2013

Identification and analysis of chunks in software projects

RACHANA S. KONERU
Iowa State University

Identification and analysis of chunks in software projects

by

Rachana S. Koneru

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

David M. Weiss, Major Professor

Robyn R. Lutz

Leslie Miller

Mack Shelley

Iowa State University

Ames, Iowa

2013

DEDICATION

I dedicate this thesis and all the work that went into it to my parents, who held me through my little successes and my downfalls alike. A special feeling of gratitude to my loving mother, Lakshmi Koneru, whose undying love and encouragement gave me the courage to pursue my dreams in a distant land. She is the strongest woman I know, who taught me not to back down from anything I believe in. I owe her the strong independent woman that I am today. My deepest love and admiration to my father and friend, Bhogeswara Rao Koneru, whose constant support, advice, and motivation pushed me towards my goals. He showed me how to live life without any regrets and taught me that failures can only bring out the best in us. He is my role model and my hero.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGEMENTS

There are a lot of people without whom I would not have been able to complete this thesis and to whom I am greatly indebted.

I wish to thank, first and foremost, my Professor, Dr. David Weiss for his guidance, encouragement, and support throughout my M.S. program. He motivated me in the right direction with his high intellect and deep insights during the entire course of my research. I would like to specially thank him for his patience and composure at times when we had little hope of any progress. I attribute the persistence and dedication that I put into this work to him.

I would like to thank my committee members Dr. Robyn Lutz, Dr. Leslie Miller, and Dr. Mack Shelley for their time and valuable inputs throughout this research. I thank and extend my sincere appreciation to my colleague Jeff St. Clair for his vast technical expertise, and valuable time and efforts in this research. He is the best geek I've ever met.

I would like to thank my amazing brothers, Rahul Koneru and Anirudh Pullela, for being my stress-busters, and standing by me through everything. They are my best cheerleaders. I would also thank my best friends, Priyanka Reddy, Sudha Lahari, Disha Reddy, and Ankita Bhangadiya, for being a family to me and for their constant love and support since the day we met. Finally, I would like to thank all my friends – Loukya, DT, CT, Satish, and Venky for making Ames feel like home.

ABSTRACT

Most software systems undergo continuous change in different phases of their lifecycle such as development or maintenance. Ideally, such changes should correspond to a system's modular design. However, some changes span across more than one component thereby resulting in discrepancies between design and implementation. In such cases, making a change to one component requires changes to other components leading to an increase in time and effort to make changes to a software system as it evolves.

This thesis investigates: 1) an approach to observe how components change together by identifying tightly coupled changes known as chunks, 2) whether there are any trends in how chunks evolve over time, and 3) whether chunks can help identify design issues in a software system.

In this work, a family of algorithms is proposed to identify independently changing chunks from change data obtained from mining version history repositories of three large software systems – Moodle, Eclipse, and Company-X. A comprehensive analysis of certain characteristics of the resulting chunks is conducted. In addition, evolution of chunks with respect to size in terms of number of files within a chunk, and percentage of changes crossing a chunk are studied. Lastly, a pragmatic interpretation of the results to identify necessary code refactoring or system redesign is presented.

The findings of this work show that the percentage correlation of a chunk decreases with an increase in the number of inter-component or subsystem couplings. We also observed that there is no association between chunk size and percentage correlation. Identifying chunks that merge helps in a better understanding of the inconsistencies between how a system is designed for change and how it is actually changed, and to identify areas of a system that require refactoring or redesign. Additionally, identifying stable chunks can provide insights into how size and percentage correlation of the corresponding empirical components change over time.

CHAPTER 1. INTRODUCTION

Software undergoes change throughout its life cycle either in the form of development or maintenance until it is not viable anymore. Often changes to a software system are made by developers, who do not understand its design, which leads to discrepancies in design and implementation thereby causing system degradation [Parnas, 1994]. Such changes result in an increase in time and effort to make future changes to the software system as it evolves. As such, design for change is a key aspect in the design of sustainable software systems. The underlying notion for the application of such design principles is modularity, achieved by organizing software in such a way that certain parts of the system can be created, used and changed independently of others. Parnas and others have discussed the importance of modularity in modern software design along with using the information hiding principle and prediction of future changes as the criteria to isolate design decisions that are likely to change independently [Parnas et al., 1985].

It is difficult to verify whether a software system has been successfully modularized. Ideally, in a modular system, every change is associated with a single module, whose design decision it hides. Identifying the percentage of changes that span more than one module can be used as a measure to indicate the ease of changeability of a system. For that, one must observe changes made to a system over a considerable amount of time during its development when most changes occur, measure the effort expended in making these changes, and identify changes that violate the system's

information hiding design principles. Furthermore, one can determine whether such changes introduce new bugs, and observe how they evolve over time in terms of effort and the number of lines of code touched. This can help software developers and architects perceive how the structure of a system changes over time with respect to its original architecture and hence aid them in designing software for change.

1.1 The idea behind chunks

Modelling approaches to design for change and identifying measures that determine the degree to which a system is amenable to change after it has been in use for a while are interrelated. Mockus and Weiss identified and explored the concept of chunks as a measure that can be used to predict how easily changes can be applied to a software system. We closely follow Mockus and Weiss in defining a chunk as "a set of code that has the property that a change that touches that set of code touches only that set of code". In other words, if a change touches one part of a chunk, then it is likely that it will touch other parts of the chunk as well. As an inference from the definition of chunks, a file that belongs to a chunk belongs to only that chunk, considering that changes modify files. One may think of chunks as the empirical information hiding modules of a system [Parnas, 1972].

Ideally, we would like to see perfect chunks that are completely independent of each other, in a way that every change touches only one chunk. However, in real world software systems, it is implausible to find perfect chunks for a variety of reasons and perturbing factors, including the following.

- Cross-cutting implementation concerns of a system that are inconsistent with the design and consequently cannot be cleanly separated from the rest of the system [Breu et al., 2006],
- Bad implementation practices that degrade the modular design of a system over time,
- Paucity of change in certain parts of a system, resulting in insufficient change data to identify chunks,
- Changes that touch a large part of a system, affecting many files and therefore many components. For example, a particular bug fix in the Moodle project touched 1272 files, and involved running lossless optimization on all .png (Portable Network Graphics) and .jpeg (Joint Photographic Experts Group) images that were distributed across various subsystems.

Identifying and analyzing chunks from change information of a particular software project over a significant fraction of its lifetime can help uncover parts of the system that are most and least changeable, and imply in which parts of the system its design needs to be changed.

1.2 Problem statement

The goal of this thesis is to investigate: 1) an approach to observe how modules in a software system change together by identifying tightly coupled changes known as chunks, 2) whether there are any trends in how chunks evolve over time, and 3) whether

chunks can help identify design issues or where a design decision needs to be changed in a software system.

A *module* is a work assignment rather than a sub-program [Parnas, 1972]. In the context of this thesis, we deviate from the term *module* as we do not have any information of the modular structure of the software systems studied. Instead, we use the term *component* to mean a set of code contained in files within any one directory in the master directory of repositories for the projects whose change data we analyzed. A change should be localized within a single component, since components should contain independently changeable design decisions. In this thesis we attempt to discover chunks in software systems by analyzing the changes that have been made to that system over a significant period of its lifecycle. There are several diverse attributes of change data pertaining to a software system, such as the type of change – bug fix or enhancement, the developer who made the change, time at which the change was made, the files modified by the change, the number of lines of code affected by the change, description of the change, or the subsystem(s) touched by the change. In this work, we use only a subset of the change data attributes required to perform chunk identification.

A family of algorithms is proposed to identify independently changing chunks from change data obtained from mining version history repositories of three large software systems – Moodle, Eclipse and Company-X (anonymized due to limited publication rights). A comprehensive analysis of certain characteristics of the resulting chunks is conducted. In addition, evolution of chunks over time with respect to size in terms of number of files within a chunk, and percentage of changes crossing a chunk is

studied. Lastly, a pragmatic interpretation of the results to identify necessary code refactoring or system redesign is presented.

The results from this work show that chunks merge over time touching many components, thereby making it increasingly difficult to make changes and to maintain the software system. In this thesis we also found that the percentage of changes crossing a chunk reduces with an increase in component and subsystem couplings. Increasing discrepancies between the developed software and its original design result in changes that require modifications to a large part of the system over longer time periods, therefore resulting in chunks with lower percentage of changes crossing the chunks.

Furthermore, in this work we identified chunks that are stable, i.e., with more than 65% of the files within the chunk existing commonly in different versions of its evolution over time. Stability is calculated as the percentage of files in the smaller version of the chunk that are in common with the larger chunk version over its evolution. For example, suppose a chunk C_1, containing 10 files evolved over time into chunk C_2 with 15 files, while having 9 files in common with C_1. In this case, stability of the chunk C_1 is 9/10, i.e., 90%. Note that a chunk does not necessarily increase in size over time. For example, suppose that during the first year of development, changes are made to two different parts of a component. These changes are then identified as a chunk. Over time, if the component is restructured, and changes to those two parts are made independently of each other. This results in a chunk that is smaller than the first, while still containing files from the common parts of the component. Clearly, a larger chunk has evolved into a smaller chunk in this scenario.

Observing stable chunks as empirical information hiding modules helps in a better understanding of the inconsistencies between how a system is designed for change and how it is actually changed. As such, chunks can be used as an indicator of when redesign of some or all of a software system is beneficial; identifying such design issues earlier in a project's lifecycle can prevent the company from losing valuable time in making changes to their system.

1.3 Structure of this thesis

The rest of this thesis is organized as follows. Chapter 2 discusses related work and how our approach differs from the others. Chapter 3 describes the data and approach. Chapter 4 presents the hypotheses, results and interpretation. Comparison of results from all three software projects — Moodle, Eclipse, and Company-X — are also included in this chapter. Chapter 5 discusses the challenges and difficulties faced. Chapter 6 describes the methods used to validate the data and the proposed family of algorithms. Finally, chapter 7 summarizes this research, and suggests future work in this area.

CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter discusses the terminology used in the rest of this thesis, background of this work along with related work, and describes how our approach differs from similar studies.

2.1 Chunking approach

The terminology used in the identification and analysis of chunks in this thesis and the background of this work is introduced in this section. In their introductory study on chunks [Mockus and Weiss, 2001], Mockus and Weiss define a *chunk* as "a set of code such that a set of work items all change that same set of code". A *work item* is defined as "the assignment of developers to a task, usually to make changes to the software". A work item can refer to any change request including an entire new software version, component, new functionality, modification request or even individual deltas within a modification request. A new functionality might require implementation of many components. A *modification request* (MR) is "a request to incorporate a specific change into software" [Herbsleb et al., 2001]. The analysis on chunks in this thesis is based on considering an MR as analogous to a commit in the change data. Every MR corresponds to a single type of change, such as a bug fix, new feature, enhancement, and fix on fix (new bugs resulting from fixing an existing bug). Other types of MRs might include auto installed batch files or header files, code clean up, etc.

We base our study on the work done by Mockus and Weiss, where the notion of identifying tightly coupled work items that can be developed independently in

distributed locations, thereby reducing communication and coordination needs within an organization is presented [Mockus and Weiss, 2001]. Files that are modified as part of an MR are tightly coupled as they all change together. As such, work items can be divided among developers by identifying sets of MRs that have strong inter-coupling between the MRs within that set, while sharing weak coupling with other MRs, thus paving the way for independent development or maintenance.

There are two kinds of coupling between any two entities as described below. They constitute the quantitative measures for dividing work items in distributed software development environments.

- *Absolute coupling:* The total number of MRs that modify or change both the entities. For example, if A and B are two components, then the total number of MRs that touch both A and B is referred to as absolute coupling between A and B.
- *Relative coupling:* The ratio of the total number of MRs that change both the entities to the total number of MRs that modify either of the entities. In the above example, relative coupling is the total number of MRs that touch both components A and B divided by the sum of MRs that touch component A and component B.

Suppose there are 20 MRs altogether that touch either component A or component B and 18 MRs that touch both component A and component B. Note that there are 2 MRs that touch either component A or component B but not both. In this case, the absolute

coupling between A and B is 18, while their relative coupling is 0.9. In the context of global location of developers, relative coupling is the ratio of multisite MRs to total MRs, and it has to be minimized to decrease coordination needs and increase the speed of producing new software. Hence, these measures can be used as criteria to optimize for the generation of chunk candidates.

In this work, we use the term *MRs crossing chunk* as analogous to absolute coupling, while *percentage correlation* is exactly the opposite of relative coupling. The number of MRs that touch files in both the chunk and rest of the system is referred to as *MRs crossing chunk*, whereas the number of MRs that touch only files within the chunk is termed as *MRs within chunk.* The ratio of MRs within chunk to the total number of MRs as a percentage is referred to as *percentage correlation* of the chunk. In the above example, suppose A and B represent two different chunks with 1 MR touching only A, 1 MR touching only B, and 18 MRs touching both A and B. In this case, each of the chunks, A and B will have a percentage correlation of 5.26%. Note that *percentage correlation* is not related to any of the statistical terms used for correlation (e.g. Spearman correlation).

2.1.1 Algorithm

A study of the literature indicates that Mockus and Weiss are the only ones who proposed an algorithm to identify chunks [Mockus and Weiss, 2001]. The algorithm generates candidates iteratively as described below and selects the best one based on the evaluation criterion chosen.

- The algorithm takes as input a set of files or modules, a set of MRs and the associated files that each MR modifies and a desired range of effort for the resulting candidate chunk.
- A module is randomly chosen as the initial candidate to be a chunk.
- A new candidate is generated by either adding a random module (chosen from the rest of the system) to the candidate, deleting a random module from the candidate or exchanging a module from the current candidate with one from the rest of the system.
- The algorithm accepts the new candidate with a probability $p > 1/3$ if the value of the selected evaluation criterion (coupling to the rest of the system) is improved.

The best possible candidate chunk with the highest value of the chosen evaluation criterion is generated, once the entire solution space is searched and explored.

2.2 Detection of logical couplings and its applications

Larry Constantine first defined the term coupling as the degree to which a module depends upon other modules [Constantine et al., 1979]. When different entities of a software system change together, as the system evolves, their common behavior is referred to as logical coupling [Gall et al., 2003]. It is a measure of the strength of dependency between the parts of a system that change together. Therefore, identifying logical couplings in a software system reveals the information hiding design structure of the entire system and exposes hidden inter-dependencies between files, modules or subsystems and other source code artifacts. In this section, we discuss several studies

aimed at identifying logical couplings (i.e., evolution of dependencies between system entities with the evolution of changes to a system over time).

2.2.1 Methods to identify logical couplings

Pearse and Oman discussed the use of code-based metrics like Lines of Code (LOC), and percentage of comment lines before and after a maintenance activity to identify the maintainability of a software system, and suggested the effects of code restructuring or addition of new features to existing code [Pearse and Oman, 1995]. Identification of syntactic and semantic dependencies between program entities was explored by Yang and Horwitz respectively [Yang, 1991], [Horwitz, 1990]. Neamtiu and others presented a tool to compare different source code versions to observe the evolution of a system by observing code-level dependencies [Neamtiu et al., 2005]. The basis of their approach is to find semantic differences between different program versions by using partial abstract syntax tree matching.

Analyzing structural dependencies on the source code level is quite challenging for large software systems involving millions of lines of code. Gall and others presented an approach to uncover logical coupling among modules by using version history data [Gall et al., 1998] to detect structural shortcomings, hence directing towards modules that require restructuring. The underlying concept is an empirical evaluation of the system's structure contrary to code-based metric approaches. In addition to identifying logical coupling between modules, such methods can be used to validate *code-level measures* that can be used only after the implementation is done and *predictive*

measures, which are derived from a system's design artifacts. Logical couplings are identified using a two-step process.

- Common change patterns for modules with respect to the system are identified over different versions of change history.
- These logical couplings are verified by observing the change reports for modules that have common change patterns. If the report identifies a common reason for change across different versions, then the logical coupling is confirmed.

The implementation of their approach used subsystems instead of modules, and calculated structural interdependencies on the subsystem level. This might hide potential dependencies at sub-modular or program level, which might lead to an undesirable increase in subsystem dependencies at a future time, thereby requiring much more cost and effort to reengineer the system. Arnold, in addition to Griswold and Notkin, in their works [Arnold, 1993], [Griswold and Notkin, 1993], investigated various software restructuring and reengineering methods.

The work that is most closely related to our work is that of Gall, Jazayeri, and Krajewski [Gall et al., 2003], in which they use version information to find dependencies between modules based on analyzing evolution of changes between classes and identifying common change patterns, also called logical couplings. Their work is based on an earlier study by Gall, Hajek and Jazayeri [Gall et al., 1998], which was discussed above. They described a method for software evolution analysis by a 3-step incremental approach. First, growth and change behavior of classes is assessed from the version

history information obtained from a CVS repository. Common change patterns across the system are then identified. Finally, classes that are changed commonly across different versions of the system are compared to observe evolution of different system components over time. They compared all changes to classes that were done on the same date and by the same author, as such changes indicate possible logical couplings between different classes and uncover potential hidden dependencies between modules or subsystems. They used the results to reveal architectural shortcomings of a software system by validating the findings with the development team of the company. A similar study was conducted by Bieman, Andrews, and Yang [Bieman et al., 2003] to identify coupling between classes using 39 releases of a commercial object-oriented software system.

Zimmermann, Diehl, and Zeller investigated an approach to identify such evolutionary couplings between functions, methods or attributes in a program by focusing on factual dependencies indicated by the revision history of the system [Zimmermann et al., 2003]. The emphasis is on the interlinking between entities within a program rather than higher level components such as modules or subsystems. A comparison of such evolutionary coupling against logical or analytical coupling, determined from evolutionary change analysis of programs, can unmask weaknesses in the system architecture.

A comparative study was conducted by Wong, Cai, Kim, and Dalton, in which an approach (CLIO) to detect modularity violations that can cause software defects or modularity decay is discussed [Wong et al., 2011]. Mismatches between how

components should change together based on a system's modular structure and how they actually change together as divulged by the version history are observed as modularity violations. The results were evaluated using the version histories of two large scale open source repositories – 10 releases of Eclipse JDT and 15 releases of Hadoop Common. Some of the detected violations were confirmed manually by in-depth analysis of the MRs concerned with a violation, while others are detected automatically by CLIO. Certain violations were identified a lot earlier in the system's lifecycle whereas the associated modules were refactored at a later time. The use of such tools can assist in identification of poor design in the beginning stages of a project's development. However, there was no mention of any design artifacts that were used to validate CLIO's findings.

2.2.2 Mining logical couplings for predictive analyses

Ying, Murphy, Ng, and Chu-Carroll investigated an approach to recommend relevant files that can be possibly changed to a developer while performing a modification task by using associative rule mining on change data obtained from two large open source repositories – Mozilla and Eclipse [Ying et al., 2004]. As a developer starts to make a change to a file, a set of additional files that most likely change together with the file being changed are suggested. These recommendations are derived from a frequent pattern mining algorithm based on frequency counts of the instances of files that change together. In other words, logical couplings between files are first identified, and then the mining rules for predicting changes are derived. The results were evaluated by classifying a recommendation as most useful or surprising, if it could not be

determined by analytical analysis of the program. The limitation of this approach is that it does not support mining on the fly.

Another similar approach was presented by Zimmermann, Diehl, and Zeller [Zimmermann et al., 2005] that uses association rule mining of multi-dimensional version information to predict further probable changes and supports mining on the fly. They proposed a tool (ROSE) to detect coupling between program entities such as functions or variables, generate multi-dimensional association rules by analyzing different types of changes, and suggest future changes by investigating how changes evolve over the project's lifetime.

2.2.3 Relationship between logical coupling and software defects

Another work in recent times that is closely related to this thesis is the one conducted by Steff and Russo, in which they construct graphs of ordered commits to identify defective modules and other software defects in the system by observing co-evolution of commits and files [Steff and Russo, 2012]. Defects with the status "fixed" were selected from the SVN repository of the Spring project and mapped with the associated commits containing the corresponding defect ID to obtain information about a particular change. A list of all files in each commit, and for each file information about the commit in which it was last changed are obtained. A commit graph is constructed with each commit representing a node and an edge existing between two nodes if both corresponding commits commonly changed one or more files, without any other commit changing any of these files in between.

A graph with one large connected component was obtained with an average node degree of 2, indicating a sparsely connected commit structure, attributed to object-oriented modularity design principles of the system being studied. Also, there were two types of special nodes in the commit graph – root and end nodes. A root node has no in-degree, indicating that none of its files has been changed before. An end node has no out degree, meaning none of its files has been changed later on, i.e., it is a one-time change such as a header import. The components of the graph that share a high number of edges between them correspond to possible logical couplings in the system and are identical to the chunks observed in this thesis.

Correlations between the history of each commit and defects in that history were calculated for all files concerned, from which it was observed that both number of commits and number of files associated with a commit's history have a high correlation with the number of defects. It was also found that bug-fixing commits share a higher number of edges or files between them and are well connected and distributed across the entire commit graph. Hence, it can be inferred that nodes with a higher degree are more likely bound to be defects. In other words, higher order logical coupling can be used to detect risky or defective code structures in files. A limitation of using this approach is the underlying complexity in constructing a commit graph for large projects containing hundreds of thousands of commits.

D'Ambros, Lanza, and Robbes analyzed the relationship between change coupling and software defects, and to statistically investigate if there is a correlation between change coupling and defects. Change couplings are correlated with defects

more than object-oriented or other complexity metrics but less than the number of changes [D'Ambros et al., 2009]. It was also observed that defects with higher severity like bug fixes exhibit a higher correlation with change coupling.

In another work conducted by Graves, Karr, Marron, and Siy [Graves et al., 2000], the extent to which code and its change history are successful in predicting the distribution of faults that arise in modules. In this context, a module is considered as a set of related files. Several statistical models were developed to see which attributes of the change history were likely to indicate the generation of a large number of faults as the module continued to be developed. Fault potential was predicted by using a sum of contributions from all changes to the module in its change history, and it was observed that old changes were weighed down by a factor of around 50% per year, i.e., changes made a year ago were only about half as influential in fault prediction as changes made yesterday. A measure of the module's age gave satisfactory prediction results, while characteristics like length of the module in terms of number of lines of code, and number of developers who had made changes to the module did not provide expected results. Interestingly, their attempt could not predict faults by using information of coupling between modules, which is a potential attribute of change history to measure defective code or system architecture.

To our knowledge, the work presented in this thesis is the first one to identify chunks as well as analyze chunk evolution as an empirical approach to reveal hidden relations between files, components or modules that assist in exposing potential shortcomings in the system's architecture or design, implementation, and

maintainability. We propose a family of algorithms to be used for chunk identification. The approach used in this work differs from other related work in that we not only identify logical couplings between files but also investigate the evolution of such couplings by using attributes of chunks rather than measurement metrics of code or modules, such as number of lines of code, length of the module, etc. As such, our approach does not require information about the modular structure of a system or other design artifacts as in other works; however, such knowledge about a system's architecture can be valuable in complementing our findings in this study.

CHAPTER 3. APPROACH

This chapter describes the approach used for identifying and analyzing chunks and their evolution by using change history information. A description of how we collected data from the version control repository of Moodle is presented, along with references to the data sources of Eclipse and Company-X projects. In addition, the characteristics of chunks that we considered for analyzing chunk evolution are discussed in subsequent sections.

3.1 Data collection

Most open source projects maintain and store data concerning changes made to the system in version control repositories as commits, containing information such as the author who had made the change, time at which the change was made, description of the change, etc. as indicated in Table 3.1. These changes are linked to a bug tracker that contains knowledge about the type of change, patch files if the change is a bug fix, or files that were modified, including the number of lines of code that is changed. The algorithm proposed and used in this thesis requires a data source to encompass a set of MRs – each MR is considered as a change, the number of files modified by each MR together with the file names, time of change, person who made the change, and a classification of MRs into bug fixes, enhancements, new features, or automated changes. After a careful consideration of change histories of about 15 projects, we finally selected the Moodle course management system, and the Eclipse integrated development environment because of the availability of appropriate change data and the ease with

which this data can be collected. Data from Company-X was generously provided to us by a colleague and contained all the required information for chunk identification.

Moodle stores all changes made to the source code in their Git repository [Moodle Git], which is a distributed version control and source code management system with an emphasis on speed of data storage and access. The Git working directory of any project is a repository on its own with complete history and tracking capabilities. Commits in the Git repository are linked to Moodle's JIRA issue tracking system. We first cloned their Git repository to obtain a list of MRs each with a commit ID, author of commit, time of commit in the form of UNIX timestamp, names of files touched by the MR, description of the MR, and a tracker reference number that allows mapping with the issue tracker. We used two simple Python scrapers, one to pull this data into a SQLite database on our server, and the other to pull the issue type of each MR from the issue tracker into a SQLite database. The issue types of MRs are organized as either a *bug, improvement, sub-task, new feature*. For MRs without a tracker reference number (16 MRs accounting for less than 1% of the total MRs), we use pattern matching to search the commit description for words like "bug" or "fix" in order to identify bug fixes. The Moodle data that we used in this work spans across 01-Jan-2008 to 31-Dec-2011.

Data for the Eclipse project was provided to us by Krishnan, who used Eclipse's change data to perform defect assessment and explored defect prediction of failure-prone files in the Eclipse product line in his work [Krishnan, 2013]. The data was extracted from the Eclipse Bugzilla database and CVS change repositories for the Eclipse Classic product. The CVS log data was mined for six-digit strings that could be matched to bug

IDs. A manual review was performed to ensure that this pattern matching caught all the log data entries containing the word “bug”. He also used the CVSps (Patchsets for CVS) tool that identifies files committed together as a changeset. This additional processing had to be done as changes are stored in a CVS repository in terms of each file that is changed contrary to other repositories such as Git or SVN (Subversion) where changes correspond to commits involving a set of files modified by the change. Custom PERL scripts were written to parse the CVS log entries into a SQL database. In addition, developers at Eclipse provided a bug database containing change history information required for this study along with an affluence of other details of bug fixes for selected Eclipse releases in the form of a SQL database. The dataset contributed to us constituted Eclipse’s Europa (release 3.3) project’s bug data from 02-July-2006 to 31-Dec-2007. We parsed the necessary information suitable for the algorithm into another SQLite database using a Python script.

We used a similar Python script as mentioned earlier to parse and extract the required information from CVS logs of change data from Company-X into a SQLite database. The dataset is a wealth of information consisting of the entire change history data collected over the time period 28-July-1993 to 04-June-2009. In this case, the MRs are classified by their type as either *enhancement, initialization, modification, new feature, problem,* or *sw_offer_build*.

A set of all the change data characteristics used in this work and are essential to identify chunks for each of the three projects – Moodle, Eclipse, and Company-X is listed in Table 3.1.

Table 3.1: **List of Change Data Characteristics Used**

Characteristic	Description	Project	Data Type/Value
CS_ID	The changeset (commit) ID, unique for every commit.	Moodle	A sequence of 40 alphanumeric characters.
		Eclipse	A numeric sequence of variable length.
		Company-X	A numeric sequence of variable length.
AUTHOR	The name or email ID of the developer assigned to make the change.	Moodle	Full name or email ID.
		Eclipse	Email ID.
		Company-X	Encoded numeric value.
DATE	The time at which the change is made (committed).	Moodle	Unix timestamp.
		Eclipse	Full date format: yyyy-mm-dd hh:mm:ss.
		Company-X	Unix timestamp.
ISSUE_TYPE	The type of MR or change.	Moodle	bug, improvement, new feature, sub-task, and task
		Eclipse	bug, and enhancement
		Company-X	enhancement, initialization, modification, new feature, problem, and sw_offer_build
MESSAGE	A short description of the change.	Moodle Eclipse Company-X	Textual description of the committed change.
FILE_NAME	Names of all the files modified by the change; FILE_COUNT is consequently calculated as the number of files	Moodle	File names in the form of file paths
		Eclipse	File names in the form of file paths.
		Company-X	Encoded in numeric file path

Table 3.1: **(Continued)**

Characteristic	Description	Project	Data Type/Value
	touched by the change		format. E.g. 1/2/3/75/1049

3.2 Algorithm(s) to identify chunks

We use a family of algorithms to identify chunks, a list of whose commonalities and variabilities is described in detail in this section. *Commonalities* are the characteristics shared by all the algorithms, whereas *variabilities* describe how the algorithms differ from each other [Weiss and Lai, 1999].

3.2.1 Commonalities

The following functionality characteristics are common to all the algorithms.

- All algorithms analyze changes that have been made to a system over some significant fraction of its lifetime, usually on the order of 18 months to 5 years, or more if the data is available. Each change is reported and characterized as an MR.
- All algorithms operate on change history data obtained from software version control repositories linked to an issue tracker.
- All algorithms start by randomly identifying a chunk candidate.
- All algorithms use a set of optimization criteria to decide whether a chunk candidate is, in fact, a chunk. Included are criteria such as the number of

changes that touch both the candidate and its complement, and the percent of changes that touch only the candidate.

- All algorithms generate chunks by adding files to the set of files of an existing candidate and checking to see if the resulting new chunk improves the optimization criteria.
- All algorithms terminate after a specified time interval. This time limit is an estimate of the maximum amount of time after which there is no improvement in the optimization criteria. It is measured over many trials of the algorithm and is dependent on the number of MRs and files involved in the dataset, the system speed, and the selected optimization criteria.

3.2.2 Variabilities

Each variability in the algorithm family is a result of a careful observation of the chunks generated from the existing algorithms, identifying errors with both the approach and data, and re-evaluating constraints necessary for a meaningful analysis of chunks. All algorithms differ by the following characteristics.

- All algorithms vary by how they identify the initial chunk candidate, either by picking a random subset of files, a random MR, or a random subset of MRs.
- All algorithms vary by the optimization criteria used. Either or both of the following optimization criteria whose threshold levels act as constraints to the algorithm can be used to generate chunk candidates.

1. *Percentage correlation* – this is a measure of how tightly coupled the files within a chunk are, and as such it should be maximized.
2. *MRs crossing chunk* – this is a measure of the number of MRs that touch both the chunk and its complement, and should therefore be minimized.

- All algorithms vary by constraints imposed on the optimization criteria.
- All algorithms vary by constraints imposed on data, such as the minimum number of files desired in a chunk, the type of changes to include in the analysis, the minimum number of files that an MR should touch to be randomly picked by the algorithm, etc.

3.2.3 Description of the proposed algorithm

A brief description of the steps involved in the algorithm used in this work for identifying independent chunks is as follows.

1. Randomly pick an MR that touches at least five files from the entire set of MRs being considered for chunk identification, obtained from the corresponding project's change data. The randomization process is dependent on the programming language used. Python uses a uniform random number generator. This guarantees a minimum chunk size that is large enough to eliminate one-time changes, i.e., MRs, such that the files modified by these MRs are not touched by any other MR. Such MRs are of no interest to us

since our goal is to study the evolution of frequently occurring changes or logical couplings.

2. The set of files touched by this MR forms the initial chunk, which is also the current candidate chunk.
3. Find a set of all MRs that touch each of the files in the current candidate chunk.
4. Find a set of all files touched by the MRs in step 3.
5. Add a file from the set of files in step 4 to the initial chunk only if it improves the optimization criteria; this forms the current candidate chunk.
6. Repeat step 5 until all the files in the set of files in step 4 are considered.

   The above steps ensure that the search space for all file combinations is sequentially explored so that potential candidates are not omitted. This also leaves out false chunks resulting from combining isolated one-time changes.
7. Repeat steps 3 to 6 as long as adding a file increases the optimization criteria.
8. Repeat steps 1 through 7 if adding a file does not improve the optimization criteria until all MRs in step 3 are considered.
9. Repeat steps 1 to 8 for a specified time interval, after which the best chunk candidate based on the best optimization criteria is generated.
10. Remove the files constituting the chunk as well as all MRs within the chunk generated in step 9 from the initial set of files and MRs considered respectively, and repeat steps 1 through 9 to generate the next independent chunk.

The algorithm iterates through the above steps for 8000, 4000, and 10000 seconds respectively for Moodle, Eclipse, and Company-X change data, after which it generates the best candidate chunk. The algorithm was run on a Dell Optiplex 980 with Intel Core i7 processor.

3.3 Attributes of chunks used for analysis

The structure of change data that we collected enabled us to determine analysis of chunks by size, percentage correlation, number of undesired MRs referred by MRs crossing chunk, and time period of changes. As mentioned in Section 3.2.2, *percentage correlation* is a measure of coupling between files within a chunk. Let **F** be the entire set of files touched by all MRs in the dataset used as input for the algorithm described in Section 3.2.3 to identify chunks. Let **C** be a chunk, and **C'** be its complement, i.e., **C'** contains all files in **F** that are not in **C**. Since perfect chunks are usually uncommon due to undesired couplings between components or cross-cutting concerns, there are a certain number of MRs in **C** that also touch files in **C'**. We refer to the total number of such MRs as *MRs crossing chunk*. Correspondingly, the number of changes in **C** that touch only files in **C** is referred to as *MRs within chunk*. Hence, *percentage correlation* is defined as the percent of total MRs in **C** that touch only files within **C**. In other words, percentage correlation is a measure of how close a chunk is to being a perfect chunk, which has a percentage correlation value of 100%. Therefore, a higher percentage correlation indicates a more cohesive chunk.

$$percentage\ correlation = \frac{MRs\ within\ chunk}{MRs\ within\ chunk + MRs\ crossing\ chunk} * 100\% \quad (3.1)$$

The number of undesired MRs within a chunk is equivalent to the number of MRs crossing the chunk. As mentioned above, these correspond to changes that touch both **C** and **C'**. In some cases, taking only *percentage correlation* into account for analysis of chunks can lead to false or misleading interpretations. For example, assume that there two chunks $\mathbf{C_1}$ and $\mathbf{C_2}$, with two sets of 20 MRs one touching each chunk, in a way that the two sets of MRs are completely disjoint. That is to say, there are no MRs in either set that touch both $\mathbf{C_1}$ and $\mathbf{C_2}$. Assume that there are 5 MRs crossing $\mathbf{C_1}$ and 3 MRs crossing $\mathbf{C_2}$. In terms of percentage correlation, $\mathbf{C_2}$ with 85% correlation appears to be a significantly better chunk than $\mathbf{C_1}$ with only 75% correlation. Alternatively, if we look at the number of MRs crossing both chunks, the difference is not too high (only 2), i.e., $\mathbf{C_1}$ is almost as good a chunk as $\mathbf{C_2}$ since the metric *MRs crossing chunk* is low for both $\mathbf{C_1}$ and $\mathbf{C_2}$. If we set the threshold for percentage correlation as 80%, the algorithm will only pick $\mathbf{C_2}$, eliminating $\mathbf{C_1}$ although it is a potential candidate chunk. Such discrepancies arise especially when there are very few changes touching files in a chunk, in which case the percentage correlation decreases drastically even with a slight increase in MRs crossing chunk. In order to account for these disparities, we used both *percentage correlation* and *MRs crossing chunk* as the criteria to be optimized for the algorithm to generate potential independent chunks.

We also consider different time periods of changes to observe how chunks evolve over time by looking at chunks that have at least 65% files in common between

any two chunks in each such time interval. This might help identify whether chunks are stable over time, and if so, whether they correspond to any component or subsystem.

The *size* of a chunk is defined as the number of files within that chunk. Observing trends of a chunk's size as it evolves over time can guide towards identifying plausible design issues of the system being studied. A consistent increase in chunk size might suggest necessary refactoring of code or design altogether. An extremely large chunk size indicates that a change to one file within the chunk requires changes to a large number of other files within that chunk, thus making the system extremely difficult to maintain. We also look at the association between percentage correlation and size of chunks over time to see how the size of stable chunks varies over time.

CHAPTER 4. RESULTS AND ANALYSIS

In this chapter, the results generated by the proposed algorithm are discussed. A detailed analysis of the identified chunks for Moodle, Eclipse, and Company-X data is also presented in the following sections.

We included MRs that touch 20 files or less for reasons explained in Chapter 5 and the presented discussion in Chapter 7. In addition, we only considered MRs that are bug fixes because they are more likely to be localized, touching only the component in which the bug arises. Other types of changes such as enhancements or new features can touch a number of different components. For example, a new feature might require new functionality to be added in multiple existing components, thereby resulting in chunks that span across many components. However, such changes to separate components can be made independently. In such cases, it is impractical to detect software system design issues by chunk analysis. On the other hand, it is not possible to make bug-fixing changes independently to multiple components. Identifying bug fixing chunks that touch more than one component usually imply limitations in the information hiding design structure of a system.

4.1 Eclipse results

We identified chunks for CVS log data of Eclipse's Europa release over the time period from 02-July-2006 to 31-Dec-2007, i.e., over an interval of 18 months. We represent this dataset as *Europa*. To study the evolution of chunks over time, we divided this data into two sets of distinct datasets, each containing change data over 9 months,

i.e., from 02-July-2006 until 31-Mar-2007, and from 01-Apr-2007 to 31-Dec-2007, represented as *Europa-A* and *Europa-B* respectively. The number of bug fixing MRs concerning each of the datasets is shown in Table 4.1. We removed duplicate MRs as discussed in Chapter 6. The change in the number of non-duplicate MRs that are bug fixes on a quarterly basis are shown in Figure 4.1. Europa's main release was during June-2007, indicated by a rise in the plot.

Table 4.1: **Number of bug fixing MRs in each Europa dataset**

Europa	Europa-A	Europa-B
11042	5144	5898

After running many trials, we enforced a threshold of 80% as the minimum value for percentage correlation of candidate chunks generated by the algorithm. In other words, the identified chunks will have at least 80% MRs within each chunk. With the specified threshold level, there were enough number of chunks generated for analysis. The algorithm identified 20 chunks for *Europa*, 11 for *Europa-A*, and 16 for *Europa-B*. The values for percentage correlation, size in terms of the number of files contained within the chunk, MRs within chunk, and MRs crossing chunk for all the datasets are listed in Table 4.2, Table 4.3, and Table 4.4.

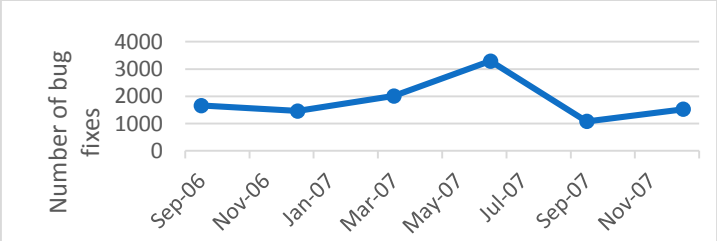

Figure 4.1: **Number of Bug Fixes in Eclipse**

Table 4.2: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Europa**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1E	100	40	13	0
2E	100	9	11	0
3E	100	26	10	0
4E	100	15	19	0
5E	94.74	28	18	1
6E	94.64	14	53	3
7E	90.91	18	10	1
8E	90.91	11	10	1
9E	90.48	11	19	2
10E	90	18	18	2
11E	87.8	33	72	10
12E	86.96	30	20	3
13E	86.67	18	13	2
14E	86.67	26	39	6
15E	83.33	14	10	2
16E	83.33	17	10	2
17E	82.86	26	29	6
18E	81.61	62	213	48
19E	81.48	27	22	5
20E	80.3	29	53	13

Table 4.3: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Europa-A**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1E-A	94.74	28	18	1
2E-A	92.86	8	13	1
3E-A	90.91	32	40	4
4E-A	89.47	31	17	2
5E-A	88.24	14	15	2
6E-A	88.24	55	15	2
7E-A	85.19	33	46	8
8E-A	85.11	64	88	6
9E-A	82.35	44	70	15
10E-A	82.35	10	14	3
11E-A	81.25	15	13	3

Table 4.4: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Europa-B**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1E-B	100	15	18	0
2E-B	100	40	13	0
3E-B	100	25	15	0
4E-B	100	13	10	0
5E-B	96.77	29	30	1
6E-B	93.1	27	27	2
7E-B	91.67	19	11	1
8E-B	90.91	27	10	1
9E-B	90.91	13	10	1
10E-B	90.91	15	10	1
11E-B	88.24	19	30	4
12E-B	87.04	39	47	7
13E-B	86.79	43	46	7
14E-B	84	26	21	4
15E-B	83.53	64	142	28
16E-B	80	20	12	3

We observed the file structure within each of these chunks to detect potential stable chunks that might represent empirical information hiding structures, analyze how chunks evolve over time with respect to their size, and percentage correlation, and identify chunks that merge into a single large chunk, or chunks that split into multiple smaller chunks over time. The percentage correlations of the top 11 chunks for each of the three datasets is shown in Figure 4.2. We chose only the top 11 chunks to be displayed since *Europa-B* has only 11 chunks, and to make comparison between the datasets more meaningful. The percentage correlations of all identified chunks along with their respective sizes are shown separately for all three datasets in Figure 4.3, Figure 4.4, and Figure 4.5.

The following observations were made from the chunks generated from *Europa*, *Europa-A*, and *Europa-B*.

- The file structure within the identified chunks is highly compact. We mapped files within each chunk to their respective directories. According to developers on Eclipse forums, every such directory in the CVS repository under HEAD, which is of the form /cvsroot/platform/modules usually represents a component, and /cvsroot/platform/ represents a subsystem. Most of the chunks contained files that were mapped to a single destination directory, indicating a one-to-one mapping between chunks and directories.

  Inferences: We attribute this difference to the primary programming language used by Eclipse - Java, which is an object-oriented language. Hence, chunks from Eclipse seem more modular.

  Detailed Analysis: All chunks from the three datasets that map to more than one directory or component are listed in Table 4.5, along with the respective chunk identifiers and, path structure of the all the components touched by each of those chunks. For example, chunk 20E is mapped to three components *org.eclipse.ui.workbench*, *org.eclipse.jdt.ui*, and *org.eclipse.ui.ide*. Although they are all related to the UI (User Interface), they are originally designed as separate components.

  Chunk 7E-A spans across 5 components and 2 subsystems – *eclipse* and *tools*. Such higher level couplings between subsystems might indicate the need for code refactoring or structural redesign concerning those specific subsystems

and their components. Architects or developers can be further consulted to suggest or verify such claims.

Table 4.5: **Chunk to component mappings for Europa, Europa - A, and Europa-B**

Dataset	Chunk ID	Component Mapping Paths
Europa	1E	/cvsroot/webtools/org.eclipse.jsdt/
		/cvsroot/webtools/sourceediting/
	2E	/cvsroot/eclipse/org.eclipse.pde.runtime/
		/cvsroot/eclipse/org.eclipse.ui.views.log/
	3E	/cvsroot/webtools/servertools/
	4E	/cvsroot/tools/org.eclipse.cdt/
	5E	/cvsroot/eclipse/org.eclipse.ui.workbench/
	6E	/cvsroot/modeling/org.eclipse.emf/
	7E	/cvsroot/eclipse/org.eclipse.pde/
	8E	/cvsroot/eclipse/org.eclipse.update.ui/
	9E	/cvsroot/eclipse/org.eclipse.jdt.ui/
	10E	/cvsroot/modeling/org.eclipse.emf/
	11E	/cvsroot/eclipse/org.eclipse.swt/
		/cvsroot/eclipse/org.eclipse.jface.text/
	12E	/cvsroot/eclipse/org.eclipse.ui.intro/
	13E	/cvsroot/tools/org.eclipse.cdt/
	14E	/cvsroot/eclipse/org.eclipse.ui.workbench/
	15E	/cvsroot/modeling/org.eclipse.emf/
	16E	/cvsroot/webtools/jeetools/
	17E	/cvsroot/webtools/webservices/
	18E	/cvsroot/eclipse/org.eclipse.swt/
	19E	/cvsroot/eclipse/org.eclipse.ui.workbench/
	20E	/cvsroot/eclipse/org.eclipse.ui.workbench/
		/cvsroot/eclipse/org.eclipse.jdt.ui/
		/cvsroot/eclipse/org.eclipse.ui.ide/
Europa-A	1E-A	/cvsroot/eclipse/org.eclipse.ui.workbench/
	2E-A	/cvsroot/modeling/org.eclipse.emf/
	3E-A	/cvsroot/eclipse/org.eclipse.jdt.ui/
	4E-A	/cvsroot/modeling/org.eclipse.emf/
	5E-A	/cvsroot/eclipse/org.eclipse.jdt.ui/
	6E-A	/cvsroot/modeling/org.eclipse.emf/
	7E-A	/cvsroot/eclipse/org.eclipse.jface.text/
		/cvsroot/eclipse/org.eclipse.ui.workbench/
		/cvsroot/eclipse/org.eclipse.jdt.ui/
		/cvsroot/tools/org.eclipse.cdt/
		/cvsroot/eclipse/org.eclipse.text/
	8E-A	/cvsroot/eclipse/org.eclipse.jdt.ui/
	9E-A	/cvsroot/eclipse/org.eclipse.swt/
		/cvsroot/eclipse/org.eclipse.pde.core/
		/cvsroot/eclipse/org.eclipse.core.resources/

Table 4.5: **(Continued)**

Dataset	Chunk ID	Component Mapping Paths
	10E-A	/cvsroot/webtools/jeetools/
		/cvsroot/webtools/common/
	11E-A	/cvsroot/webtools/sourceediting/
Europa-B	1E-B	/cvsroot/tools/org.eclipse.cdt/
	2E-B	/cvsroot/webtools/org.eclipse.jsdt/
		/cvsroot/webtools/sourceediting/
	3E-B	/cvsroot/eclipse/org.eclipse.ui.intro/
	4E-B	/cvsroot/eclipse/org.eclipse.ui.workbench/
	5E-B	/cvsroot/eclipse/org.eclipse.debug.ui/
	6E-B	/cvsroot/eclipse/org.eclipse.ui.workbench/
	7E-B	/cvsroot/eclipse/org.eclipse.jdt.junit/
	8E-B	/cvsroot/eclipse/org.eclipse.ui.ide/
		/cvsroot/eclipse/org.eclipse.ltk.ui.refactoring/
	9E-B	/cvsroot/tools/org.eclipse.cdt/
	10E-B	/cvsroot/tools/org.eclipse.cdt/
	11E-B	/cvsroot/eclipse/org.eclipse.ui.workbench/
	12E-B	/cvsroot/eclipse/org.eclipse.jdt.ui/
	13E-B	/cvsroot/eclipse/org.eclipse.ui.workbench/
		/cvsroot/eclipse/org.eclipse.ui.ide/
	14E-B	/cvsroot/modeling/org.eclipse.emf/
	15E-B	/cvsroot/eclipse/org.eclipse.swt/
		/cvsroot/eclipse/org.eclipse.jdt.apt.core/
	16E-B	/cvsroot/eclipse/org.eclipse.compare/

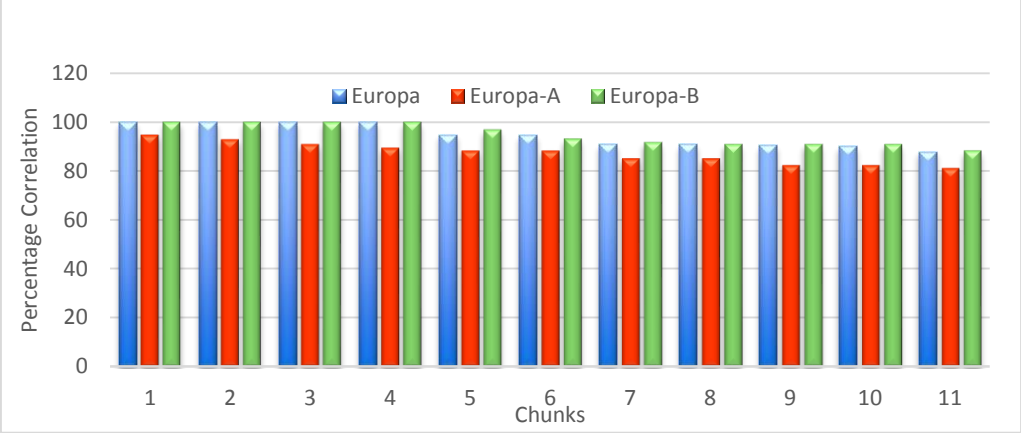

Figure 4.2: **Percentage Correlation of Top-11 Chunks for All Eclipse Datasets**

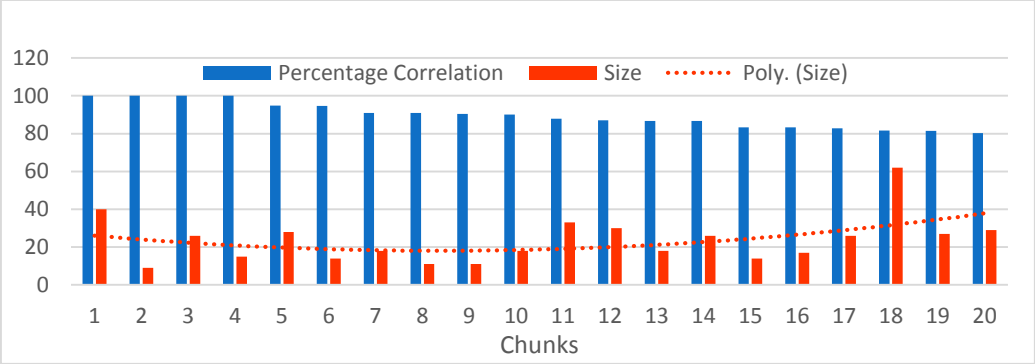

Figure 4.3: **Percentage Correlation and Size of Chunks for Europa**

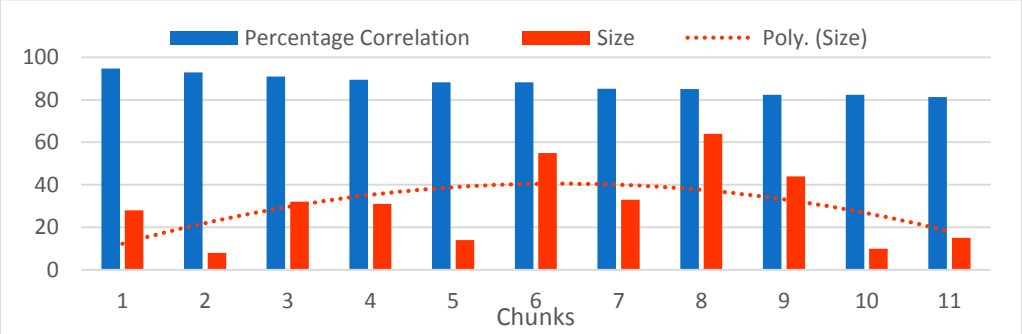

Figure 4.4: **Percentage Correlation and Size of Chunks for Europa-A**

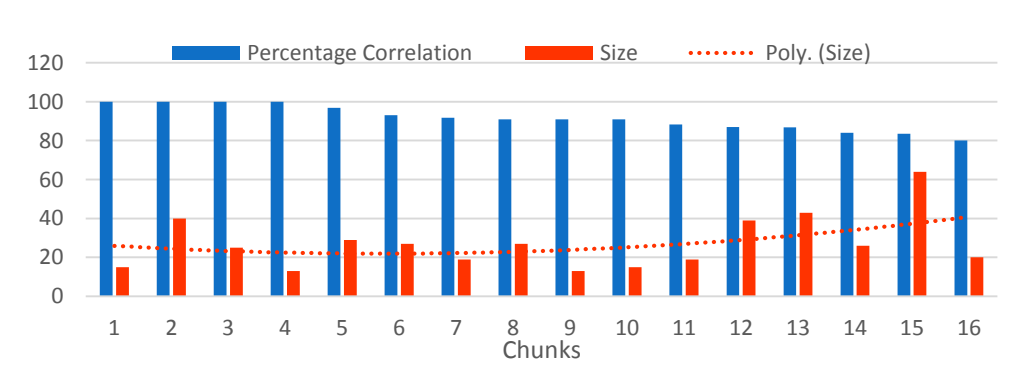

Figure 4.5: **Percentage Correlation and Size of Chunks for Europa-B**

- The dataset *Europa* generated the highest number of chunks (20), followed by *Europa-B* (16), and *Europa-A* (11) with percentage correlation above 80%. As seen from Figure 4.2, the percentage correlation of chunks from *Europa-A* is

lower than that of chunks from *Europa* and *Europa-B*. Chunks from *Europa-B* appear to be relatively better than chunks from the other two datasets with respect to percentage correlation, meaning that these chunks have tight internal coupling between their files relative to the rest.

Inferences: 1) Since the number of bug fixing MRs is higher in *Europa*, followed by *Europa-B* and lastly *Europa-A*, we hypothesize that the number of chunks generated from each of these datasets follows the same order. This is because more bug fixes might touch more parts of a system, thus resulting in more chunks. However, Hamill and Goseva-Popstojanova [Hamill and Popstojanova, 2009] noted that although non-localized faults are related to individual bugs, they are mostly contained in a small part of the system.

2) Subsystem coupling identified in chunk 7E-A might be one reason why the percentage correlation of chunks from *Europa-A* is lower.

3) *Europa-A* has the highest number of inter-component couplings when compared to *Europa* and *Europa-B*, where 11 chunks are mapped to 18 components, resulting in an average of 1.63 components touched by every chunk, followed by *Europa* and *Europa-B*, both touching 1.25 components per chunk. The effort to make changes increases with an increase in inter-component or subsystem couplings, thus reducing the system's maintainability resulting in chunks that touch multiple areas of code. This explains the lower percentage correlation of chunks from *Europa-A*.

4) Although the average number of components touched per chunk is the same for both *Europa* and *Europa-B*, higher number of MRs in *Europa* result in modifications to relatively more parts of the system, and therefore resulting in a higher proportion of MRs crossing respective chunks. Hence, chunks from *Europa-B* are more cohesive than those from *Europa*.

- There is no consistent increasing or decreasing trend in size of chunks with respect to percentage correlation as shown by the trend lines in Figure 4.3, Figure 4.4, and Figure 4.5.

  Inferences: There is no association between size and percentage correlation of chunks. Size and percentage correlation depend solely on the MRs touching a chunk, and the coupling between files touched by these MRs. If a chunk contains MRs that touch a wider area of code, then the size of that chunk will be large. A tighter coupling between files touched by MRs within a chunk indicates a higher percentage correlation.

- We observed two stable chunks. Recall that chunks are considered stable over time if there are at least 65% of the files within the smaller chunk in common with the larger chunk. Chunks 1E-A, and 2E-A evolved into chunks 5E, and 6E, both having a stability of 100%. Chunks 1E-A and 5E are exactly the same. However, both size and percentage correlation of 6E increased, making it a better chunk (only if effort remains constant) after its evolution from 2E-A.

  Inferences: It appears that both size and percentage correlation of a stable chunk increases as it evolves over time. We do not have sufficient information to

confirm this conclusion since we analyzed the evolution of only one chunk (2E-A).

- Chunks 8E-A and 9E-A appear to have merged into chunk 18E. In other words, chunk 18E contains files that are common to both 8E-A and 9E-A, i.e., changes that could be made independently in *Europa-A* merged together in a way that changes to 8E-A requires making changes to 9E-A as well, as indicated by the merged chunk 18E from *Europa.* There are 20 files common between 8E-A and 18E, while 9E-A and 18E share 17 files in common. This merging is clearly visible when we look at the corresponding chunk 15E-B from *Europa-B*, indicating that the merging occurred during the time period of *Europa-B* data. Since the algorithm picks the highest correlation chunk, 18E has fewer files than 8E-A, and 9E-A combined although it is a combination of MRs involved in 8E-A, 9E-A and 15E-B.

  Inferences: Over time, coupling between files increases, thereby resulting in an increase in difficulty to independent changes to different parts of a system. Identification of such merging between chunks over longer time periods can determine parts of a system that require redesign or restructuring.

4.2 Moodle results

Change history data during the time period 01-Jan-2008 to 31-Dec-2011, i.e., over a period of 4 years was collected for this analysis. This dataset is represented as *Moodle.* For the sake of analyzing chunk evolution over time, we divided *Moodle* into

two separate datasets, over the time periods 01-Jan-2008 to 31-Dec-2009, and 01-Jan-2010 until 31-Dec-2011, represented as *Moodle-A* and *Moodle-B* respectively. The number of bug fixing MRs in each of the datasets is shown in Table 4.6. Duplicate MRs were discarded as with Eclipse and discussed in Chapter 6. The change in the number of non-duplicate MRs that are bug fixes on a half-yearly basis are shown in Figure 4.6. The rise in the plot during Dec-2010 marks the release of Moodle 2.0, which was a major release.

Table 4.6: **Number of Bug-Fixing MRs for All Moodle Datasets**

Moodle	Moodle -A	Moodle -B
6016	2444	3572

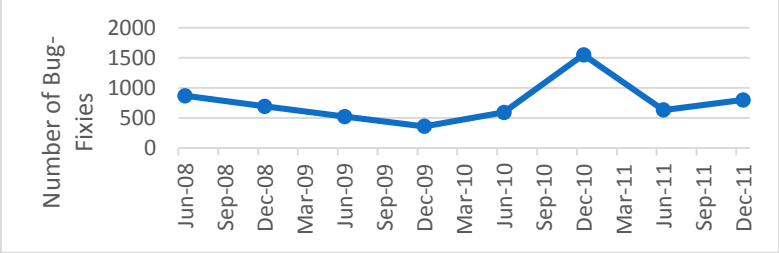

Figure 4.6: **Number of Bug Fixes in Moodle**

We enforced a threshold of 80% as the minimum value for percentage correlation of chunks generated by the algorithm, i.e., each of the identified chunks will have at least 80% MRs within chunk, same as Eclipse. The algorithm identified 10 chunks for *Moodle*, 7 for *Moodle-A*, and 10 for *Moodle-B*. The values for percentage correlation, size in terms of the number of files contained within the chunk, MRs within chunk, and

MRs crossing chunk for all three datasets are listed in Table 4.7, Table 4.8, and Table 4.9.

Table 4.7: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Moodle**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1M	93.53	26	130	9
2M	88.24	36	30	4
3M	88.24	10	15	2
4M	85.06	37	148	26
5M	84.51	13	60	11
6M	81.68	25	107	24
7M	81.3	38	100	23
8M	81.3	48	187	43
9M	80.77	10	21	5
10M	80	16	64	16

Table 4.8: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Moodle-A**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1M-A	87.5	29	14	2
2M-A	87.18	27	102	15
3M-A	84.45	10	28	5
4M-A	84	15	21	4
5M-A	83.56	25	61	12
6M-A	81.44	25	136	31

Table 4.9: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Moodle-B**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1M-B	93.41	12	85	6
2M-B	89.47	19	17	2
3M-B	86.67	10	13	2
4M-B	85.71	5	18	3
5M-B	84.1	28	37	7
6M-B	83.9	42	99	19
7M-B	83.21	55	109	22
8M-B	82.54	41	104	22

Table 4.9: **(Continued)**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
9M-B	82.04	37	233	51
10M-B	80	9	16	4

The percentage correlations of the top 6 chunks for each of the three datasets is shown in Figure 4.7. Only the top 11 chunks are displayed since *Moodle-A* has only 6 chunks, and to make comparison between the datasets meaningful and consistent. The percentage correlations of all identified chunks along with their respective sizes are shown separately for all three datasets in Figure 4.8, Figure 4.9, and Figure 4.10.

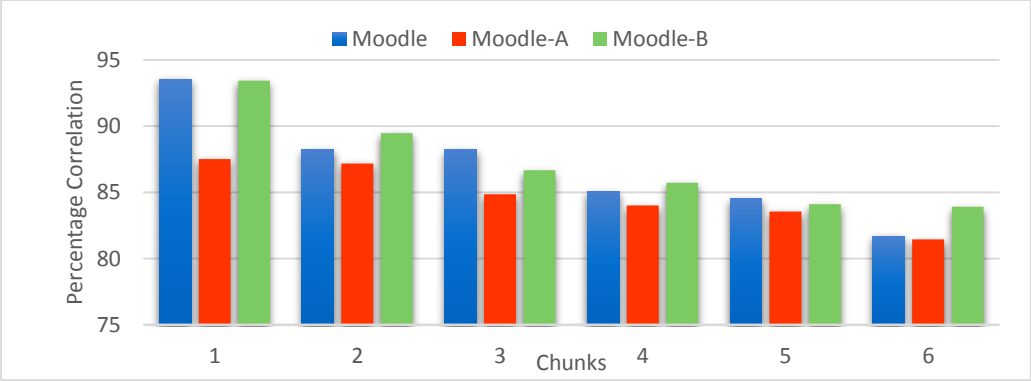

Figure 4.7: **Percentage Correlation of Top-6 Chunks for All Moodle Datasets**

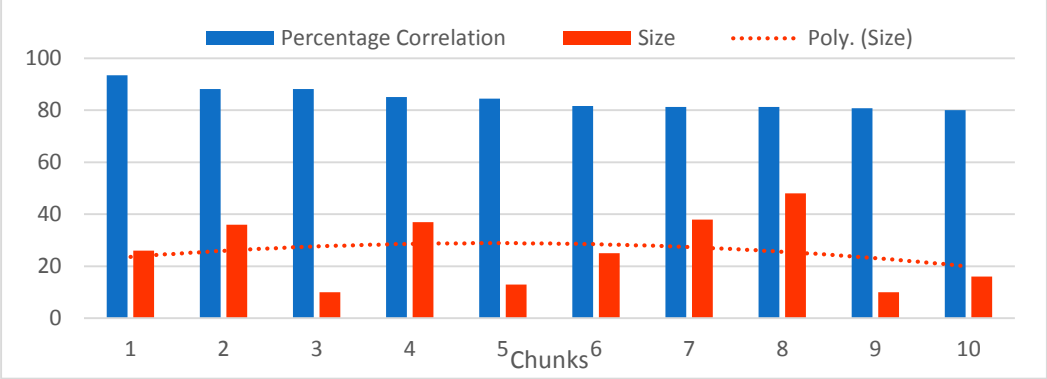

Figure 4.8: **Percentage Correlation and Size of Chunks for Moodle**

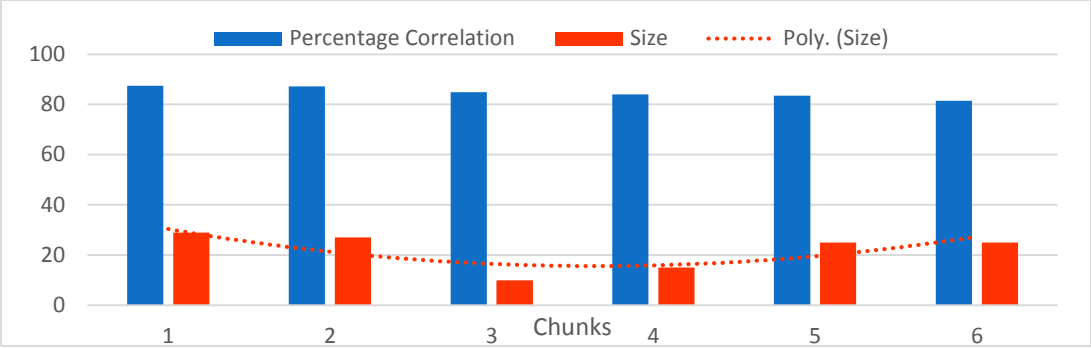

Figure 4.9: **Percentage Correlation and Size of Chunks for Moodle-A**

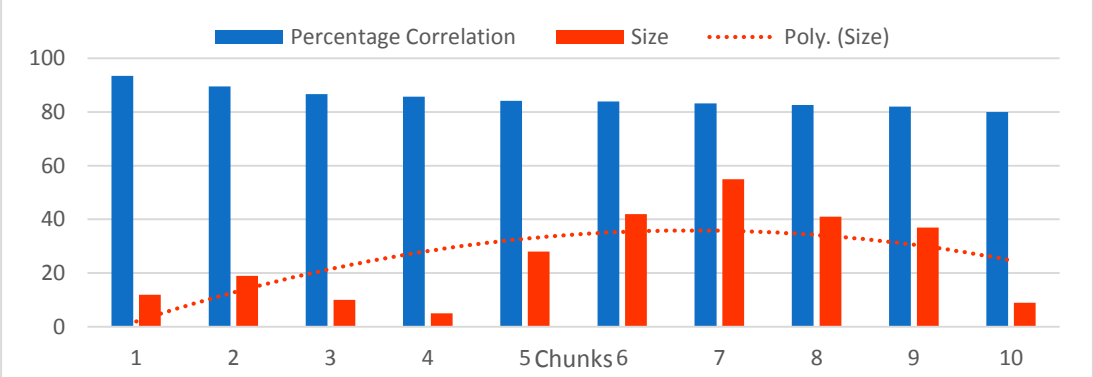

Figure 4.10: **Percentage Correlation and Size of Chunks for Moodle-B**

The following observations were made from the chunks generated from *Moodle*, *Moodle-A*, and *Moodle-B*.

- The file structure within the identified chunks is moderately compact relative to the chunks from Eclipse data. We mapped files within each chunk to their respective directories. We followed discussions with developers in Moodle forums and their online documentation to understand how components are stored in the Git repository [Moodle Docs]. Usually directories correspond to plugins and are of the form /plugin (or subsystem)/subdirectories (component s). Although there is no clear-cut component to directory mapping as in Eclipse, the

paths to various components are listed. Most of the identified chunks have a one-to-many mapping with their corresponding file system directories, unlike those from Eclipse data.

Inferences: As discussed earlier, we attribute this difference to the programming language used by Eclipse, which is Java – object-oriented and has a more modular code structure in contrast to PHP of Moodle, which is usually considered as class-oriented or semi-object-oriented.

Detailed Analysis: All chunks from the three datasets that map to more than one component are listed in Table 4.10, along with the respective chunk identifiers and, path structure of all the components touched by each of those chunks. For example, chunk 8M touches 10 components and 4 subsystems – *mod*, *lib*, *admin*, and *login*, pointing towards re-evaluating architecture of the involved subsystems.

Table 4.10: **Chunk to component mappings for Moodle, Moodle-A, and Moodle-B**

Dataset	Chunk ID	Component Mapping Paths
Moodle	1M	admin/report/
		admin/cli/
		lib/dml/
		lib/simpletest/
	2M	search/documents/
		mod/hotpot/
		mod/resource/
		mod/glossary/
		grade/report/
		lib/form/
	3M	blocks/quiz_results/
	4M	mod/lesson/
		mod/survey/
		mod/imscp/
		mod/label/
		mod/page/

Table 4.10: **(Continued)**

Dataset	Chunk ID	Component Mapping Paths
Moodle	4M	mod/workshop/
	5M	mod/quiz/
	6M	lib/form/
		backup/
		admin/roles/
		admin/report/
		enrol/authorize/
		mod/hotpot/
		mod/wiki/
	7M	calendar/
		blog/
		lib/geoip/
		user/
	8M	lib/dml/
		lib/simpletest/
		lib/ddl/
		mod/data/
		mod/resource/
		mod/lesson/
		mod/scorm/
		admin/roles/
		admin/report/
		login/
	9M	mod/assignment/
		course/report/
		blocks/completionstatus/
		lib/simpletest/
	10M	auth/mnet/
		mnet/
		portfolio/mahara/
		repository/remotemoodle/
Moodle-A	1M-A	grade/report/
		search/documents/
		mod/resource/
		mod/hotpot/
		course/format/
	2M-A	backup/
		admin/roles/
		mod/assignment/
		course/report/
		mod/wiki/
		mod/lesson/
		enrol/authorize/
	3M-A	mod/scorm/
	4M-A	filter/
		course/report/
		course/import/

Table 4.10: **(Continued)**

Dataset	Chunk ID	Component Mapping Paths
Moodle-A	4M-A	enrol/
	5M-A	mod/data/
		mod/resource/
		mod/choice/
		mod/scorm/
		mod/lesson/
		mod/folder/
		mod/page/
		lib/grade/
		user/
	6M-A	mod/quiz/
		mod/forum/
		mod/survey/
Moodle-B	1M-B	lib/dml/
		lib/filestorage/
	2M-B	mod/hotpot/
		mod/glossary
		lib/form/
		search/documents/
	3M-B	admin/user/
	4M-B	mod/quiz/
	5M-B	calendar/
		mod/feedback/
		mod/chat/
		mod/data/
		lib/grade/
		mnet/
		filter/
	6M-B	mod/lesson/
		mod/imscp/
		mod/quiz/
		course/report/
	7M-B	question/type/
		question/format/
		question/engine/
		question/simpletest/
		mod/quiz/
	8M-B	lib/htmlpurifier/
		lib/simpletest/
		lib/filestorage/
		lib/pear/
		blocks/navigation/
		course/report/
	9M-B	blog/
		mod/page/
		mod/label/
		mod/forum/

Table 4.10: **(Continued)**

Dataset	Chunk ID	Component Mapping Paths
Moodle-B	9M-B	admin/roles/ lib/simpletest/ lib/filestorage/ webservice/rest/ user/
	10M-B	course/report/ blocks/completionstatus/ lib/simpletest/

- The datasets *Moodle* and *Moodle-B* generated the highest number of chunks (10) with percentage correlation above 80%, followed by *Moodle-A* (6). As shown in Figure 4.7, the percentage correlation of chunks from *Moodle-A* is lower than that of chunks from *Moodle* and *Moodle-B*. Chunks from *Moodle-B* are only slightly better than the chunks from *Moodle* with respect to percentage correlation, indicating higher coupling between their files relative to the rest.

  Inferences: 1) Since the number of bug fixing MRs is higher in *Moodle*, and *Moodle-B* when compared to *Moodle-A*, the number of chunks generated from both these datasets is higher than *Moodle-A*. Again, we hypothesize that more bug fixes might touch more parts of the system, thus resulting in more chunks. It is important to note that *Moodle* and *Moodle-B* have the same number of chunks although *Moodle* has a higher number of bug fixes. A possible reason for this might be that MRs in *Moodle-B* and *Moodle* span equivalent portions of the system.

  2) *Moodle-A* has the highest number of subsystem couplings, touching 11 different subsystems relative to *Moodle* (7) and *Moodle-B* (7) for the top-6 chunks being compared.

3) *Moodle-A* has the highest number of inter-component couplings when compared to *Moodle* and *Moodle-B*, where 6 chunks are mapped to 25 components, resulting in an average of 4.17 components touched per chunk, followed by *Moodle* and *Moodle-B*, touching 3.5 and 3.3 components per chunk respectively. This reiterates our earlier finding that chunks from *Moodle-B* are slightly better in terms of cohesiveness than those from *Moodle*. The effort to make changes increases with an increase in inter-component or subsystem couplings, thus reducing the system's maintainability resulting in chunks that touch multiple areas of code. This explains the lower percentage correlation of chunks from *Moodle-A*. However, the average inter-component couplings per chunk for Moodle chunks are more than twice as high as those of chunks from Eclipse data.

- There is no consistent increasing or decreasing trend in size of chunks with respect to percentage correlation as shown by the trend lines in Figure 4.8, Figure 4.9, and Figure 4.10.

  Inferences: There is no association between size and percentage correlation of chunks. Size and percentage correlation depend solely on the MRs touching a chunk, and the coupling between files touched by these MRs.

- We identified three stable chunks. Chunk 1M-A evolved into chunk 2M, with 99.55% stability and 28 files in common between the two chunks. There is an increase in both size and percentage correlation as shown in Table 4.7 and Table 4.8. Chunk 2M-A evolved into chunk 6M, with a stability of 68%, and a decrease

in both size and percentage correlation. Chunk 6M-A evolved into chunk 5M, with a stability of 69.23%, and a decrease in size and increase in percentage correlation.

Inferences: There is no association between chunk evolution and how its size and percentage correlation changes.

4.3 Company-X results

We collected change data during the time period 01-Jan-2004 to 31-Dec-2009 for this analysis. This dataset is represented as *Company-X*. We divided *Company-X* into two separate datasets, over the time periods 01-Jan-2004 to 31-Mar-2007, and 01-April-2007 until 31-Dec-2009, represented as *Company-X-A* and *Company-X-B* respectively. The number of bug fixing MRs in each of the datasets is shown in Table 4.11. Duplicate MRs were discarded as with Eclipse and Moodle. The change in the number of non-duplicate MRs that are bug fixes on a half-yearly basis are shown in Figure 4.11. It appears from the plot that major development or maintenance occurred during the period Jun-2005 to Jun-2009, although this information is not publicly available.

Table 4.11: **Number of Bug-Fixing MRs for All Company-X Datasets**

Company-X	Company-X-A	Company-X-B
9949	4994	4955

We enforced a threshold of 70% as the minimum value for percentage correlation of chunks generated by the algorithm, i.e., each of the identified chunks will have at least 70% MRs within chunk. We decreased the threshold since there were very few chunks

identified with percentage correlation above 80%. The algorithm identified 5 chunks for *Company-X*, 4 for *Company-X-A*, and 2 for *Company-X-B*. The values for percentage correlation, size in terms of the number of files contained within the chunk, MRs within chunk, and MRs crossing chunk for all three datasets are listed in Table 4.12, Table 4.13, and Table 4.14.

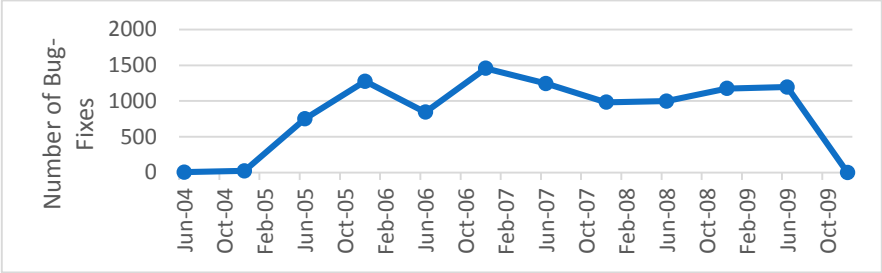

Figure 4.11: **Number of Bug Fixes in Company-X**

Table 4.12: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Company-X**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1X	78.47	12	266	73
2X	78.44	9	291	80
3X	75.54	8	278	90
4X	74.5	13	263	90
5X	73.91	20	17	6

Table 4.13: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Company-X-A**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1X-A	100	24	22	0
2X-A	84.56	27	115	21
3X-A	78.95	15	15	4
4X-A	73.33	28	11	4

Table 4.14: **Percentage correlation, size, MRs within chunk, and MRs crossing chunk for Company-X-B**

Chunk ID	Percentage Correlation	Size	MRs within chunk	MRs crossing chunk
1X-B	89.29	44	25	3
2X-B	80.95	19	187	44

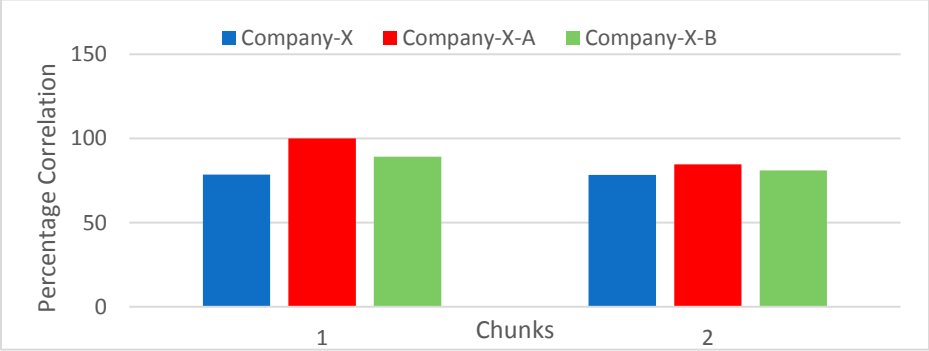

Figure 4.12: **Percentage Correlation of Top-2 Chunks for All Company-X Datasets**

The percentage correlations of the top 2 chunks for each of the three datasets is shown in Figure 4.12. We chose only the top 2 chunks since *Company-X-B* has only 2 chunks. This would make comparison between the datasets meaningful and consistent. The percentage correlations of all identified chunks along with their respective sizes are shown separately for all three datasets in Figure 4.13, Figure 4.14, and Figure 4.15.

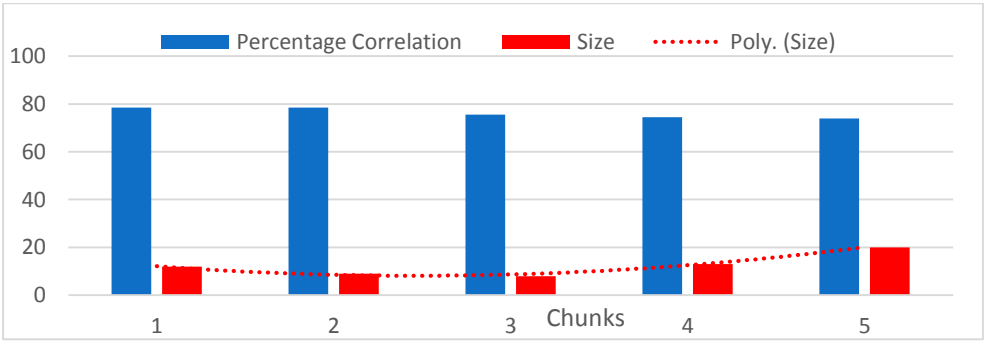

Figure 4.13: **Percentage Correlation and Size of Chunks for Company-X**

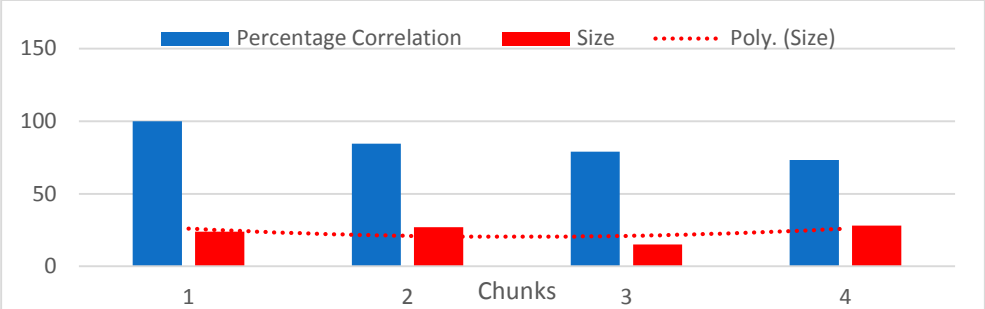

Figure 4.14: **Percentage Correlation and Size of Chunks for Company-X-A**

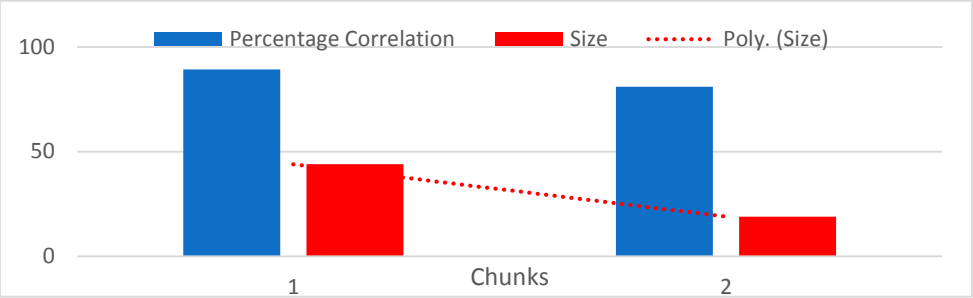

Figure 4.15: **Percentage Correlation and Size of Chunks for Company-X-B**

The following observations were made from the chunks generated from *Company-X*, *Company-X-A*, and *Company-X-B*.

- We do not have information about the directory structure of *Company-X*'s version history repository or how components are stored. Therefore, we could not perform any mapping between files in chunks and the corresponding components or subsystems that they touch.
- The dataset *Company-X* generated the highest number of chunks (5), followed by *Company-X-A* (4), and *Company-X-B* (2) with percentage correlation above 70%. As shown in Figure 4.12, the percentage correlation of chunks from *Company-X* is lower than that of chunks from *Company-X-A* and *Company-X-B*. Chunks from

Company-X-A are better than the chunks from both *Company-X* and *Company-X-B* with respect to percentage correlation, indicating higher coupling between their files relative to the rest.

Inferences: 1) Since the number of bug fixing MRs is higher in *Company-X*, followed by *Company-X-A* and lastly *Company-X-B*, the number of chunks generated from each of these datasets follows the same order. Our hypothesis is that more bug fixes touch more parts of a system, thus resulting in more chunks. 2) The identified chunks are lesser in number and have relatively less correlation when compared to chunks from Moodle and Eclipse. Additionally, there are more bug-fixes for *Company-X* data than that of Moodle. As such, we speculate that *Company-X*'s system architecture underwent more degradation as opposed to Moodle and Eclipse. However, we do not have sufficient subsystem or modular design information to prove our hypothesis.

- There is no consistent increasing or decreasing trend in size of chunks with respect to percentage correlation as shown by the trend lines in Figure 4.13, Figure 4.14, and Figure 4.15.

  Inferences: There is no association between size and percentage correlation of chunks. Size and percentage correlation depend solely on the MRs touching a chunk, and the coupling between files touched by these MRs.

- We identified one stable chunk. Chunk 3X-A evolved into chunk 5X, with 100% stability. There is an increase in size, but a decrease in percentage correlation of chunk 3X-A after its evolution into chunk 5X.

Inferences: We cannot ascertain how chunk size and percentage of a chunk changes after it evolves since we only have one evolved chunk, i.e., 3X-A.

CHAPTER 5. CHALLENGES IN CHUNKING ANALYSIS

This chapter discusses significant challenges faced in our analysis of chunks and their evolution. The challenges are organized into those dealing with the data and those dealing with the proposed algorithm in separate sections. We believe that this will benefit those who wish to replicate the results presented in this thesis by averting the mistakes that we made in earlier stages of this research.

5.1 Challenges of data collection and analysis

In this section, we present the difficulties and obstacles that we faced during data collection, identifying the right type of change data required for the analysis, and during data analysis. Our focus is on the Moodle project as the other two projects were already refined to a certain extent when provided to us.

5.1.1 Non-compliant and multiple data sources

During the initial data collection period, we used Moodle's Fisheye with REST (Representational State Transfer) service to obtain MRs from their version history repository over the time period 01-Jan-2008 to 31-Dec-2011. Fisheye is a revision-control browser and search engine that provides the notion of changesets and changelog, and direct resource-based URLs. Fisheye's REST APIs provide access to data entities via URI (Uniform Resource Identifier). An URI is a string of characters used to identify a name or a web resource over a network, such as the World Wide Web using specific network protocols. Moodle's Fisheye service provides access to revision data from specific repositories by making HTTP (Hypertext Transfer Protocol) requests and

obtaining data in the form of XML or JSON responses. XML (Extensible Markup Language) is a simple text format for encoding documents or web resources in both human readable and machine readable form, and is inefficient as a data interchange format. JSON (JavaScript Object Notation) is a light weight human readable data interchange format, with all the advantages of XML and well suited for data interchange.

Moodle has two main repositories concerning changes made to their source code [Moodle].

- *Moodle Integration*: When a contributor commits changes and pushes them into his public repository, Moodle integrators pull these changes from there and if they like them, they put them into the Moodle Integration repository.
- *Moodle Production*: The integrated changes from the Moodle Integration repository are tested, and if passed, are pushed into the Moodle Production repository.

The development workflow of Moodle is depicted in Figure 5.1, along with their repositories and the steps involved in applying changes to the source code. The Integration repository is named as integration.git, while the Production repository is named as moodle.git in GitHub. This separation of the two repositories was not documented anywhere in their Fisheye service page, and as such we extracted change data from their Integration repository instead of the Production repository, which

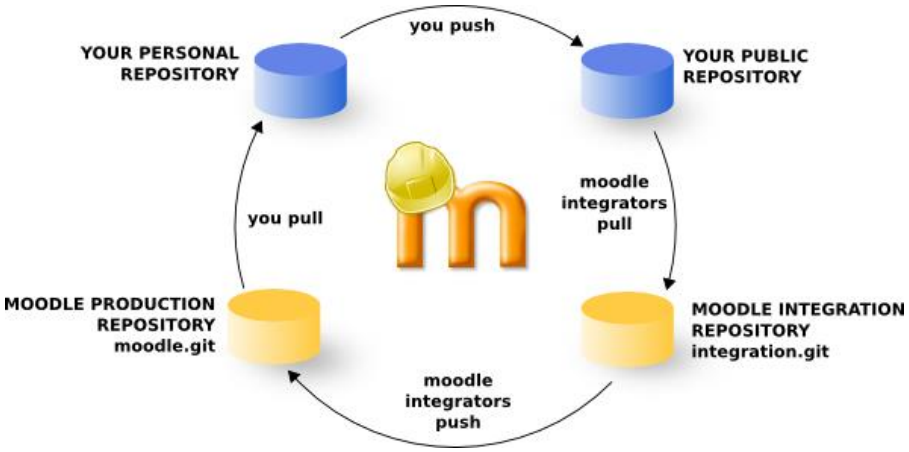

Figure 5.1: **Moodle Development Workflow Using Git [Moodle]**

contains changes that are actually applied to their source code. We discovered the difference between the two repositories when we observed a few commits that were listed as bug fixes, but did not modify any files. Description of the related issue and other comments related to such commits showed that these changes dealt with branch merges in the repository, which are of no interest since they do not change any file and are not really bug fixes. As such, we decided to use MRs from the Moodle Production repository since integration commits are eliminated from the Production repository. We also obtained MRs from Moodle's Git repository (moodle.git) as an alternative data source. After comparing both datasets containing change data from 01-Jan-2008 to 31-Dec-2011, we found that Fisheye and GitHub had a different number of MRs over the same time period. For example, during that time period, there were 5154 bug-fixing commits using Fisheye against 6066 bug-fixing commits in GitHub. This inconsistency between the datasets obtained from GitHub and Fisheye created uncertainty as to which data source to use for our analysis.

We collected a sample of 50 non-matching commits between the two datasets and observed that they were commonly present in both datasets with the same description, author, timestamp and affecting the same set of files, but with a different commit ID. Fisheye might have its own repository different from Git, or an integration of multiple different public repositories, and hence a different list of MRs. Also, there are certain MRs in GitHub that are missing from the dataset obtained using Fisheye, for reasons that are unknown and undocumented. Based on information from Moodle forums and their JIRA issue tracker, we found that Moodle is planning to replace their Fisheye service with either Stash or by linking issues directly to GitHub. In addition, Fisheye is a web service, which is slow and inefficient when compared to cloning the Git repository, which is a faster and more reliable way of obtaining Moodle's version history data. Hence, we decided to use MRs obtained from the Git repository as the dataset to be used in this work. It is important to note that there are inconsistencies between the Git repository and Production repository.

5.1.2 Does a commit correspond to a single bug fix?

Changes that touch more than 100 files do not correspond to meaningful modification requests [Ying et al., 2004]. We found certain bug-fixing commits in the Moodle dataset that touched more than 500 files, which made us skeptical that they really corresponded to an atomic bug fix. We looked into a sample of 30 bug-fixing commits that touched more than 100 files to check if they are originally multiple bug fixes that were committed together. We found that the majority of them were auto-installed header changes or library imports, which clearly do not represent a single bug-

fix, but rather multiple bug fixes committed as a single change. For example, in the Moodle dataset, there is an MR that touched 633 files, and involves replacing file permissions of a number of arbitrary files from 755 (executable) to the right permission, 644. The files that were touched by this MR were spread all across the repository and hence mapped to many directories in the Git repository, indicating that it probably touched more than one component. However, one might be able to design the system in such a way that only one component knows the file permission for each file. A change of such file permissions would then require a change to that component only, and such a component would represent a chunk.

There are other instances where it is difficult to make a clear judgment if a commit corresponds to exactly one bug fix for a variety of reasons that include lack of documentation, clarity in issue description or comments, lack of experience in distinguishing such commits, etc. This ambiguity hampers a primary assumption in our analysis that each commit represents a single MR. As such, the algorithm identified independent chunks out of which one chunk touched more than 2000 files while the remaining chunks together touched less than 300 files, thus resulting in a bias in our analysis of the generated chunks. Hence, we excluded MRs that touched 50 files or more from our analysis. We view this as reducing the number of false positives in chunk identification.

Still speculative of the results, we again picked a sample of all commits that touched 20 files or more and less than 50 files and distinguished them into single or multiple bug-fixing MRs based on the description and comments from developers who

fixed the issue [Moodle tracker]. If the commit description and related documentation hints at a single bug that is fixed by the commit, we refer to that commit as a single bug-fixing MR. Otherwise, if the commit description points towards fixing more than one bug, we refer to that commit as a multiple bug-fixing MR. After review by Weiss and Mockus, who are well acquainted with and share a good knowledge of software chunks, we observed that 12 out of 24 commits, i.e. 50% of the commits from the sample that we picked, were in fact multiple bug fixes listed as a single commit. Therefore, we thought it pertinent to exclude commits that touch more than 20 files from our analysis to ensure that each bug fix corresponds to a single MR. Such commits account for about 99.6% of all bug-fixing commits; almost no data is lost during this filtering process.

Considering such cases makes it clear that identification and analysis of chunks must include careful manual data analysis. In other words, validating the data is critical.

5.1.3 No availability of sources to verify hypotheses

There is no design documentation accessible for any of the projects studied in this work in order to verify our hypothesis that chunks represent design modules and to confirm the suggested source code or structural refactorings. It is possible that the algorithm identifies chunks that are extremely large, i.e., they contain more than 200 files, and yet have a very small percentage of MRs crossing the chunk. Our hypothesis is that when chunks grow that large in size, they might be representing changes to most of an entire module, component or a subsystem. Again, there is no way to verify this unless we discuss it with an architect or developer who is familiar with the architecture or

design structure of the system, or have access to appropriate design artifacts, such as the modular structure, or interface specifications. For both Eclipse and Company-X projects, we were not provided with any similar or corresponding design documentation. We were also unable to locate such information anywhere in their available online documentations.

5.2 Algorithmic challenges

As discussed in Chapter 3, the algorithm picks a random set of files from which the optimization criteria used for evaluation, i.e., percentage correlation and MRs crossing chunk, are calculated. This set of files forms the initial candidate chunk only if it meets the optimization threshold. The process is repeated, and the initial candidate chunk is replaced by another candidate only if it has a higher percentage correlation while still satisfying the threshold value for MRs crossing chunk. When the algorithm terminates after a certain specified time interval, we get the chunk with the highest percentage correlation.

After a detailed analysis of the top ten resulting chunks identified by one of the algorithms in the family of algorithms discussed in Chapter 3, different from the algorithm used in this study, we observed that they were not independent. i.e., there were files common among multiple chunks. This is because we overlooked the notion that chunks are independent by definition. Therefore, a clear understanding of the concept of chunks is paramount in arriving at the right algorithm for chunk identification. We fixed this issue by removing the files and MRs within the generated chunk from the files and

MRs being considered to identify the next chunk, thus eliminating the possibility of chunks sharing files between them.

It also took us a considerable amount of time to discover that another algorithm in the family of algorithms identifies multiple chunks as a single chunk, resulting in chunks with sub-chunks. As an example, two independent chunks each with 80% correlation can be identified as a single chunk with a percentage correlation of 80% if the algorithm randomly picks a subset of files that constitute the two independent chunks. This is an undesirable aspect since it is possible that such an algorithmic approach can identify a chunk that contains changes touching an entire subsystem by assimilating multiple chunks that independently change different parts of that subsystem. This makes it impossible to identify structural weaknesses at the component level or other lower levels of the system design. We identified this shortcoming of the algorithm by observing that files contained in a chunk came from different components as indicated by their file paths. A combination of one-time changes, i.e., changes such that files touched by such changes are not affected by any other changes, will not have any MRs crossing chunk, thus resulting in false perfect chunks. We resolved this issue by enforcing the constraint that every file in a chunk is modified together with at least one other file in the same chunk, and excluding changes that touch less than two files. Such issues cannot be identified easily unless one has a deep understanding of the change data being studied, the logical couplings generated by such data, and their relation to chunks. Also note that there may be many potential chunks in a large system, but if no changes are made to the parts of the system where they reside, then they will never be detected

by the algorithm. Chunks can only be identified where there is considerable change going on.

CHAPTER 6. VALIDATION

As in any study dealing with data from open source projects with undergoing development, it is of importance to understand and clean the data, and to ensure that both the data and the algorithm used for analysis are correct and useful. Reliability of data and the proposed algorithmic approach is important to replicate the findings as well as to strengthen our analysis. In this chapter, a detailed description of our validation and verification processes is presented to provide guidance and confidence to those who wish to use the chunking approach, either with the same data used in this study, or version history data of other projects. The methods used to validate data and approach are organized into separate sections.

6.1 Validation of data

Verification of change data from different projects is crucial to ensure that we have used the right data for the algorithm to identify valid chunks. A fundamental assumption of our approach in this regard is that every commit in the repository is equivalent to a single MR. As discussed in Chapter 5, we only included MRs that touched 20 files or less for the identification and analysis of chunks presented in this thesis. For Moodle data, for validation purposes, we collected and manually inspected a random sample of 50 MRs that touched 20 files or less in order to make sure that each MR corresponded to a single change request in general and, a single bug fix in particular. After a careful observation of developers' comments on how the issue was

resolved, and description of the issue, we found that about 92% of them represented single changes. These results were again cross-verified by Mockus and Weiss.

We also observed that there were multiple duplicates of a few commits, with the same author, description of change, timestamp, and touching the same set of files, but with different commit IDs. Such duplicates usually arise when a developer accidentally commits the same change more than once. Since it is difficult and time consuming to identify all duplicates and exclude such commits in large datasets, we ensured that such duplicate MRs do not have any effect on the chunks generated by the algorithm by imposing a constraint to disallow duplicates files within a chunk. Therefore, there is no increase or decrease in chunk size, or the number of chunks obtained even if there are duplicate MRs in the datasets. However, we excluded duplicates from our datasets for the purpose of analysis in this thesis.

Eclipse data provided by Krishnan was validated, and discussed in his work [Krishnan, 2013]. In order to validate if the collected data represented the real picture, Krishnan and his team communicated with developers at Eclipse through forums that were actively maintained by Eclipse community. The bug database provided to us was originally provided to Krishnan by a developer team at Eclipse, and this database is actively used by the developers. The fields in the bug database are verified by the developer who fixes the bug, in case it is wrongly entered by a user. Hence, we can place confidence that issues stored in this database are actually bug fixes as required for our analysis since it is maintained and supervised by developers who make changes and, commit fixes for the bugs. For the Europa dataset used in this work, six-digit strings

from the CVS log data were matched to bug IDs in the bug database. A manual review was performed to confirm that no entries containing the word “bug” existed, which were not caught by the performed pattern matching. To ensure that data resulting from different sources contained matching file names, instances of a certain file pattern was removed from all files to make them uniform. A CVS rlog tool with date filtering was used to make sure that all data sources covered the same time periods.

Although we do not have any issue descriptions for each commit or MR for change data from Company-X, we are confident of its validity as it has been verified by developers and architects at Company-X.

6.2 Validation of algorithm

We performed a mapping of files within a chunk to the corresponding directory to which it belongs in the Moodle Git repository. A chunk consisting of mappings to a large number of different directories might indicate the possibility of sub-chunks within that chunk. So, we manually inspected the MRs within the generated chunks to make sure that every file has been changed together with at least one other file within that chunk, so as to validate that the resulting candidates are in fact chunks rather than just a set of highly coupled files, and that they are atomic without any sub-chunks. We also verified that every MR touching the chunk has at least one file modified by that MR within that chunk. We used a Python script to detect any common file names between the chunks generated by the algorithm. This ensures that the algorithm identifies

independent chunks. Thus, we can place confidence that our algorithm identifies valid chunks from version history information.

CHAPTER 7. CONCLUSION AND FUTURE WORK

This chapter discusses the limitations of this work and scope for future work in the field of software measurement using chunks and presents the conclusions drawn from this work.

7.1 Conclusions

This thesis is an attempt to analyze tightly coupled changes represented by chunks in large software projects that can assist in improving software design and implementation methods. This work can be replicated by using any alternative algorithmic approach that can identify areas of code in a software system that tend to change together. The backbone for generating valid chunks is to collect the right type of change history data consisting of MRs that correspond to individual single changes, along with information about each MR, such as the type of change, time of change, and files touched by the change. Our approach can be used to identify chunks in any software project, provided it has the required change data available in its version history repositories. This work can be further extended to incorporate information such as developer effort, subsystem or module artifacts that can strengthen the findings in this thesis.

The conclusions drawn from this thesis are as follows.

- We have successfully identified and analyzed chunks in three major software projects – Eclipse, Moodle, and Company-X.

- Chunks from Eclipse are highly modular and more structured with respect to the file structure within each chunk, and have a higher coupling between the files constituting them, followed by Moodle and Company-X in that order. This may be because Eclipse uses Java as its primary programming language, object-oriented while Moodle's PHP is only semi-object-oriented and Company-X uses mostly C.
- There is no association between chunk size and percentage correlation.
- A higher number of inter-component or subsystem couplings requires making many changes since making a change in one component requires changes to other components as well, thus resulting in a decrease in percentage correlation within chunks. Percentage correlation of chunks and maintainability of the system are therefore inversely correlated.
- As a system evolves over time, making further changes becomes increasingly difficult due to multiple reasons as discussed in Chapter 1. The evidence was given by Gall and others in their work [Gall et al., 1998]. Therefore we hypothesized that as chunks evolve, there will be an increase in size and a decrease in percentage correlation. This is because over time making a change requires touching more areas of code, with increasing dependencies between components or subsystems leading to an increase in size and a decrease in percentage correlation. However, we could not verify this due to insufficient number of chunk evolutions in Eclipse, Moodle, and Company-X.

- We identified merging of chunks over time in Eclipse into a chunk with reduced percentage correlation and relatively less cohesive (lesser percentage correlation) than the merging chunks. This suggests that over time, coupling between files increases, thereby resulting in an increase in difficulty to independent changes to different parts of a system, and indicates that the components touched by these chunks might require refactoring or redesign.

7.2 Limitations and future work

A major limitation of this thesis is the non-availability of design documentation, such as a module guide of the type discussed by Parnas, Clements, and Weiss [Parnas et al., 1985], uses structure, or module interface specifications. Hence, we could not find any substantial evidence or proof that chunks represent design modules and are not just empirical constructs. Mapping chunks to modules of a software system can provide wide scope for locating defects in software design or limitations in implementation techniques followed in the organization or company. If a chunk is mapped to more than one module, it might suggest necessary source code refactoring. On the other hand, mapping of one or more chunks to the same module might indicate probable flaws in the system's modular design, i.e., it is possible that the module contains more than one design decision, and can be therefore decomposed into two or more modules.

Studying evolution of chunks over time can help answer many questions or hypotheses with respect to chunks and their attributes. An increase in the size of chunks over time along with an increase in effort might provide a measure of difficulty to make

a further change as software evolves. It might also indicate increasing inconsistencies between how a system is designed, and how it is actually being implemented, which puts forth the need to refactor code, in such a way that it conforms to its design as closely as possible. Identifying such necessary refactorings earlier in a project's life cycle can prevent further decay of the system's architecture, and save time and effort for making future changes to software. We can also evaluate developer performance over time by taking into account developer information, such as proportion of total changes made within a chunk, effort in terms of the number of hours spent in making all changes within a chunk, complexity of committed changes, and proportion of changes crossing chunk. Developers who make a higher number of changes with as few changes crossing a particular chunk as possible may have better performance relative to others with respect to making changes that conform to a system's design. Also, tracking the effort spent per chunk along with the complexity of changes made and the percentage of changes crossing that chunk can give insights into how a developer's performance evolves over time with respect to making changes that are consistent with a system's design.

It would be of interest to see how software maintainability changes by observing how total effort spent on making all changes within a stable chunk evolves over time. An increase in effort indicates a decrease in code maintainability and vice versa. Maintainability can also be determined by identifying similar changes over a time period, and observing how the number of files touched by such MRs changes over time. Alternatively, identifying stable chunks and determining trends in their size with respect

to the number of files can expose trends in maintainability of a system. An increase in chunk size would imply a decrease in the system's maintainability. Additionally, mapping each of the MRs crossing a chunk to the respective modules can uncover hidden inter-modular couplings. An increase in number of such couplings points towards decreasing maintainability.

The concept of *stability* can be further strengthened by considering other measures such as effort. Even if a chunk grows or shrink in size over time, if the effort remains constant, then we might call such a chunk a stable chunk, while considering that at least 65% of the files remain constant over its evolution. Also, considering LOC for size of a chunk rather than the number of files might prove to be a better measure.

We believe that this work is a small leap towards providing empirical evidence for information hiding design shortcomings of a software system by using chunks. We hope that this work paves way for further research in the field of software chunking.

BIBLIOGRAPHY

Arnold R. S. Software Reengineering. In *Proceedings of the IEEE Computer Society Press*, 1993.

Bieman, J. M., Andrews, A. A., and Yang, H. J. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 44–53, May 2003.

Breu, S., Zimmermann, T., and Lindig, C. Mining Eclipse for Cross-Cutting Concerns. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR ’06, pages 94-97, 2006.

Constantine, L., Myers, G., Stevens, W. Structured Design. Book, *Classics in Software Engineering*, pages 205-232, 1979.

D’Ambros, M., Lanza, M., and Robbes, R. On the Relationship between Change Coupling and Software Defects. In *Proceedings of the 16th Working Conference on Reverse Engineering*, WCRE ’09, pages 135-144, Oct. 2009.

Gall, H., Hajek, K., and Jazayeri, M. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance*, ICSM ’98, pages 190-198, Nov. 1998.

Gall, H., Jazayeri, M., and Krajewski, J. CVS Release History Data for Detecting Logical Couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE ’03, pages 13-23, 2003.

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering*, 26(7):653-661, 2000.

Griswold, W. G., and Notkin, D. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228-269, July 1993.

Hamill, M., Goseva-Popstojanova, K. Common Trends in Software Fault and Failure Data. *IEEE Transactions on Software Engineering*, 35(4):484-496, July 2009.

Herbsleb, J. D., Mockus, A., Finholt, T. A., and Grinter, R. E. An Empirical Study of Global Software Development: Distance and Speed. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE ’01, pages 81-90, 2001.

Horwitz, S. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, June 1990.

Krishnan, S. Evidence-based Defect Assessment and Prediction for Software Product Lines. PhD Thesis, Iowa State University.

Mockus, A., and Weiss, D. M. Globalization by Chunking: A Quantitative Approach. IEEE Software, 18(2):30-37, Mar./Apr. 2001.

Mockus, A., and Weiss, D. M. The Chunking Pattern. DAPSE, 2013.

Neamtiu, I., Foster, J. S., and Hicks, M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR ’05, pages 1-5, 2005.

Parnas, D. L. On the Criteria To Be Used in Decomposing a System into Modules. Communications of the ACM, 15(12):1053-1058, Dec. 1972.

Parnas, D. L. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE-16, pages 279-287, May 1994.

Parnas, D. L., Clements, P. C., and Weiss, D. M. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, 11(3):259-266, 1985.

Pearse, T., and Oman, P. Maintainability Measurements on Industrial Source Code Maintenance Activities. In *Proceedings of the International Conference on Software Maintenance*, ICSM ’95, pages 295-303, 1995.

Steff, M., and Russo, B. Co-evolution of Logical Couplings and Commits for Defect Estimation. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR ’12, pages 213-216, 2012.

Weiss, D. M. and Lai, C. T. R. Software Product-Line Engineering: A Family-based Software Development Process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Wong, S., Cai, Y., Kim, M., and Dalton, M. Detecting Software Modularity Violations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE ’11, pages 411-420, 2011.

Yang, W. Identifying Syntactic Differences between Two Programs. Software - Practice and Experience, 21(7):739–755, 1991.

Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9), pages 574-586, 2004.

Zimmermann, T., Diehl, S., Zeller, A. How History Justifies System Architecture (or Not). In *Proceedings of the 6th International Workshop on Software Evolution*, pages 73-83, 2003.

Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429-445, 2005.

Moodle.git. In *Moodle Git repository*. Retrieved Feb, 2013, from

http://git.moodle.org/gw. Used Feb, 2013 to May, 2013.

Issues. In *Moodle issue tracker*. Retrieved Aug, 2012 to May, 2013, from

https://tracker.moodle.org/browse/MDL. Used Aug, 2012 to May, 2013.

Plugins. In *Moodle documents.* Retrieved Mar, 2013 from

http://docs.moodle.org/dev/Plugins. Used Mar, 2013.

Development workflow for Git. In *Moodle documents for developers*. Retrieved Apr, 2013, from http://docs.moodle.org/dev/Git_for_developers. Used May, 2013.

APPENDIX. ALGORITHM IMPLEMENTATION

The Python script used in this work to implement the algorithm for chunk identification is as follows.

```
#required imports
import random
import heapq
import datetime
import sqlite3
import time
import sys
import copy
from operator import itemgetter
from collections import OrderedDict
class ModReq:
    def __init__(self, _id, _files):
        self.id = _id
        self.files = set(_files)
```

```
    def doesTouch(self, id):
        return id in self.files
    def getFiles(self):
        return self.files
class SingleFile:
    def __init__(self, _id, _name):
        self.name = _name
        self.id = _id
        self.mrs = set()
    def getName(self):
        return self.name
    def getID(self):
        return self.id
    def getTouchingMRs(self):
        return self.mrs
```

```
    def addTouchingMR(self, other):
        self.mrs.add(other)
    def __repr__(self):
        return self.name
    def __str__(self):
        return self.name
# Given a sequence, seq, returns a random element of seq
def randomElement(seq):
    return list(seq)[random.randint(0, len(seq) - 1)]
#initializes a chunk candidate
class ChunkCandidate:
    def __init__(self):
        self.filesConsidered = 0
        self.optionsTested = 0
        self.files = set()
        self.mrs = set()
        self.correlation = 0.0
```

```
        self.MRsWithin = 0
class Chunker:
    def __init__(self, pathToDB):
        # Set to true if we can use RAM to cache data
        # If true, all MRs will be cached by the end of initialization (prior to calling
            getPercentWithinChunk)
        self.useRAM = True
        self.error = False
        if self.useRAM:
            self.MRtoFilesDict = dict()
            self.FiletoMRsDict = dict()
            self.FileIDtoNameDict = dict()
        self.paused = False
        self.candidatesTried = 0
        self.dbname = pathToDB
        # Initialize database connection
        conn = sqlite3.connect(self.dbname)
        conn.text_factory = str
```

```
self.dbcursor = conn.cursor()

# Start by getting all of the changesets we're dealing with

query = "select changesets.id from changesets where issuetype='problem' and
    filecount<=20 and datetime(date,'unixepoch','localtime')<'2010-01-01' and
    datetime(date,'unixepoch','localtime')>='2007-04-01';"

self.dbcursor.execute(query)

# At this point, dbcursor contains our results. We need to store the data it provides
    before we can use it to execute another query

activeCSIDs = self.dbcursor.fetchall()

self.files = set()

self.mrs = set()

self.allMR = set()

# These lines just create a set of all files touched by all active CSIDs - this speeds
    up processing later

if self.useRAM:

    for csid in activeCSIDs:

        tempSet = self.getCSFiles(csid[0], self.dbcursor)

        self.mrs.add(csid[0])
```

```
                self.allMR.add(csid[0])
                self.files |= tempSet
                self.MRtoFilesDict[csid[0]] = tempSet
            else:
                for csid in activeCSIDs:
                    self.mrs.add(csid[0])
                self.files |= self.getCSFiles(csid[0], self.dbcursor)
            # End result is that self.files is the set of all files affected by active CSIDs
            # If self.useRAM is set, self.MRtoFilesDict is a dictionary (map) from CSIDs to
              their files, which lets us skip future database queries
    # Return best candidate after timelimit; sets self.error to true if ctrl-c is captured
        def getNext(self, timeLimit):
            retChunks = []
            MRsTested = set()
            selMR = set()
            retVal = ChunkCandidate()
            retVal.filesConsidered = len(self.files)
            startTime = time.time()
```

```
endTime = timeLimit + startTime

print "init"

while True:

    if not self.paused:

        try:

            selMR = self.mrs-MRsTested

            if len(selMR)==0:

                return retVal

            randMR = randomElement(selMR)

        # Randomly select MR that affects more than 5 files

            if len(self.MRtoFilesDict[randMR]) < 5:

                continue

            print "rand" + str(randMR)

            if randMR not in MRsTested:

                MRsTested.add(randMR)

            # Temporary chunk created, if it meets the correlation threshold, it is added
    to the candidate chunks

            tempRetVal = ChunkCandidate()
```

```
        curMR = randMR
        curSet=set()
        curSet = self.getCSFiles(curMR, self.dbcursor)
        MRsWithin = self.getMRsWithinChunk(curSet, self.dbcursor)
        correlation = self.getPercentWithinChunk(curSet, self.dbcursor)
        print "MRs within chunk" + str(MRsWithin) + "correlation" +
str(correlation)
        if correlation >= tempRetVal.correlation:
            tempRetVal.MRsWithin = MRsWithin
            tempRetVal.correlation = correlation
            tempRetVal.files = curSet
        mrList = set()
        mrList.add(randMR)

        if tempRetVal.correlation >= retVal.correlation and tempRetVal.correlation
>=70 and tempRetVal.MRsWithin >= 10:
            retVal.MRsWithin = tempRetVal.MRsWithin
            retVal.correlation = tempRetVal.correlation
```

```
            retVal.files = tempRetVal.files
            retVal.optionsTested = tempRetVal.optionsTested
        while tempRetVal.correlation > 60.0:
            fileset = set()
            tempFileSet = set() #todo
            temp = set() #set of all MRs that touch curSet
            tempCurset = copy.deepcopy(curSet)
            # Store all the MRs that affect the files in question in temp
            for f in curSet:
                tempMRs = self.getFileCSs(f, self.dbcursor)
                temp |= tempMRs
# Remove the MR randomly chosen from this list because we have already
considered the files it affects
            for m in mrList:
                if m in temp:
                    temp.remove(m)
            if len(temp)==0:
                break
```

```
        #Find a list of all files touched by the MRs in temp
        for mr in list(temp):
            tempFileSet |= self.getCSFiles(mr, self.dbcursor)
        tempFileSet = tempFileSet - tempCurset
        #Add each file to curSet to see if correlation increases, keep file if it does,
else go to next
        for file in list(tempFileSet):
           f=set()
           f.add(file)
           shouldBreak = time.time() > endTime
           if shouldBreak:
               if tempRetVal.correlation >= retVal.correlation and
tempRetVal.correlation >=70 and tempRetVal.MRsWithin >= 10:
                   retVal.MRsWithin = tempRetVal.MRsWithin
                   retVal.correlation = tempRetVal.correlation
                   retVal.files = tempRetVal.files
                   retVal.optionsTested = tempRetVal.optionsTested
                   return retVal
```

```
            tempRetVal.optionsTested = tempRetVal.optionsTested + 1
            tempSet = set()
            tempSet = copy.deepcopy(curSet)
            tempSet |= f
            MRsWithin = self.getMRsWithinChunk(tempSet, self.dbcursor)
            correlation = self.getPercentWithinChunk(tempSet, self.dbcursor)
            if correlation >= tempRetVal.correlation:
                tempRetVal.MRsWithin = MRsWithin
                tempRetVal.correlation = correlation
                tempRetVal.files = tempSet
        curSet = tempRetVal.files
        if len(tempCurset)==len(curSet):
            break
    # Adds chunk only if temporary chunk has the number of crossing MRs less
than that of current chunk
    if tempRetVal.correlation >= retVal.correlation and tempRetVal.correlation
>=70 and tempRetVal.MRsWithin >= 10:
        retVal.MRsWithin = tempRetVal.MRsWithin
```

```
                retVal.correlation = tempRetVal.correlation
                retVal.files = tempRetVal.files
                retVal.optionsTested = tempRetVal.optionsTested

            shouldBreak = time.time() > endTime
            if shouldBreak:
                print str(len(retVal.files))
                return retVal
        except KeyboardInterrupt:
            self.error = True
            return retVal
def pause(self):
    self.paused = True

def resume(self):
    self.paused = False
def getPercentWithinChunk(self, files, cursor):
```

```
    setOfMRsForFiles = set()
    if len(files) == 0:
        return 0
    for file in files:
        setOfMRsForFiles |= self.getFileCSs(file, cursor)
    numMRs = len(setOfMRsForFiles) * 1.0
    if numMRs == 0:
        return 0
    MRsContained = 0.0
    for mr in setOfMRsForFiles:
        if self.getCSFiles(mr, cursor).issubset(files):
            MRsContained = MRsContained + 1
    return (MRsContained / numMRs) * 100.0


def getMRsWithinChunk(self, files, cursor):
    setOfMRsForFiles = set()
    if len(files) == 0:
```

```
            return 0
        for file in files:
            setOfMRsForFiles |= self.getFileCSs(file, cursor)
        numMRs = len(setOfMRsForFiles) * 1.0
        if numMRs == 0:
            return 0
        MRsContained = 0.0
        for mr in setOfMRsForFiles:
            if self.getCSFiles(mr, cursor).issubset(files):
                MRsContained = MRsContained + 1
        return MRsContained
    def getCSFiles(self, int_id, cursor):
        if self.useRAM:
            if int_id in self.MRtoFilesDict:
                return self.MRtoFilesDict[int_id]
        query = "SELECT links.file_id FROM links WHERE cs_id=" + str(int_id) + ";"
        retval = set()
```

```
    cursor.execute(query)
    for row in cursor:
        retval.add(row[0])
    return retval
def getFileCSs(self, int_id, cursor):
    #print int_id
    if self.useRAM:
        if int_id in self.FiletoMRsDict:
            return self.FiletoMRsDict[int_id]
    query = "SELECT links.cs_id FROM links WHERE file_id=" + str(int_id) + ";"
    retval = set()
    cursor.execute(query)
    for row in cursor:
        if row[0] in self.allMR:
            retval.add(row[0])
    return retval
def fileIDtoName(self, int_id, cursor):
```

```
        if self.useRAM:
            if int_id in self.FileIDtoNameDict:
                return self.FileIDtoNameDict[int_id]
        query = "SELECT files.filename FROM files WHERE id=" + str(int_id) + ";"
        cursor.execute(query)
        result = cursor.fetchone()[0]
        if self.useRAM:
            self.FileIDtoNameDict[int_id] = result
        return result
    #returns the text to be written to the output file – chunks with file names of files within
            each chunk and percentage correlation, MRs within and crossing chunk
    def stringifyCandidate(self, result, cursor):
        setOfMRsForFiles = set()
        if len(result.files) == 0:
            return 0
        print result.files
        for f in result.files:
            setOfMRsForFiles |= self.getFileCSs(f, cursor)
```

```
        numMRs = len(setOfMRsForFiles) * 1.0
        if numMRs == 0:
            return 0
        MRsContained = 0.0
        for mr in setOfMRsForFiles:
            if self.getCSFiles(mr, cursor).issubset(result.files):
                MRsContained = MRsContained + 1
        print str(MRsContained) + " " + str(result.correlation)+ "
            "+str(MRsContained/numMRs)
        retval = "\n\n" + str(result.correlation) + "% correlation within chunk made up of "
            + str(result.MRsWithin) + " MRs within chunk and " + str(numMRs) + " total
            MRs and size " + str(len(result.files)) + ":\n" + "\n".join([self.fileIDtoName(file,
            cursor) for file in result.files])
        return retval


    def updateParameters(self, **kwargs):
        if "minFiles" in kwargs:
            self.minFilesPerChunk = kwargs["minFiles"]
```

```
        if "maxFiles" in kwargs:
            self.maxFilesPerChunk = kwargs["maxFiles"]


if __name__ == "__main__":
    # Set to the path of your database on your hostname (this allows easier testing with
        svn checkouts on different machines
    # Socket.gethostname() returns the hostname of your machine (doesn't actually require
        that a socket be created)
    import socket
    if socket.gethostname() == 'Rac-Twin':
        dbname = 'companyx.db'
    print "Start"
    c = Chunker(dbname)
    results = []
    for i in range(int(sys.argv[1])):
        print "Starting to process a chunk with " + str(len(c.files)) + " still under
            consideration.\n"
        results.append(c.getNext(int(sys.argv[2])))
```

```
        # list index -1 is last element; this removes the previously returned files
        # from consideration for the next chunk -  a file can belong to at most one chunk
        c.files = c.files - results[-1].files
        setOfMRsForFiles = set()
        if len(results[-1].files) == 0:
            continue
        if len(c.files) == 0:
            break
        for file in results[-1].files:
            setOfMRsForFiles |= c.getFileCSs(file, c.dbcursor)
        # Remove MRs already in a chunk for the next candidate chunk as we are looking
            for independent chunks
        for mr in setOfMRsForFiles:
            if mr in c.mrs:
                c.mrs.remove(mr)
        if c.error:
            break
    print "Printing results to ./out.dat and exiting\r\n"
```

```
f = open("C:\Python27\AvayaNewLast." + str(datetime.date.today()) + ".txt", 'a')

f.write("\n" + '-' * 80 + "\n")

f.write("\n" + str(datetime.datetime.now()))

for chunk in results:

    f.write(str(c.stringifyCandidate(chunk, c.dbcursor)))
```