50 and 51 shows the estimated time, actual time and error percentage for maintainers and team leaders.

Table 48: Bugs from recommmended locations in different releases

Release	**Percentage of bugs**
Release 2.9.0	29.32 %
Release 2.10.0	45.31 %
Release 2.11.0	29.31 %
Release 2.12.0	31.66 %
Release 2.13.0	25.91 %
Release 3.4.0	39.62 %
Release 3.5.0	35.31 %
Release 3.6.0	18.85 %
Release 3.7.0	14.17 %
Release 3.8.0	34.31%

Table 49: Maintenance strategy proposed for 10 releases

Release	**B1**	**B2**	**B3**	**Recommended strategy**
Release 2.9	22	56	18	Remaining in maintenance
Release 2.10	22	51	2	Remaining in maintenance
Release 2.11	16	56	16	Remaining in maintenance
Release 2.12	13	21	10	Remaining in maintenance
Release 2.13	11	21	4	Remaining in maintenance
Release 3.4	19	37	21	Remaining in maintenance
Release 3.5	15	21	14	Remaining in maintenance
Release 3.6	9	43	11	Remaining in maintenance
Release 3.7	9	36	8	Remaining in maintenance
Release 3.8	6	39	3	Remaining in maintenance

B1: Number of bugs from the orogin software
B2: Number of reopened bugs from previous releases
B3: Number of new introduced bugs on fixes

Table 50: Estimation accuracy for maintainers in ten releases

Maintainers	**Total Estimation Time**	**Total Spent Time**	**Error**
Maintainer 1	1254:00	1489:00	18.74 %
Maintainer 2	1764:00	1983:00	12.38 %
Maintainer 3	1811:00	2182:00	20.47 %
Maintainer 4	1698:00	2300:00	35.40 %
Maintainer 5	1470:00	1932:00	31.42 %

Table 51: Estimation accuracy for team leaders in ten releases

Team Leaders	**Total Estimation Time**	**Total Spent Time**	**Error**
Team Leader 1	6789:00	7932:00	16.84 %
Team Leader 2	5981:00	8932:00	49.34 %

Chapter 8

Related Work

Our work is based on retrieving and analyzing information from a release history database in order to help maintainers and managers for their next releases. Our proposed *release history database* can help them to be more productive and to keep within their time schedule and budget.

Some of the latest studies are based on the manipulation of historical data in order to build prediction models. They use historical data to predict bugs and the location of defects, and managers now have more chances to allocate resources for testing towards these identical areas [15]. Those studies used source codes more than any other sources in release histories. Some of them are focused on the data coming from information about bugs, issues, and change requests, which are located in the release history database. Other groups have focused on visualizing release history database to provide more understanding about making decisions that improve the software maintenance phase.

Patrick *et al.* [32] presented an approach to detect visual patterns in the data that are stored in the issue tracking system. They have presented different views such as *Polymetric View*, and the *Phase View*. The Polymetric View consists of a box whose width is the value of the *estimated effort* and whose height represents the *actual effort*. In this way, managers and developers can have a quick and effective overview of the quality of the estimation. Square boxes show balanced estimates. Thin tall boxes show *underestimated* tasks that need more resolution efforts. On the other hand, *overestimated* predictions feature more effort than is actually needed. The second view is the *Phase View*, which is a visualization of the time which is being spent on process steps. They consider *submitted*, *in analysis*, *in resolution* and *in evolution* as the process steps. The *Phase View Visualization* is a third view which displays a visualized statistics for the Process Life-cycle Sequence, which includes *submitted*, *analysis*, *resolution*, and *evaluation*. This view displays the time which is spent for each Process Life-cycle and helps to improve the process and speed up the project. The difference between this thesis and their work is that they do not bring up any information from the code repository.

Harald *et al.* [21] worked on having 2D and 3D visualizations of the Software Release History in order to make is easier to understand large softwares systems, which contains *Time*, *Structure*, and *Attribute*. Time is representing Release Sequence Number (i.e: 1.0, 1.1 ...), and the structure shows the decomposition of systems which are modules and their relationships. Finally, the attributes are the version number, size, complexity, and defect density which are associated with system modules. This

data was extracted from the source code stored in the database, and then, based on the stored data, they were presented in 2-D and 3-D graphs . In the 3-D graphs, z represents the Time, which is the *Release Sequence Number*. For each Release Sequence Number, we have a 2-D or 3-D graph which is associated with the structure. In each structure, the detailed diagram shows attributes. A hierarchal visualization of software based on the Software Release History allows developers, maintainers, and managers to visualize and to compare multiple releases. The authors preformed a pioneering study on the visualization Software Release History, but they didn't consider more detailed metrics for retrieving data from the database.

Tang *et al.* [35] proposed a practical software maintenance model which takes into account that 50-80 percent of budget is spent on the maintenance phase. They have tried to improve the maintenance process. They have defined four roles in the maintenance organization: user, coordinator, decision-maker, and maintenance operator. Every Maintenance Request Form(MRF) experiences six states: accepted, analyzing, waiting, maintenance, reviewing, and finished. They have found that it is not enough to manage the flow and track a MRF because it will not help to improve the maintenance process. What they need is to have some measurements to help them resolve more questions.They have defined three measures which come from the release history information: 1) average time of a MRF 2) average cost of a MRF 3) number of MRFs in a week. These three metrics help managers to build more productive and efficient teams. For example, an increased 'average time of MRF' as compared to previous releases, could signal the lack of efficiency in the maintenance

team. The percentage of tasks which are finished on time or the percentage of tasks which exceed budget are two other measures which are provided in their paper on the maintenance phase. Their focus was practically on management issues and proposing a practical maintenance model. We have the same concern which they had about management of the maintenance phase, but we have involved more sources on the metrics which we have provided. For example, they didn't use the code repository in their status while we have it in our study.

Michael *et al.* [20] introduced an approach for populating a Release History Database that comes from Version Control and Bug Tracking applications. They have used a SQL database and scripts for retrieving information from those two sources. They have implemented their approach on CVS as their Version Control and Bugzilla as their Bug Tracking System. They have imported information from these two sources into a single schema in a RDBMS. Then, as a case study, they evaluated their approach for the timescale, historical, and coupling aspects of the software. The flaw in their work is that they have not provided any predefined queries or metrics for use in software maintenance or software evolution. They have just provided a platform which contains data and software developers should retrieve information based on their software knowledge.

Michael *et al.* [19] pointed out that the data derived from the software's evolution enables engineers to know more from the past and anticipate future changes. They have tried to bundle together pieces of problem report information which correspond to a certain feature and to determine out the dependencies between files. Their

contribution was towards tracking features by analyzing and reporting bug report data which comes from a release history database. Moreover, they have visualized the tracked features by emphasizing their non apparent dependencies. They have used their release history database to help maintainers to locate features.

Chapter 9

Automation and tool support

For a proof of concept, we have implemented a prototypical web based application tool to support the proposed methodology. This tool allows maintainers and managers to build their own release history database and then, in the second step, it lets maintainers and managers run queries against the *release history database*.

This implementation could be done in different ways; for example, as a stand alone application, IDE plug-in, or web application. We chose the third approach for two reasons: 1) A web based application can be used by different people around the world in teams without installing any tool into their local computers 2) Our required data is on the servers and usually takes an enormous amount of space; we would not be able to replicate data to local personal computers.

The tool is architecturally developed based on the J2EE architecture. We have used three layer architecture, and we have used Model-View-Controller (MVC) architecture. MVC is particularly well-suited for interactive web applications. We have

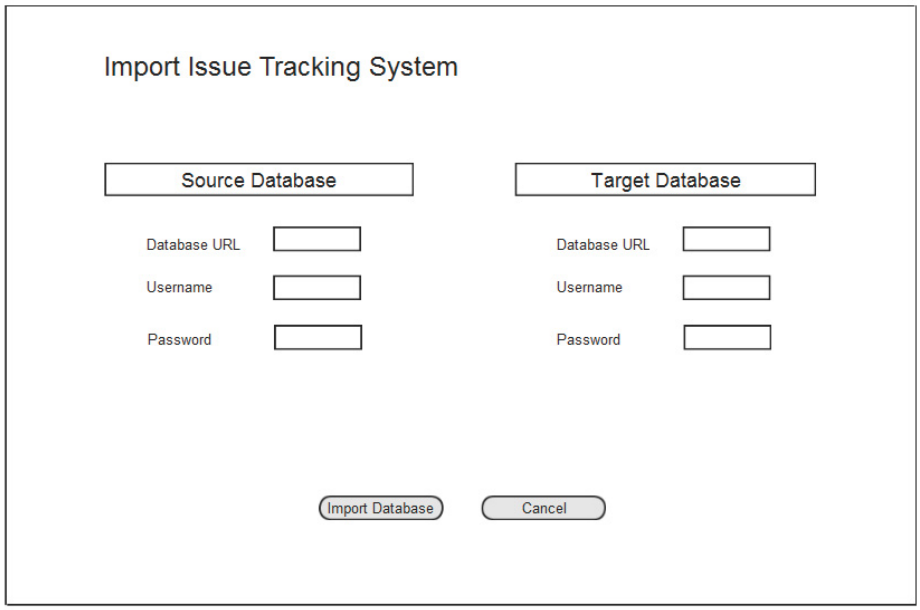

Figure 10: Choose import issue tracking system to import.

used Tomcat [60] as our application server, and MySql [44] as our Relational Database Managment System.

The tool is composed of the following components:

Import component of issue tracking system: This component allows maintainers to import the issue tracking system to our *release history database*. Figure 10 displays the page which allows the maintainer to choose the source and target database to import. In the next step, the maintainer chooses the right mapping between the source and target fields. In Figure 11, we need to choose the right fields for mapping fields to the *issues* table in our proposed schema.

Import component of code repository: This component is responsible for importing data from the *code repository* to our release history database. Figure 12

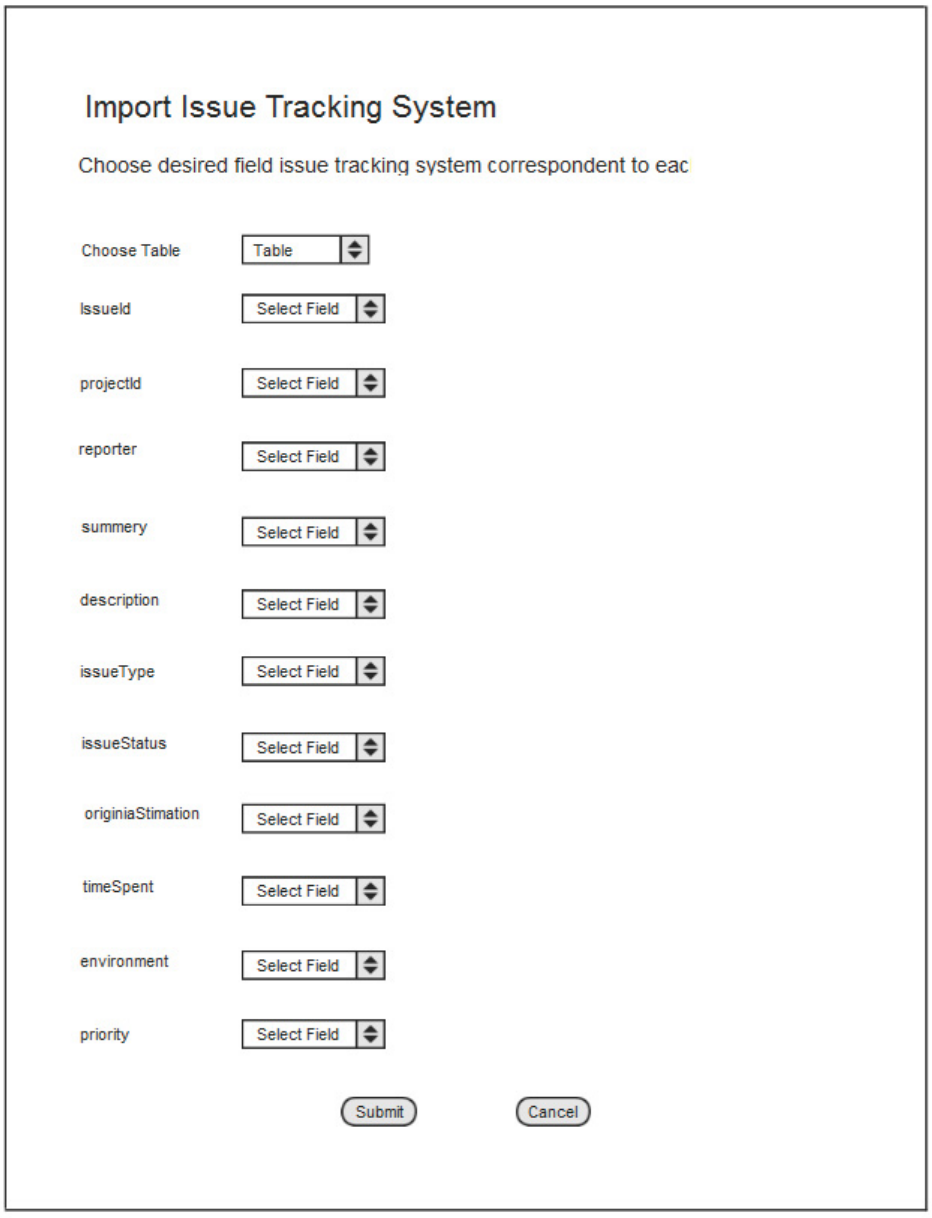

Figure 11: Choose the right fields to import into the issues table.

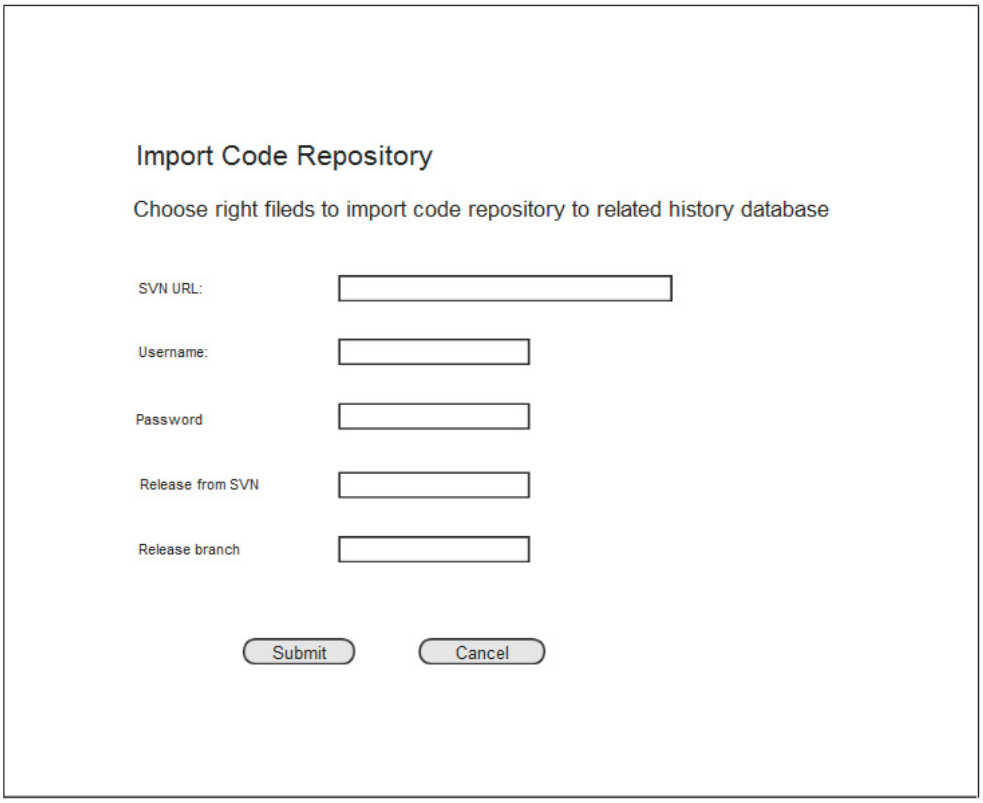

Figure 12: Choose code repository specifications to import.

shows the page where maintainers may enter the code repository information in order to import it to our *release history database*. The tool allows maintainers to import only data which is for the specific version in the code repository. The reason is that we do not want to import huge amounts of data pertaining to all of the development and maintenance history. This tool supports the SVN [59] code repository and the tool could be extended in order to support more code repositories in the software industry.

Import time entry system:. This component is used to import data from the time entry system to the release history database. In Figure 13, maintainers can

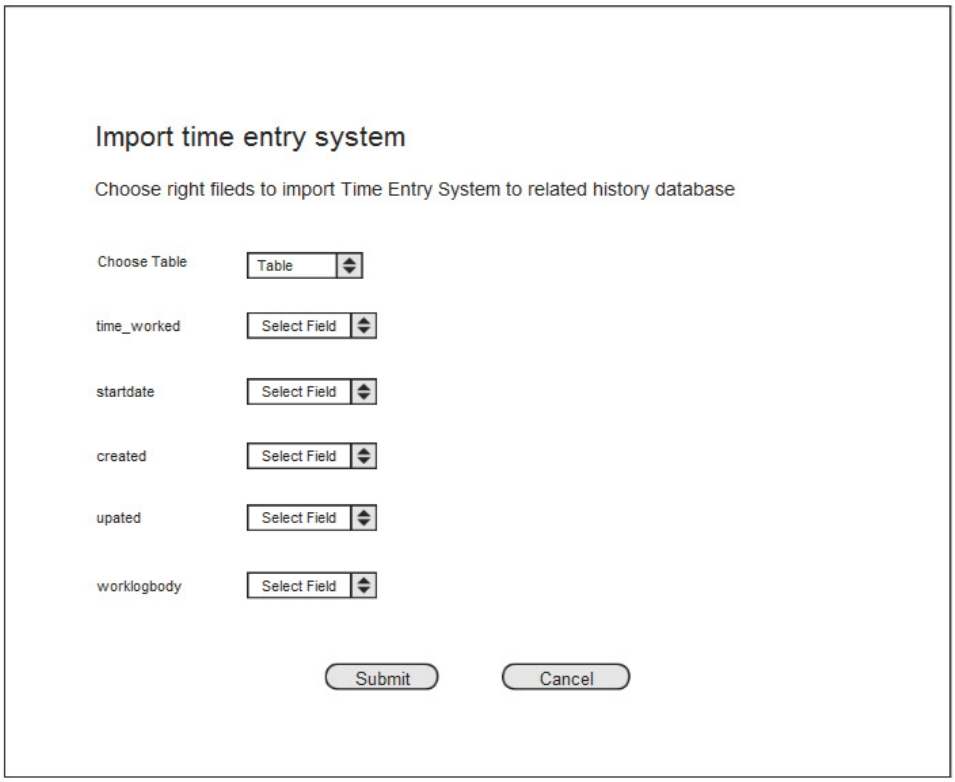

Figure 13: Choose time entry specifications to import.

choose source fields in order to import data to the *worklog* table.

Executing queries: There is a page for executing queries. In Figure 14, maintainers can specifiy the release and type of the query to execute.

Our developed tool supports a typical maintenance environment and should be customized for any other maintenance environment.

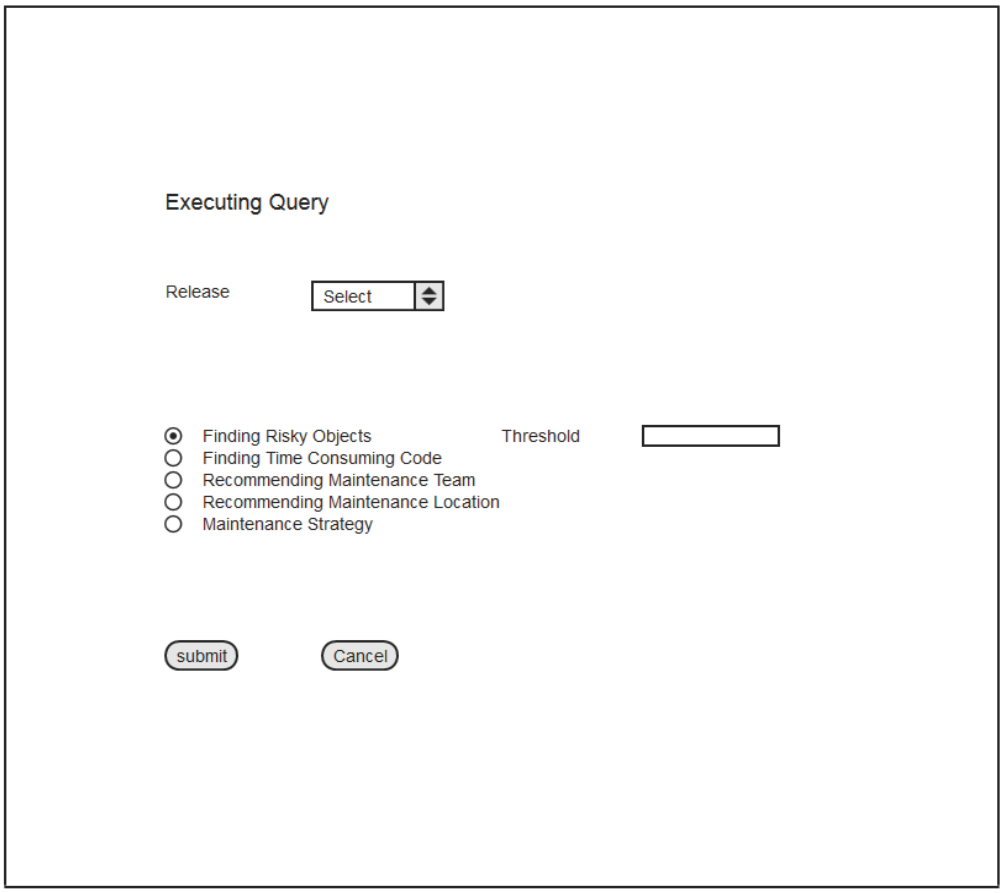

Figure 14: Executing queries

Chapter 10

Conclusion and recommendations for future work

10.1 Summary and conclusion

In this dissertation, we have proposed an approach towards analyzing a release history database in order to improve performance and management in software maintenance. The first step in our approach was building a release history database from the data that is hidden in different resources during the development and maintenance phase of the software. Secondly, we have defined new metrics derived from the analysis on the *release history database* in order to improve efficiency of software maintenance. We chose three important resources in the software environment and configuration management which we used to build our release history database. Those were issue tracking system, code repository, and time entry system. We then built a powerful

association between the various data retrieved from release histories. Integrating data driven from these three different resources allowed us to make some innovations for managers in software maintenance. By integrating our findings managers would be able to have a better general understanding, based on what heppened in the last releases of the software. Then, the maintenance team will have a clear and real picture of the maintenance phase. We believe that this approach can aid managers and team leaders in software maintenance to improve the quality and performance. Our approach will not add any load or complexity to the maintenance or to the software process and will not add any overhead to the software maintenance.

Analysis against the proposed *release history database* provided us with metrics including *average time to change a line of code in release*, *average of new introduced bugs in 100 lines of code per release*, and *average of reopened bugs in 100 lines of code per release*. We try to improve management in software maintenance using those metrics. Improvements in maintenance include: finding and highlighting *risky objects* which may introduce new bugs to the software; reporting *time consuming code* in the software which may also introduce new bugs, recommending team and location in order to host maintenance activities, proposing best solution for future maintenance, helping to have more accurate estimation and visualizing the product line.

10.2 Recommendations

For future work, we intend to conduct investigations the inclusion of code merges from branches to the trunk. This will help us to have an accurate and precise snapshot of what happens during branches. Another interesting subject for future work would be to have more focus on the information visualization of our proposed release history database. In our study, we used *centralized version control system*, however *distributed version control system* can be investigated for future work. Moreover, excluding non-significant changes like comment updates, indentations and whitespaces can be considered as future works [30].

Another possible direction for future work would involve using more resources to build the release history database in order to have more accurate and more meaningful data; and so that managers will have more comprehensive reports and analyses available to them.

Another interesting subject for future work would be to focus on bug prediction and the locations of future bugs using our release history database; this could be a chance for managers to put more testing efforts into those locations. Involving more resources in building the release history database will result in more accurate and specific bug predictions.

These are a set of directions related to this research that one can follow as the future work on this research.

Appendix A

Questionnaire for threshold

In this Appendix, we describe the questionnaire questions, audience and results of questionnaire we executed for specifying the *threshold* described in Section 6.3.1.

A.1 Audience

We executed the survey from fifteen managers as follow: five development team leaders, five QA team leaders, and five delivery managers. We chose these leaders among the people who have been in maintenance teams for at least three years and were involved in maintenance continuously for at least the last eighteen months. Audiences were chosen from three different companies in North America, Europe and Asia.

A.2 Survey Question

The questionnaire had following questions:

1- How many years you have been in the software industry?

2- How many years you have been involved in the software maintenance?

3- Do you agree that maintainers do not need to make huge change in code for fixing a bug? and making huge change will introduce new bugs to the software?

4- If your answer to the question three is "yes", then do you think how many percentages of code change is acceptable to not consider code change as root of new introduced bugs in the software?

We collected the answers via email and in the next Section, we will present the questionnaire result.

A.3 Questionnaire result

After sending questionnaire, we received answers and Table 52 shows the result

Table 52: Questionnaire result for determining *threshold*.

Questions	**Question result**
Question One	minimum: 7 years, maximum: 15 years, average: 8 years
Question Two	minimum: 3 years, maximum: 7 years, average: 4.2 years
Question Three	Yes: 14, No:1
Question four	minimum: 5%, maximum: 45%, average: 24%

The questionnaire result showed us that the threshold for determining *risky objects* is 24%.

Bibliography

[1] Alail Abran and Hong Nguyenkim. Analysis of maintenance work categories through measurement. *International Conference on Software Maintenance*, 0:104–113, 1991.

[2] Atlassian. Website: http://www.atlassian.com.

[3] Marla J. Baker and Stephen G. Eick. Visualizing software systems. In *Proceedings of 16th International Conference on Software Engineering, ICSE-16.,*, pages 59 –68, may 1994.

[4] Victor Basili, Lionel Briand, Steven Condon, Yong-Mi Kim, and Walclio L. Melo. Understanding and predicting the process of software maintenance releases. *International Conference on Software Engineering*, 0:464, 1996.

[5] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering International Conference on Software Maintenance*, 22(10):751 – 761, oct 1996.

[6] Victor R. Basili and Dieter Rombach. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering International Conference*, 14(6):758–773, 1998.

[7] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[8] Ronald Berlack. *Software Maintenance Management*. John Wiley Sons, Inc., Boston, MA, USA, 2002.

[9] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Proceedings of Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pages 11–18, New York, NY, USA, 2007. ACM.

[10] Rational Team Concert. Website url: https://jazz.net/products/rational-team-concert.

[11] Marcela Genero Coral Calero, Mario Piattini. Method for obtaining correct metrics. *In Proceedings of 3rd International Conference on Enterprise and Information Systems (ICEIS 2001)*, 29(6):779 – 784, 2001.

[12] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 31 –41, may 2010.

[13] Brian De Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? In *Cooperative and Human Aspects on Software Engineering, 2009. CHASE '09. ICSE Workshop on*, pages 36 –39, may 2009.

[14] DrProject. Website url: https://stanley.cdf.toronto.edu/drproject/drproject.

[15] Jayalath Ekanayake, Jonas Tappolet, Harald C. Gall, and Abraham Bernstein. Tracking concept drift of software projects using defect prediction quality. In *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*. IEEE Computer Society, 2009.

[16] Mahmoud O. Elish and Mojeeb Al-Rahman Al-Khiaty. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Empirical Software Engineering (EMSE)*, 2011.

[17] Sinan Eski and Feza Buzluca. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In *Proceedings of Fourth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 566–571, 2011.

[18] Letha H Etzkorn, Sampson E Gholston, Julie L Fortune, Cara E Stein, Dawn Utley, Phillip A Farrington, and Glenn W Cox. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, 46(10):677 – 687, 2004.

[19] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. *Working Conference on Reverse Engineering*, 0:90, 2003.

[20] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. *IEEE International Conference on Software Maintenance*, 0:23, 2003.

[21] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: the use of color and third dimension. pages 99 –108, 1999.

[22] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, pages 113–122, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Hackystat. Website url: http://code.google.com/p/hackystat/.

[24] Ahmed E. Hassan and Richard C. Holt. The top ten list: dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263 – 272, sept. 2005.

[25] Hibernate. Website: http://www.hibernate.org.

[26] Lyndon Hiew, Gail C. Murphy, and John Anvik. Who should fix this bug? *Software Engineering, International Conference on*, 0:361–370, 2006.

[27] Java. Website: http://www.oracle.com/technetwork/java/index.html.

[28] Jira. Website: http://www.atlassian.com/software/jira/.

[29] JSP. Website: http://www.oracle.com/technetwork/java/javaee/jsp/index.html.

[30] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 351–360, New York, NY, USA, 2011. ACM.

[31] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Patrick Knab, Martin Pinzger, and Harald Gall. Visual patterns in issue tracking data. In Jrgen Mnch, Ye Yang, and Wilhelm Schfer, editors, *New Modeling Concepts for Todays Software Processes*, volume 6195 of *Lecture Notes in Computer Science*, pages 222–233. Springer Berlin / Heidelberg, 2010.

[33] Maximilian Kögel. Towards software configuration management for unified models. In *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, CVSM '08, pages 19–24, New York, NY, USA, 2008. ACM.

[34] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gal Guhneuc, and Giuliano Antoniol. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Journal of Software: Evolution and Process*, pages 45–59, 2012.

[35] Tang Li, Mei YongGang, and Ding JianJie. Metric-based tracking management in software maintenance. *International Workshop on Education Technology and Computer Science*, 1:675–678, 2010.

[36] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[37] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management:A study of the management of computer application software in 487 data processing organizations.* Addison-Wesley Longman Publishing Co., Inc., 1980.

[38] Alok Mishra Liguo Yu. Risk analysis of global software development and proposed solutions. *IEEE Transactions on Software Engineering*, 51:89–98, 2010.

[39] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 5–9, Washington, DC, USA, 2007. IEEE Computer Society.

[40] Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno. Spam filter based approach for finding fault-prone software modules. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 503–512, New York, NY, USA, 2002. ACM.

[42] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance*, pages 120–130, 2000.

[43] Kjetil Molokken and Magne Jorgensen. A review of software surveys on software effort estimation. In *Proceedings of International Symposium on Empirical Software Engineering, ISESE 2003*, pages 223 – 230, sept.-1 oct. 2003.

[44] MySql. Website url: http://www.mysql.com/.

[45] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284 – 292, may 2005.

[46] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[47] Thanh . Nguyen, Timo Wolf, and Daniela Damian. Global software development and delay: Does distance still matter? In *IEEE International Conference on Global Software Engineering, ICGSE 2008.*, pages 45 –54, aug. 2008.

[48] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[49] Dien Dean. Phan. Information systems project management: An integrated resource planning perspective model. The University of Arizona.

[50] Elke Pulvermueller, Andreas Speck, James Coplien, Maja DHondt, and Wolfgang De Meuter. Feature interaction in composed systems. In kos Frohner, editor, *Object-Oriented Technology*, volume 2323 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2002.

[51] Mookerjee Suresh P. Sethi Qi Feng, Vijsay S. Optimal policies for the sizing and timing of software maintenance projects. *European Journal of Operational Research*, 172(3):1047–1066, 2006.

[52] Adrian Schroter. Predicting defects and changes with import relations. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 31–8, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Sitemesh. Website: http://www.opensymphony.com/.

[54] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[55] Ian Sommenville. Software engineering. In *Software Engineering- Ninth Edition*, CVSM '08, New York, NY, USA, 2009. Addison-Welsley.

[56] Spring. Website: http://www.springsource.org/.

[57] Krishnamoorthy Srinivasan and Douglas Fisher. Machine learning approaches to estimating software development effort. *Software Engineering, IEEE Transactions on*, 21(2):126 –137, feb 1995.

[58] Subversion. Website http://subversion.apache.org.

[59] SVN. Website url: http://tortoisesvn.tigris.org.

[60] Tomcat. Website: http://tomcat.apache.org/.

[61] Michiel van Genuchten. Why is software late? an empirical study of reasons for delay in software development. *IEEE Transactions on Software Engineering*, 17(6):582 –590, jun 1991.

[62] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–4, Washington, DC, USA, 2007. IEEE Computer Society.