Mining Software Repositories to Determine the Impact of Team Factors on the Structural Attributes of Software

Ahmmad Youssef

Supervisors: Dr. Andrea Capiluppi
Prof. Tracy Hall

Department of Computer Science
Brunel University, London

This dissertation is submitted for the degree of
Doctor of Philosophy
March 2019

Dedication to be completed on final submission.

Declaration

I hereby declare that, except where specific reference is made to the work of others, the contents of this thesis is original and has not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other, University. This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Ahmmad Youssef
March 2019

Acknowledgements

Acknowledgements to be completed on final submission.

Abstract

Software development is intrinsically a human activity and the role of the development team has been established as among the most decisive of all project success factors. Prior research has proven empirically that team size and stability are linked to stakeholder satisfaction, team productivity and fault-proneness. Team size is usually considered a measure of the number of developers that modify the source code of a project while team stability is typically a function of the cumulative time that each team member has worked with their fellow team members. There is, however, limited research investigating the impact of these factors on software maintainability - a crucial aspect given that up to 80% of development budgets are consumed in the maintenance phase of the lifecycle.

This research sheds light on how these aspects of team composition influence the structural attributes of the developed software that, in turn, drive the maintenance costs of software. This thesis asserts that new and broader insights can be gained by measuring these internal attributes of the software rather than the more traditional approach of measuring its external attributes. This can also enable practitioners to measure and monitor key indicators throughout the development lifecycle taking remedial action where appropriate.

Within this research the GoogleCode open-source forge is mined and a sample of 1,674 Java projects are selected for further study. Using the Chidamber and Kemerer design metrics suite, the impact of development team size and stability on the internal structural attributes of software is isolated and quantified. Drawing on prior research correlating these internal attributes with external attributes, the impact on maintainability is deduced.

This research finds that those structural attributes that have been established to correlate to fault-proneness - coupling, cohesion and modularity - show degradation as team sizes increase or team stability decreases. That degradation in the internal attributes of the software is associated with a deterioration in the sub-attributes of maintainability; changeability, understandability, testability and stability.

Publications

Some aspects of this work have been published in various venues, receiving formal feedback from reviewers which, in turn, has impacted upon this research. They are listed below in chronological order.

Developing an H-index for OSS developers
Andrea Capiluppi, Alexander Serebrenik, Ahmmad Youssef
2012 9th IEEE Working Conference on Mining Software Repositories (MSR)
June 02-03, 2012 - Zurich, Switzerland.

Impact of Collaboration on Structural Software Quality
Ahmmad Youssef
2014 OpenSym Doctoral Symposium (Part I: Open Source Software)
August 27-29, 2014 - Berlin, Germany.

The impact of developer team sizes on the structural attributes of software
Ahmmad Youssef, Andrea Capiluppi
2015 International Workshop on Principles of Software Evolution
August 31-September 4, 2015 - Bergamo, Italy.

The data set extracted and studied as a part this research, along with analysis artefacts are available at the following address: https://github.com/ahmmadyoussefgithub/PhD

Table of contents

List of figures

List of tables

Glossary of Terms

Note that the definition of the terms below is offered in the context in which they are mentioned within this thesis.

Bonferroni correction: An adjustment made to p-values when several statistical hypothesis tests are simultaneously performed on a single data set.
Branch: A duplication of a source code directory structure within a VCS.
Class: The template that defines the behaviour and/or state of objects of its type.
Code clone: Sequences of duplicated source control residing across multiple VCS repositories.
Collinearity: Correlation between two or more independent variables in a regression model.
Commit: A set of changes made to the source code in a software repository.
Committer: An contributer who modifies source code in a software repository.
Confounding factors: Confounding factors are those that influence both the dependent and independent variables within a model causing an association to be made which may not be genuine.
Coupling: The degree to which components within software systems are interdependent.
Cohesion: The measure of the extent to which functionality within a single component belongs together.
Functional Complexity: Functional complexity which has no single definition but generally refers to the degree of sophistication in the logic encoded within a software system.
Database schema: The definition of the structure of the database, including its tables and the relationships between them.
Dependent variable: The variable that is subject to testing and measurement in an experiment .
External attributes: These are externally visible properties which manifest in how the software relates to its environment. Examples include maintainabilty and fault-proneness.
Fault-proneness: The extent to which software exhibits 'faults' which are structural imper-

fections which causes a system not to perform its required function.
FLOSS: Free/Libre Open Source Software (FLOSS) is developed by informal collaborative networks of programmers. Source code is openly shared to encourage others to build upon the software.
Forge: A collaborative platform designed to facilitate the creation of a community of developers to collaborate on the creation of software. Offers software development and management tools.
Fork: The process of creating an alternate and independent software development stream from an existing project.
Integrated Development Environment (IDE): An application that provides comprehensive features to support programmers in software development.
Independent variable: The variable that is controlled in an experiment to measure the effects on the dependent variable.
Inheritance: The hierarchical arrangement of classes such that a child class derives behaviour from its parent.
Internal attributes: These can be measured through direct observation of the software artefacts. Examples include structural properties such as coupling and cohesion.
Linear mixed models: A form of linear regression that allows for both fixed effects that apply to all groups and random effects that apply individually to subgroups within data sets.
Linear regression: A statistical model that attempts to establish a linear relationship between dependent and independent variables.
Maintainability: The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.
Mann-Whitney U test: Used to test the null hypothesis that two samples come from the same population or whether observations in one sample tend to be larger than observations in the other. **Method**: A programmed procedure defined in a class that is included in all its instances.
Modularity: The extent to which a systems functionality is logically partitioned into independent components.
Module: A logical grouping of related components making up part of a software application.
Multivariate: A statistical model using multiple variables to predict an outcome.
Normal distribution: A probability distribution symmetric about the mean. Observations are more frequent near the mean. When plotted, appears as a bell curve.
Object: An instance of a class.
Object Oriented: A programming paradigm based around 'objects' rather than 'actions'.
Outlier: An observation that is abnormally distant from other observations in a sample.

Principal Component Analysis: A technique to reduce dimensions in a data set to transform it to a number of linearly uncorrelated dimensions while retaining the maximum variance within the data set.
Probability distribution: A mathematical function providing probabilities of occurrence of different possible outcomes that a variable can assume.
p-value: The probability of finding the observed, or more extreme, results given a true null hypothesis.
Regression coefficients: In a linear model these are estimates of multipliers on independent variables.
Repository: A structured data store archiving files, their revision histories and other associated meta-data. Used to facilitate and manage change to source code.
Revision: A distinct changeset in a VCS repository.
Root Mean Square Estimate: A measure of the distance between observations and the values predicted by a regression model.
R-squared: A measure of how close the observations are to a regression line.
Scrum: A framework for managing agile development.
Software artefact: Tangible products of a software development process such as source code, compiled binary files or documentation.
Spearman correlation: A nonparametric measure of rank correlation rank-order assessing the relationship between two variables.
Stakeholders: People or groups affected by the outcome of a software development process.
Standard error: A measure of the typical distance between data points and a regression line.
Static code analysis: An analysis of software through direct inspection of its artefacts (particularly source code) without the execution of the software.
Structural complexity: The measure of the degree of interactions between components in a software system.
T-statistic: The ratio of the distance of the estimated value of a parameter from its regression line value to its standard error.
Univariate: A statistical model using a single variable to predict an outcome.
Variance: The expectation of the squared deviation of a random variable from its mean.
Version Control System: A system that enables and tracks changes to a file or set of files to enable recovery to previous revisions.

Chapter 1

Introduction

1.1 Research Context

There are many critical decisions that face software development practitioners throughout the development lifecycle, ultimately contributing to the success or failure of a project. These decisions, broadly, fall under the categories of Process, Technology, or People (Nasir and Sahibuddin, 2011). Decisions in the Process category are those such as the choice of development methodology (Chow and Cao, 2008; Vijayasarathy and Butler, 2016), development standards (Rainer and Hall, 2002), or the decision to invest in test automation to achieve shorter testing cycles (Lewis, 2016). Technology decisions can vary from questions of what hardware or programming languages to use for development to the selection of tools that development teams should employ (Scheer and Habermann, 2000; Ray, 2017; Chen et al., 2018; Eichhorn et al., 2018). Finally, the People category of decisions focuses on issues such as the resourcing and staffing of software development project teams and how those teams fit into the wider organisation (Krishnan, 1998; Andrejczuk et al., 2017; Alfayez et al., 2018).

Each of these categories of decisions has seen a great deal of academic research. In the Process category Kuhrmann et al. (Kuhrmann et al., 2015) conducted a mapping study of the field of software process improvement, finding 635 publications over the past 25 years. To give a flavour of these studies, some investigate how greater adherence to established software process models such as CMM (Paulk et al., 1993) or ISO (ISO 15504-5:2012, 2012) can result in better quality (Harter et al., 2012; Abrahamsson, 2013). Others are case studies into how organisations manage adoption of process models with refinements proposed for

particular contexts such as small or medium enterprises (Balla et al., 2001; Sulayman et al., 2012). Research in the Technology category is extremely broad covering topics such as the suitability of particular technologies for a given use (Sharp et al., 2003; Baker et al., 2006), the security implications of using a given technology stack (Mirheidari et al., 2012; Choukse et al., 2012; Gangwar et al., 2014), or studies of the tools that can be used to facilitate team communication in a global context (Portillo-Rodríguez et al., 2012). Finally, in the People category the research, again broad and diverse, can vary from a study of what motivates open-source software contributors through to the impact of team size or team diversity on team performance (Hoch et al., 2010; Von Krogh et al., 2012).

Of these three categories, 'People' decisions are established to have the greatest impact on development team productivity (Trendowicz and Münch, 2009) and critical success factors for software development projects are, by far, more likely to be in the realm of people factors (Boehm et al., 1978; Onoue et al., 2018). Software development is intrinsically a human activity and a more people-oriented approach is in the ascendency evident in, amongst other things, increasing adoption of agile methodologies (Pirzadeh, 2010).

Nasir et al. (Nasir and Sahibuddin, 2011) conducted an extensive literature survey of the critical success factors that impact software projects finding that in the People category some of the most cited project success factors, based on industrial case studies as well as surveys, relate specifically to the composition of the software development team (Schmidt et al., 2001; Sauer and Cuthbertson, 2003; Humphrey, 2005; Kappelman et al., 2006; Glass, 2006). This is pertinent because within most real world software development projects those 'people' decisions - for instance the composition of individual development teams - can be made by managers who are not particularly senior and, indeed, often with input from individual developers. By contrast, location strategies or project budgets are often dictated by senior management with little or no influence from those lower in the management chain and are therefore within the sphere of influence of significantly fewer practitioners. When considering decentralised volunteer-based Free Libre Open-Source (FLOSS) projects, it is also true that, beyond the composition of the software development team, there can be limited tools with which stakeholders can influence the success or failure of a project (Schweik et al., 2008).

In the context of this research, team size is a measure of the number of developers that modify a project source code. Within that same context team stability is taken as function of the cumulative time that each developer has worked with their fellow team members. Within the 'People' category of research there is an extensive body of work that empirically proves that

team size and stability are linked to specific external attributes of the produced software such as fault-proneness as well as more general aspects such as project success rates and team productivity. As will be detailed in the next chapter titled 'Related Work', earlier research established that smaller or more stable development teams are more productive, produce less fault-prone software, and have higher levels of stakeholder satisfaction. Much of this research has been motivated, at least in part, to help inform practitioners on how to compose their own development teams so that the associated risks can be limited as far as practicable, recognising that team composition is one of the few levers within the grasp of management to strongly influence project outcomes.

The research asserts that new insights can be gained into how team size and stability impact the produced software by measuring the internal attributes of the software instead of the more traditional approach of measuring its external attributes. Through this approach practitioners can form a more complete picture to inform decision-making. Uniquely, this also enables practitioners to measure and monitor key indicators, taking remedial action at earlier stages in the development lifecycle.

1.2 Team Size and Stability in the Literature

In 1974 Brooks, in his popular book 'The Mythical Man Month', stated that adding additional developers to a project can result in a loss of productivity due to the exponential difficulties involved in maintaining effective communication within a larger team (Brooks, 1986). In 2000, Raymond, in his book 'The Cathedral and the Bazaar', asserted that in open-source software development larger development teams are more effective at identifying and resolving bugs, leading to less fault-prone software. Raymond termed this 'Linus Law' named after the lead linux developer. In contrast to both Brooks and Linus law is the 'Core Team principle' that states that the size of a team should not have an impact on the success of the project as core development groups are always small. For researchers or practitioners, attempting to navigate these somewhat conflicting principles to understand which factors will win out is no easy task (Schweik et al., 2008).

Greater consensus is evident when reviewing the literature around team stability - a significantly smaller body of work - which agrees that more stable teams produce less fault-prone software. One particularly stark observation was that as team stability increased by 50% defects decreased by 19% (Huckman et al., 2009).

Questions of how development team size and stability impact fault-proneness is also very much at the forefront of practitioner minds. Many practitioners have taken the time to present their evidence and document their experience around these two particular strands in whitepapers and blog posts, generally agreeing with the academic research - some through empirical means (Macheronne, 2013; Mcconnell, 2018), others purely anecdotal (Miller, 2006; Erickson, 2012; Meccia, 2015; Plowman, 2015) - that smaller, more stable teams produce less fault-prone software. This is covered in more detail in the 'Related Work' chapter.

Interest in fault-proneness is not without good reason. There are numerous examples of software faults causing governmental or corporate institutions severe reputational damage. Knight Capital is oft-cited as an example of how costly fault-proneness can be after a defect in their order routing software caused a $465 million trading loss (SEC, 2013). Behind the headline grabbing incidents is a more pervasive issue throughout the industry. A 2013 Cambridge University study estimated that software bugs cost the global economy $312 billion annually (Britton et al., 2013). That same study found that developers spend half their time debugging software.

However, fault-proneness is not the only aspect that of concern to stakeholders. Maintainability has also seen significant research activity. This refers to the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment (Radatz et al., 1990). ISO 9126 states that maintainability is comprised of four sub-attributes - analysability, changeability, stability, and testability (ISO 9126-1:2001, 2001). Highly analysable software requires lower effort to investigate and understand sections of the codebase in order to remediate defects or to adapt the codebase to new requirements. Similarly, high levels of changeability require less effort to implement changes in the codebase. Stability implies a lower likelihood that making changes to the software may have unintended negative impacts. Finally, testability is a measure of the effort required to adequately test software. Taken together, a codebase exhibiting high-levels of each of these sub-attributes of maintainability support a more adaptable business against a backdrop of an oft-changing competitive landscape.

It is of crucial importance that researchers and practitioners alike understand the impact of any factors that can have a material impact on the maintainability of software. The focus of this thesis is to add evidence and insights on how development team size and team stability play a role as factors in the maintainability of produced software.

1.3 Internal and external factors in the literature

Figure 1.1 depicts, at a very high-level, the relationship that existing literature has established between the internal structural attributes of software and its maintainability.

Prior research has focused on establishing mathematical models that describe the impact of the internal attributes of software on its external attributes including maintainability. In these models the internal attributes are the independent variables while the external attributes are the dependent variables. Broadly, these models establish that lower coupling and complexity are more favourable structural properties, leading to lower fault proneness and greater maintainability. Conversely, higher cohesion and modularity are associated with that same favourable outcome of lower fault proneness and greater maintainability. Tables 2.2 and 2.5 neatly summarise these relationships which will be detailed in the next chapter titled 'Related Work'.

This work takes an alternative yet complementary approach to the existing body of research. The research questions in this thesis centre around establishing the impact of team composition on the internal attributes of software, essentially treating the team factors as the independent variables and the internal attributes as the dependent variables. Using the aforementioned models from existing research, these observations are subsequently used to deduce the likely impact of these team factors on maintainability. Given the breadth of the work modelling the impact of internal attributes on external attributes, this could be used to apply the observations in this research to other external attributes beyond maintainability.

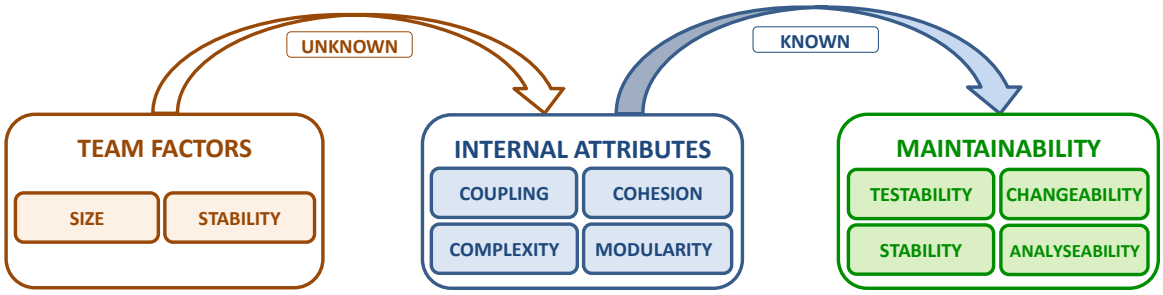


Fig. 1.1 The impact of 'people' factors on internal and external attributes of software

1.4 Research Problem and Approach

Effective teams are crucial to the success of organisations, especially in environments that require teams to be constantly created and dismantled as is the case in software development (Andrejczuk et al., 2017). When organisations are tasked with delivering business critical software, they are often faced with multiple options to resource the project. These options could include externally recruiting a number of developers from the market and forming a new team or alternatively seconding an existing stable team comprised of developers with prior experience of working together - either from within the organisation or from an external vendor.

After reviewing the existing research which does correlate team size and stability with fault-proneness it is notable that no insight is gained into how the internal attributes of software - such as coupling, cohesion, complexity and modularity - are impacted through these aspects of team composition. Given that the internal attributes of software essentially drive the aforementioned externally observable attributes (amongst others), it follows that, by all rights, this should be a crucial area of study, through which researchers can drive a deeper understanding of the impact of team composition on the aspects of stakeholder relevance such as fault-proneness and maintainability. The current state of research leaves academics and practitioners alike to draw their own conclusions on what changed internal attributes could be driving any externally observable attributes - and whether, for example, increased likelihood of fault-proneness could be observed at an earlier stage in the development lifecycle at the code level and subsequently mitigated. As Fenton rightly points out, practitioners are accustomed to measuring and monitoring internal attributes throughout the development process, and hence would be well placed to monitor and mitigate risks if they were broadly observable (Fenton and Bieman, 2014).

In order to qualitatively or quantitatively assess the negative effect of inappropriately sized or unstable teams, it is essential to analyse the impact that team size and stability have on the sub-attributes of maintainability. Such research is essential to providing practitioners with the requisite insights to inform their decisions around team composition. While existing research informs us that more stable development teams produce less fault-proneness, understanding the impact on the analysability, changeability, testability and stability of the software would enable practitioners to forecast how team stability would impact the maintainability phase of the project. This would empower practitioners to make team composition decisions that are more likely to be aligned with business goals and increase the likelihood of project success.

Table 1.1 Summary of the relationship between structural attributes and the externally observable attributes of software, as established in prior research.

	Trend Objective	Fault Proneness	Testability	Understand -ability	Change- ability	Stability
Coupling	↓	↓	↑	—	—	↑
Complexity	↓	↓	↑	↑	↑	↑
Cohesion	↓	↓	↑	—	↑	↑
Modularity	↓	↓	↑	—	—	↑

The empirical approach of this thesis is to measure the impact of team size and stability on the internal attributes of a software system that, in turn, have a proven impact on its maintainability. This is illustrated by Figure 1.1: by measuring the impact that these factors have on the internal attributes of software, this work provides a indirect link between the people factors and the maintainability of such a system. The primary contribution of this thesis is to add to the existing body of work and to add evidence in the form of trends, correlations and models describing the relationship between team size and team stability with the produced software's internal attributes, complementing the previously established trends relating internal and external attributes as summarised in Table 1.1 and discussed in detail in 'Related Work'.

This research draws upon a formally popular FLOSS 'forge' - a centralised online system with tools to enable distributed development teams to work together - to provide a data set which can be mined for observations to drive the empirical work in this thesis.

1.5 Research Questions and Hypotheses

Based on the survey of the related literature, the research questions focus on the two strands highlighted in the previously stated research problem. For each research question the null and alternative hypotheses are stated below.

- **RQ1** *What is the impact of development team size on the internal structural attributes of software projects and what are the implications on its maintainability?*

  - **H0,1** *Development team size does not impact the coupling, complexity, cohesion or modularity of the produced software.* Naturally the default starting position is to hypothesize that there is no relationship between team size and the structural attributes of software.

  - **H1,1.1** *Larger development teams produce software which exhibits greater coupling and complexity and lower cohesion and modularity when compared to that produced by smaller development teams.* Given existing research detailing the challenges that larger teams face in communication (Brooks, 1986) and given the body of empirical research that finds that larger teams produce more fault prone software (Weyuker et al., 2008; Nagappan et al., 2008; Meneely and Williams, 2009; Foucault et al., 2015), it follows that a reasonable hypothesis is that the internal structural attributes of the software produced by such a team will trend in a direction that is consistent with increasing fault-proneness; namely greater coupling and complexity, and lower cohesion and modularity. As both Linus law and the Core Team principle indicate the presence of forces that may ultimately work in favour of larger teams, such a hypothesis can only be proposed cautiously.

  - **H1,1.2** *Larger development teams will produce less maintainable software when compared to that produced by smaller development teams.* As discussed in 'Related Work' in detail, cohesion is correlated with testability (Badri et al., 2011) and analysability (Boehm et al., 1978) while coupling and complexity have been negatively correlated with stability (Elish and Rine, 2003). Given the previous hypothesis (H1,1.1) that larger teams will produce software which exhibits greater coupling and complexity and lower cohesion and modularity, the hypothesis follows that maintainability will likely deteriorate.

- **RQ2** *What is the impact of the development team stability on the internal structural metrics of coupling, cohesion, complexity, and modularity of software projects and what are the implications on its maintainability?*

  - **H0,2** *Development team stability does not impact the coupling, complexity, cohesion or modularity of the produced software.* Again, here the default starting position is to hypothesize that there is no relationship between team stability and the structural attributes of the software produced by that team.

  - **H1,2.1** *Less stable development teams produce software which exhibits greater coupling and complexity and lower cohesion and modularity when compared to*

more stable development teams. Similarly to larger development teams, existing research shows that less stable teams also produce more fault-prone software and provide lower client satisfaction levels (Huckman et al., 2009; Gardner et al., 2012). Naturally, this leads to a similar hypothesis to H1,1.1 that less stable teams will produce internal structural attributes which trend in a direction counter to the objective; namely greater coupling and complexity and lower cohesion and modularity.

– **H1,2.2** *Less stable development teams will produce less maintainable software when compared to more stable development teams.* Following a similar rationale to that expressed H1.1.2, given the hypothesis that less stable teams will produce software which exhibits greater coupling and complexity and lower cohesion and modularity, it follows that maintainability is hypothesised to deteriorate.

1.6 Research Goals and Objectives

The research questions and the related hypotheses are connected, in logical order, to the research goals. In the section below, each goal is formulated with its own rationale which is further elaborated on with a series of objectives, each justified by a rationale.

- **Goal 1** *To establish the impact of team size on the internal attributes of software and deduce the likely impact to maintainability.* This research goal is to conduct an analysis on the impact of team size on the structural metrics of software as a pathway to drawing insights into how this factor impacts the externally observable attributes of software. Within this overarching goal there are several objectives that facilitate a deeper knowledge of the underlying trends that impact structural metrics as precursor to formulating a credible methodology to execute the team size analysis.

  – **Objective 1,1** Observe structural metrics trends throughout the evolution of software projects. As a codebase undergoes development iterations, increasing in functional complexity and code volume, the progression of the structural metrics exhibit trends which are necessarily of significance to any further analysis.

  – **Objective 1,2** Control for confounding factors. These are factors that influence both the dependent and independent variables within a model causing a spurious association to be drawn. These factors can pose a significant threat to validity.

For this reason, this objective aims to devise and execute an analytical approach to control for these confounding factors in order to ascertain the impact of team size alone.

  - **Objective 1,3** Formulate a definition of the software development team which enables its size to be observed through mining software repositories and analyse structural metrics across a sample data set to observe the impact of team size on the structural attributes of software. This objective goes to the heart of answering the first research question - RQ1.

  - **Objective 1,4** Deduce the likely result that the impact from team size on the structural metrics on software will have on the four sub-attributes of maintainability; changeability complexity, testability and analysability. This is to be done by referencing the relationships established in prior research between the internal and the external attributes of software. Once the impact of development team size on the structural metrics of a codebase is observed, the focus shifts to deducing the impact that this will have on the external attributes of the software.

- **Goal 2** *To establish the impact of team stability on the internal attributes of software and deduce the likely impact to maintainability.*

  - **Objective 2,1** From the prior research identify the pitfalls that exist in mining software repositories, how they apply to team stability analysis, and how they can be mitigated. Two challenges exist when conducting network analysis in team stability analysis. The first is the effect that forking can have on the validity of results. Forking refers to the process of creating an alternate and independent software development stream from an existing project. As forked projects can retain the revision history of its parent, without proper identification and treatment, they can appear to be two independent projects with each set of authors contributing twice. The second challenge concerns tracking users throughout a forge - a task made complex by the fact that users often use subtly different identifiers through a project or while traversing a forge. These challenges should be met to ensure that they do not pose a significant threat to the validity of this research.

  - **Objective 2,2** Formulate a definition of the software development team stability and analyse structural metrics across a sample data set to observe the impact of team stability on the structural attributes of software. A nuanced approach is

necessary to distinguish between team stability accrued through the course of a project and that stability that comes from the team remaining stable through the course of multiple projects. This objective drives towards an answer to the second research question - RQ2.

– **Objective 2,3** Deduce the likely result that the impact from team stability on the structural metrics on software will have on the four sub-attributes of maintainability; changeability complexity, testability, analysability. Again, this is to be done by referencing the relationships established in prior research between the internal and the external attributes of software. Mirroring objective 1,4 concludes the answer to the second research question.

1.7 Thesis Contribution

Two main contributions to the state of the art can be identified within this thesis:

- **Advanced methodology to measure team size and stability:** This thesis presents an alternative approach to measuring the impact of team composition on external attributes by directly measuring the impact on its internal attributes and leveraging established research to deduce the ultimate impact on its external attributes. The impact of team size and stability on maintainability is studied through the GoogleCode forge and, in the process, numerous practical difficulties involved in mining a large and diverse forge are solved. In particular, this work identifies, quantifies and mitigates the previously undocumented and significant threat that forking can pose to the accuracy of forge network analysis.
- **Impact of team size and stability on internal structural attributes:** A clear relationship is established between team size and stability on the internal structural attributes of software. This research concludes that those projects developed by smaller or more stable teams exhibit lower levels of coupling and inheritance complexity and higher levels of cohesion and modularity. In addition to the observed trends, the state of the art is furthered through the proposal of two new measures to capture team stability, distinguishing between stability that accumulates as a team remains unchanged across projects and the stability which accumulates through the lifespan of an individual project through the collaboration of team members.

Table 1.2 A summary of the research questions, hypotheses, goals and objectives of this research.

Research Questions	Null hypothesis	Alternative hypothesis	Goal	Objectives
RQ1 What is the Impact of team size on the structural properties of software and its resultant maintainability?	**H0,1** Development team size does not impact the coupling, complexity, cohesion or modularity of the produced software.	**H1,1.1** Larger development teams produce software exhibiting higher coupling, higher complexity, lower cohesion and lower modularity. **H1,1.2** This leads to lower maintainability.	**G1** Establish correlations between team **size** and the structural attributes of FLOSS software and deduce the impact that these correlations will have on the externally observable attributes of the software.	**O1,1:** Observe how structural metrics progress as software projects evolve. **O1,2** Isolate and eliminate the confounding impact of functional complexity on the team size analysis. **O1,3:** Formulate a definition of the software development team size and analyse structural metrics the impact of this factor on the structural metrics. **O1,4:** Deduce the likely result of the impact of team size on the maintainability of software.
RQ2 What is the Impact of team stability on the structural properties of software and its resultant maintainability?	**H0,2** Development team stability does not impact the coupling, complexity, cohesion or modularity of the produced software.	**H1,2.1** Less stable development teams produce software exhibiting higher coupling, higher complexity, lower cohesion and lower modularity **H1,2.2** This leads to lower maintainability.	**G2** Establish correlations between team **stability** and the structural attributes of FLOSS software and deduce the impact that these correlations will have on the externally observable attributes of the software.	**O2,1:** Identify and mitigate the pitfalls associated with mining software repositories for the purposes of team stability analysis **O2,2:** Formulate a definition of the software development team stability and analyse structural metrics the impact of this factor on the structural metrics. **O2,3:** Deduce the likely result of the impact of team stability on the maintainability of software.

1.8 Intended Audience

This research is intended for both the research and practitioner communities. This thesis complements the existing body of research which correlates the internal attributes of software with observed external attributes by specifically studying the impact of team composition on these internal attributes. Researchers with an interest in relating software metrics to measures of stakeholder interest will find relevance in this work. It is also intended for this work to be of value to those practitioners in the field of software development. It is the intention of this thesis to contribute towards more informed practitioner decision-making around development team composition - particularly at the middle-management level. Furthermore, practically oriented observations of the impact of sub-optimal team composition, which can be monitored through static analysis of software, may find interest in the developer community.

1.9 Thesis Structure

The remainder of this thesis is arranged over five chapters as described below and illustrated in Figure 1.2.

Chapter 2. Related Work Prior research is documented in three distinct fields. Firstly, a survey is conducted for previous research that establishes correlations between the development team's size and stability against attributes of stakeholder importance such as fault-proneness and team productivity. Secondly, a survey is carried out in the established field of mining software repositories and a sampling of research that employs mining techniques to observe changes in the properties of software is discussed. Finally, software metrics are considered with a focus on structural metrics and how they are interpreted and correlated with externally observable attributes of software such as maintainability.

Chapter 3. Methodology In this chapter the methodological approach to this research is detailed. Existing mining tools are surveyed and the mining toolchain that underpins this research will be discussed in depth. Metrics suites are also surveyed and justification is provided for the choice of metric suite for this work.

Chapter 4. The Impact of Team Sizes on Structural Metrics This chapter focuses on answering the first research question by conducting a detailed analysis on a sample of

projects from the GoogleCode repository and establishing correlations between team size and the modularity, coupling, cohesion, and complexity of software.

Chapter 5. The Impact of Team Stability on Structural Metrics This chapter addresses the second research question with a focus on the impact of team stability accumulated through the lifespan of an individual project and across projects within a forge. To facilitate this work, network analysis is conducted across the entirety of the GoogleCode forge and, in the process, several threats to validity are identified and mitigated. The network analysis is used to identify the population of projects which is used to establish correlations between team stability and the modularity, coupling, cohesion, and complexity of software.

Chapter 6. Discussion The discussion chapter provides a summary of the results against the hypothesis, objectives and goals of this thesis. The results are analysed using individual projects as case studies to enable a qualitative analysis. Threats to internal and external validity and distilled conclusions are discussed. Finally, possible future avenues of research are proposed.

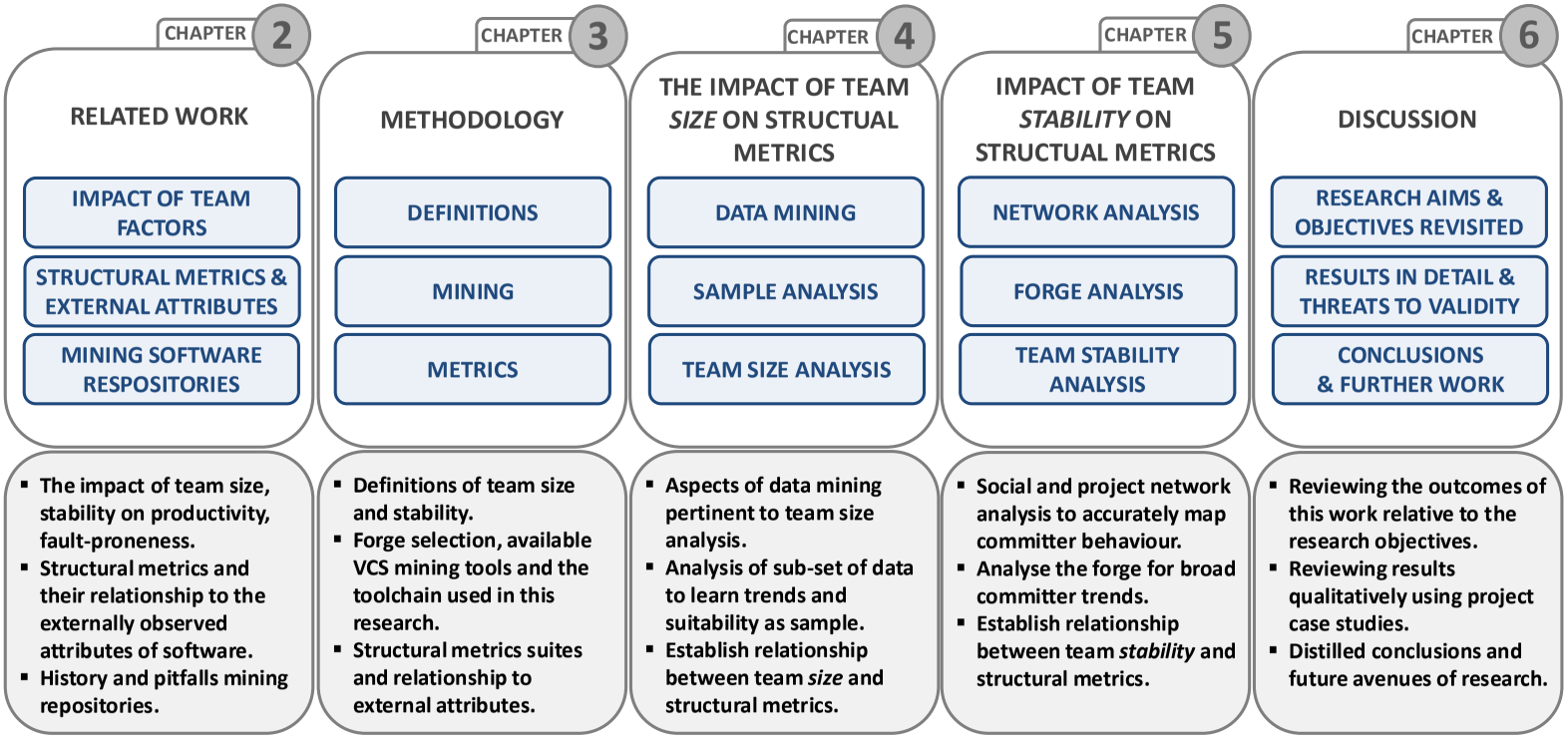


Fig. 1.2 Overview of chapter structure, colour-coded to distinguish mining from analysis.

Chapter 2

Related Work

2.1 Introduction

As outlined in the introduction, this thesis focuses on how software team size and stability impact the internal structural attributes of software. Out of this come three individual strands of related work which will be the focus of this chapter.

The first part of this chapter reviews the studies of the impact of people factors on the externally observable attributes of software, with a focus on factors of team size and stability. This work has the greatest direct relevance to this research and is one that this thesis endeavours to further by offering an alternative approach based on the direct measurement of the internal structural attributes of software rather than the observation of external attributes. The second strand of related work in this chapter comprises the body of research that establishes correlations between the internal structural metrics of software and its external attributes. This is of crucial relevance to this work as it is that very body of research that will later be relied upon to map observable trends of structural metrics onto conclusions that have meaning from the perspective of a non-technical stakeholder with a sole interest in the external attributes of the software. The third strand of related work concerns the practicalities of mining software repositories. A survey is provided of the available tools and a summary of the pertinent challenges and pitfalls associated with mining software repositories.

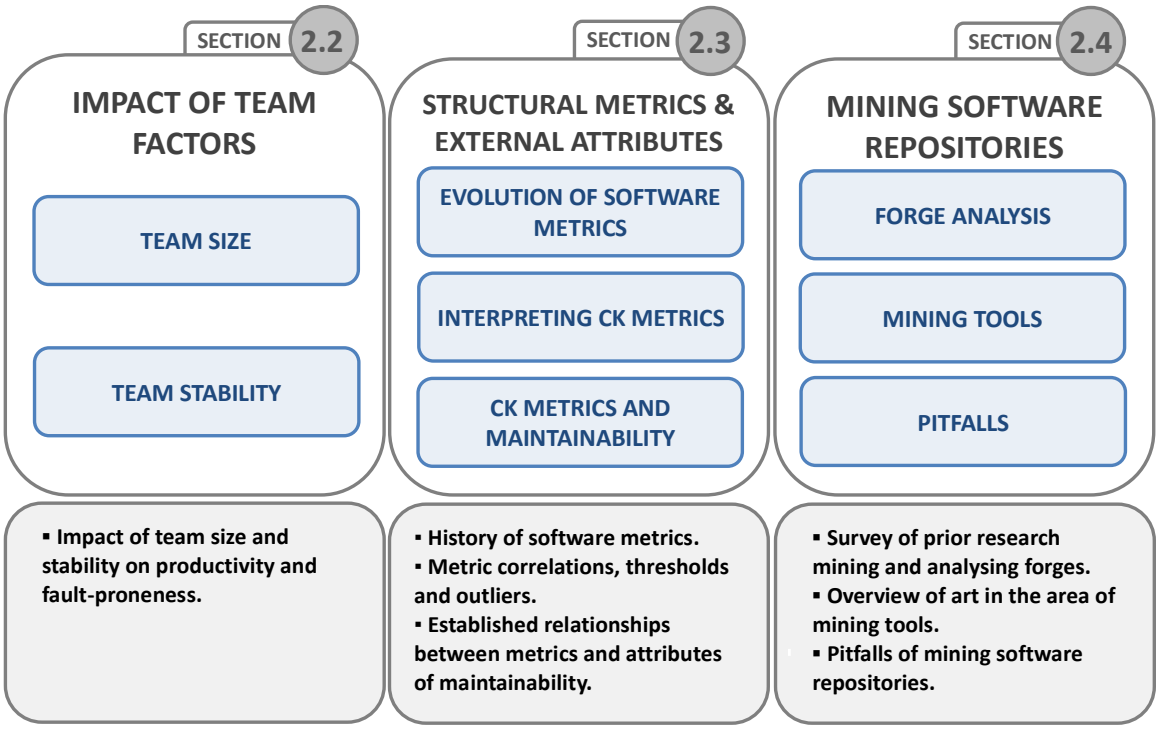


Fig. 2.1 Chapter 2 outline providing an overview of the contents of each section.

2.2 The Impact of Team Factors

There is a significant corpus of existing work concerned with establishing the impact of factors of team composition on the external attributes of software. This body of research models the relationship between developer and organisational factors with aspects such as fault-proneness and team productivity. This work provides the basis for the null hypotheses detailed in the previous chapter as well as the empirical approach of this study as we adopt and adapt previously established measures of team size and stability.

2.2.1 Team Size

There have been several empirical studies investigating the relationship between development team size and the team productivity. In his popular book 'The Mythical Man Month', Brooks argues that, since software development is a complex task, the communication effort is great and adding more developers can lengthen rather than shorten the time taken to complete a task as it adds an exponentially greater number of necessary communication paths between developers (Brooks, 1986) - although it is notable that Hsia et al. (Hsia et al., 1999) argue that

it won't necessarily take longer but will always increase the overall cost of delivery compared with correctly sizing the team from the outset. Roger et al., using data from 130 projects, empirically tested the impact of a number of factors on software development productivity concluding that larger team sizes significantly negatively impact software development time and productivity (Rodger et al., 2011). This is corroborated in other research and using a number of empirical methods (McLeod and MacDonell, 2011; Lalsing et al., 2012). Scholtes et al., using a data set of FLOSS projects, perform network analysis concluding that the magnitude of the productivity decrease is related to the growth dynamics of developer coordination networks (Scholtes et al., 2016). In contrast, Maillart and Sornette find that occasionally an OSS development team will exhibit 'superlinear productivity' in direct relation to the development team size, arguing that occasionally the whole is more than the sum of its parts (Maillart and Sornette, 2016).

Schweik et al. (Schweik et al., 2008) highlight the need to inform development managers who have a stake in FLOSS projects on whether increasing the development team size is more likely to result in a successful project and help avoid 'project abandonment', arguing that this is the primary tool at their disposal to influence outcomes. Rodriguez et al., acknowledging this impact on outcomes, seeks to advise managers on the ideal team size to facilitate a process of project decomposition and distribution of work amongst appropriately sized development teams (Rodriguez et al., 2012). The productivity of teams sized above and below an arbitrary threshold are compared, controlling for the functional complexity of the produced software. In-line with prior literature, it is noted that those teams sized below the threshold are more productive than the larger teams.

Pendharkar and Rodger investigated the relationship between team size and the associated cost of development (Pendharkar and Rodger, 2009). They observed that the team size does not linearly increase software development cost and that, in some cases (hypothesised to be those projects suffering communication inefficiencies), larger teams require a greater than proportional increase in resources. Blackburn et al. make similar observations while also noting that greater functional complexity leads to larger teams (Blackburn et al., 2006). This is intuitive given that larger teams have greater knowledge and expertise and therefore would typically be deployed to more complex problems. Hericko et al. worked to define the optimal team size given these two conflicting drivers, proposing a model to minimise development effort for a given project size (Heričko et al., 2008).

There has also been research negatively correlating team sizes to measures used as a proxy for software quality. Nagappan used data from Microsoft's Windows Vista project to establish

that metrics based on organisational structures (of which team sizes were one aspect) are a significant predictor of software fault-proneness (Nagappan et al., 2008). Nagappan's work was later validated by Caglayan et al. who found that, while organisational metrics were out-performed by pre-release metrics such as defect counts as a predictor of ultimate fault-proneness, they were a significant predictor nonetheless (Caglayan et al., 2015). Mockus also noted a correlation between team sizes and fault-proneness (Mockus, 2010). Bird et al. developed a more sophisticated code ownership model that distinguished between frequent 'major' committers and infrequent 'minor' committers and found that minor committers are more likely to introduce defects (Bird et al., 2011). Bell et al. observed that the number of developers that modify a file increased the probability of that file being defect prone (Bell et al., 2013). Recently, Chopra et al. and others have moved this research forward by building prediction models to identify fault-prone classes built upon a number of predictors including team size (Madeyski and Jureczko, 2015; Chopra et al., 2018).

2.2.2 Team stability

Team stability (in literature also referred to as 'team familiarity' or by the antonym 'team fluidity') is also viewed as a critical success factor for an effectively functioning and performing group wherever complex problems are tackled. From cardiac surgery teams to flight crews and basketball teams, those teams that experience continuity in personnel make-up are likely to be higher performing (Carthey et al., 2001; Akgün and Lynn, 2002; Yeh et al., 2005; Wiegmann et al., 2010; Huckman and Staats, 2013; Joshi et al., 2018). Software development teams are no exception (Bao et al., 2017). The Scrum Agile software development methodology, for example, favours avoiding changing team members for the stated reason that stable development teams are more productive (Deemer et al., 2010). There is anecdotal evidence to back this claim up with practitioners reporting that fluid teams are likely to be less productive as they tend to go through the 'Tuckman cycle' (Forming, Storming, Norming, Performing) with the addition of every new team member (Tuckman, 1965; Linders, 2011).

There has been comparatively few academic studies investigating the role of team stability within the field of software development. On the empirical side, Huckman et al. conducted a detailed study of the team stability and role experience on the output of development teams (Huckman et al., 2009). Armed with a data set of over a thousand projects and defining team success criteria in terms of software defect count and adherence to deadlines and budgets, they found that a conventional measure of experience - years of experience at a firm - was not linked with team performance. However, team stability (measured as the average number

of times that each member has worked with every other member of the team) was associated with less error-proneness and more budget adherence. One stark result was that, as familiarity increased by 50%, defects decreased by 19%, and deviations from budget decreased by 30%. This was confirmed by Gardner et al. who observed that teams with a high degree of team stability, captured by measuring the length of time that each team member had worked with their teammates, yielded a 10% increase in client satisfaction (Gardner et al., 2012). Mockus, studying a large commercial software project, observed that new developers were not associated with a decrease in quality (postulating that this was due to new developers being assigned peripheral tasks) while departures from the project were associated with greater fault-proneness (Mockus, 2010).

2.3 Structural Metrics and External Attributes

2.3.1 Overview

Underpinning the empirical approach to this research is the use of structural metrics to measure the internal attributes of software. While ISO/IEC 25010 recognises that internal quality drives external quality, it does not offer any specific direct measures for internal quality but instead offers a framework to define metrics that are influenced by internal quality. For example, maintainability is measured by the resources expended to modify software. These measures are neither direct nor predictive. Fortunately, there has been significant research in formulating such direct measures.

Coleman et al., in the early 90s, developed a maintainability model that used static measures of source code to produce a percentage figure on how easy a system is to support and change derived from a blend of measures including complexity metrics (Coleman et al., 1994). This metric is still used today within Visual Studio which classifies ranges which correspond to High, Moderate and Low maintainability (MSDN, 2015). Heitlager et al. from the Software Improvement Group (SIG) critiqued this model as presenting difficulties in re-constructing the root causes that drive a particular measure and suggested an alternative maintainability model which blends structural metrics including complexity, volume, and unit size to drive its metrics (Heitlager et al., 2007). The strength in the SIG maintainability model is the simplicity with which a calculated index would be mapped to its constituent structural attributes.

In a similar vein, the approach of this research to observing indicators of internal quality is to directly measure the internal structural attributes of a codebase and draw upon established models to ascertain if the trends observed indicate enhanced or degraded internal software quality.

There are two primary categories of internal attributes of software - size and structure. Software size is a broad term and measurements can vary from basic line of code counts through to function-point analysis. It is tempting to assume that size is directly correlated to external attributes of fault-proneness and development effort while negatively correlated to maintainability (Akiyama, 1971). However, as Fenton and Bieman state, experience shows that this is not a valid assumption and that the structural attributes of software play a vital part in driving these external attributes (Fenton and Bieman, 2014).

This section first discusses the metrics which can be used to capture structural attributes followed by a brief historical context. A survey of relevant software metric suites is provided alongside an overview of the research efforts to interpret metrics and modelling the impact of structural metrics on maintainability. The latter is particularly relevant as these relationships will be drawn upon later in this thesis to infer the likely impact of observations of structural metrics on the external attributes of the studied software systems.

2.3.2 What Are Structural Metrics?

Software metrics embody an empirical approach to software engineering and are primarily designed to assist in making assessments of software artefacts and development processes, in the process guiding engineers and project managers in their decision-making. A software metric is the quantitative measure of the degree to which a component, system, or process possesses a given characteristic or attribute (Ordonez and Haddad, 2008). If used appropriately, software metrics can lead to a significant reduction in costs of the overall implementation and maintenance of the final software product.

Structural metrics are a specific category of software metrics that allow us to measure and express the adherence of a codebase's structural attributes to sound engineering design principles. The key attributes in Object Oriented Programming are the interlinked concepts of coupling, cohesion, complexity, and modularity. These are explained as follows.

- **Coupling:** This is the degree to which components within software systems are interdependent. A high degree of coupling in a set of components implies that changes

in one component may impact the other components in that set (Parnas, 1972). At a relatively low level, modern object-oriented languages provide paradigms (such as interfaces) to facilitate lower coupling between classes. Interfaces allow a developer to separate the method of interaction into a component from the implementation of functionality. At a higher level, enterprise design patterns such as the Service-Oriented Architecture facilitates lower levels of coupling between applications (Jamshidi et al., 2018). Lower levels of coupling are desirable at each level of an enterprise software system (Chidamber et al., 1998; Pressman, 2005).

- **Cohesion:** Cohesion is a measure of the extent to which functionality within a single component belongs together. A component that exhibits a high degree of cohesiveness typically encapsulates a single set of highly related functionalities. In object-oriented languages, a high degree of cohesiveness is a desirable trait and is consistent with the 'single responsibility principle' which states that a class should have a single purpose and its functionality should be encapsulated within the class - i.e. not exposing its inner workings to other components (Fenton and Bieman, 2014). A high degree of cohesiveness and encapsulation is associated with a low level of coupling (Chidamber et al., 1998).

- **Complexity:** In the context of structural metrics, complexity refers to the structural complexity of software rather than the functional complexity. This is a contrast that will be discussed in the coming chapter in Section 4.4.3.2. Structural complexity is a measure of the degree of interactions between components in a software system (Fenton and Bieman, 2014). A highly complex system would contain a large number of components and a large number of interactions between the components. Although the number of interactions between components is a driver of structural complexity, the nature of the interaction between two components can introduce further structural complexity if there is a dependency on the inner workings of the components (i.e. poor encapsulation and high coupling). Although functional complexity can often introduce structural complexity, all things being equal, lower measures of structural complexity are desirable.

- **Modularity:** This refers to the extent to which a system's functionality is logically partitioned into independent components (Parnas, 1972). A high degree of modularity is desirable as it encourages low coupling and high cohesion which, in turn, reduces structural complexity (Parnas, 1972; Sullivan et al., 2001).

2.3.3 Evolution of Software Metrics

The study and application of software metrics dates back to the mid-1960's when the primitive Lines of Code metric was routinely used as the basis for measuring software development productivity (developer LoC per month) and quality (defects per KLoC). In 1971 Akiyama proposed the use of metrics for software quality prediction proposing a regression-based model for module defect density (number of defects per line of code) where line of code was used as a crude indicator of complexity (Akiyama, 1971). This was one of the earliest attempts, albeit a simplistic one, to extract an objective measure of software quality through the analysis of artefacts of a system. With the increasing diversity of programming languages, it became necessary to introduce a more sophisticated model of the structural attributes of software.

McCabe, recognizing the importance of testable and maintainable software systems, broke new ground in the area of software metrics introducing the first meaningful structural metrics (McCabe, 1976). In 1976, motivated by the observation that half the development time is spent in testing and that most of the cost of owning a system is in its maintenance, he developed a software metric which he termed 'cyclomatic complexity'. This metric is based on a formula to calculate the number of linearly independent paths through source code. Its purpose is to identify complex software modules based on program flow. Around the same period Halstead designed a structural metrics suite based on definitions of operators and operands modelling the complexity of individual lines of code (Halstead, 1977). To give a flavour of these metrics, the Halstead Difficulty uses a formula to assess the complexity based on the numbers of unique operators and operands capturing a measure of how difficult the code is to write and maintain. Halstead Effort is an estimate on the effort to rewrite a particular method.

The research community continued to be highly active in the field of structural metrics throughout the next decade (Côté et al., 1988) with advances in the usage of existing 'classical' metrics (Behrens, 1983; Gaffney Jr, 1981) as well as the formulation of new structural metrics (Boydston, 1984; Prather, 1984).

In the 90s, with the increasing adoption of Object-Oriented (OO) programming languages, the research in structural metrics took another significant step forward. Chidamber and Kemerer argued that Object-Orientation, as the most prominent advance in software development, and with yet to be established practices, necessitated measures that could guide organizations to its successful adoption. This fact, coupled with criticisms of existing metrics suites, saw

the development of the Chidamber and Kemerer (CK) metrics suite, detailed in Table 2.1 (Chidamber and Kemerer, 1991, 1994). For its popularity and simplicity, as detailed later in this chapter, this is the suite that will be used to underpin the empirical work in this thesis.

2.3.4 Survey of Metrics Suites

There are a number of structural metric suites that commonly appear in both academic literature and practitioner tools. In this section a comparative survey is provided for the most popular metric suites and justification is given to the choices in this research. The discussion is limited to object-oriented structural metrics given that, as will be discussed in the next chapter titled 'Methodology', this research studies Java software only. This is to the exclusion of suites such as the Halstead metrics. Furthermore, consideration is only given to those metric suites that comprehensively cover the key internal structural attributes - coupling, cohesion, complexity and modularity - to the exclusion of, for example, the McCabe metrics with their almost exclusive focus on structural complexity (McCabe, 1976; McCabe and Butler, 1989). For a fuller survey of the available object-oriented structural metric suites, the reader is encouraged to review the work of Gomathi and Linda Edith (Gomathi and Linda Edith, 2013) and Xenos et al. (Xenos et al., 2000). The survey is therefore limited to the Chidamber and Kemerer suite, the MOOD metric suite and the Lorenz and Kidd metric suite.

- **Chidamber and Kemerer metric suite** The Chidamber and Kemerer (CK) metric suite is one of the most cited of all structural metric suites within the academic community (Pressman, 2005) with a great deal of research spanning the two decades since its creation.

  The CK metric suite is designed to operate on the most fundamental unit in object-oriented systems - the class. The CK metric suite contains measures that capture complexity, coupling and cohesion. The values of the measures are fairly arbitrary and there has been research into defining the thresholds that could indicate classes which would be more likely to require remedial action (Rosenberg, 1998). While there has been some doubt cast on the theoretical and empirical validity of one of the measures in the suite (Fenton and Bieman, 2014), nonetheless, this remains the most validated OO metric suite available (Kitchenham, 2010) with a great deal of research successfully correlating the metrics to external quality attributes (Rosenberg, 1998; El Emam et al.,

Table 2.1 A summary of the CK metric suite. For further detail on the calculations and their theoretical basis refer to the research of Chidamber and Kemerer (Chidamber and Kemerer, 1994)

Metric	Attribute	Detail
Coupling Between Objects (CBO)	Coupling, Modularity	CBO is a count of the classes to which the class being inspected references. This metric is a measure of the number of other objects to which the class being considered is coupled. A high number can indicate poor encapsulation and lower modularity resulting in a low level of reusability.
Depth of Inheritance tree (DIT)	Complexity	DIT is calculated as the number of classes from that which is being measured to its top-level parent. This is a measure of design complexity, capturing the number of parent classes from which a class inherits. A high number may indicate excessive design complexity.
Lack of Cohesion of Methods (LCOM)	Cohesion	The LCOM is a count of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero. The degree of similarity for two methods m1 and m2 in a class is given by: $\text{LCOM} = \{v_1\} \cap \{v_2\}$ $\{v_1\}$ and $\{v_2\}$ are the sets of instance variables used by M_1 and M_2. This metric is a measure the dissimilarity of methods in a class via instanced variables. A high number can point towards poorly designed classes that do not adhere to the "single responsibility principle".
Number Of Children (NOC)	Reuse	NOC is the number of direct subclasses extending the class being measured. This metric is an indicator of reuse and abstraction. High numbers may indicate poor design or diluted abstraction.
Response For a Class (RFC)	Complexity	RFC is the number of methods within a class added to the number of methods invoked by any of those methods. This is a measure of the count of methods which may be executed in response to a message. High numbers may highlight objects with undue complexity.
Weighted Methods per Class (WMC)	Complexity	WMC is calculated as the number of methods in the class where each method complexity is considered to be 'unity' or equal to 1. This metric is the sum of the complexity of the methods of a class and is an indicator of the complexity of a class through its method count. A high number can indicate undue complexity and limited scope for re-use.

2001; Basili and Perricone, 1984; Subramanyam and Krishnan, 2003). This suite is covered in more detail later in this chapter.

- **MOOD metric suite** The MOOD metric suite was developed by Abreu and Carapuça in 1994 to provide system-level measures (as opposed to class-level measures) for object-oriented systems in order to guide and assess OO design quality (Abreu and Carapuça, 1994). The measures capture a broad range of structural attributes including encapsulation and polymorphism factors - factors absent from the CK metrics suite. In contrast to CK metrics where the values of the measures are arbitrary, the MOOD metric values are probabilities with values from 0 to 1 representing the likelihood of the existence of a particular attribute. Harrison et al. conducted research detailing the utility of MOOD metrics to practitioners finding that they present information that would be of general use to software managers to understand the overall attributes of a system (Harrison et al., 1998b). MOOD metrics are not widely adopted in academic research nor industry and this is reflected in the very scarce availability of tools that generate these metrics with Project Anaylzer (Abounader and Lamb, 1997) being the only distributed tool that the author could find. This tool only had support for the Visual Basic programming language. In its favour there has been significant research analysis which largely validated the MOOD metric suite (Abounader and Lamb, 1997; Harrison et al., 1998a).

- **Lorenz and Kidd metric suite** In their book 'Object-Oriented Software Metrics' Lorenz and Kidd proposed a metric suite consisting of eleven metrics that, in a similar fashion to the CK metric suite, measured attributes at a class level (Lorenz and Kidd, 1994). Metrics are broadly in four categories - size, inheritance, class internals (attributes that can be measured on a class in isolation such as cohesion) and class externals (attributes that capture how a class interacts with other classes such as coupling and reuse). Lorenz and Kidd also propose threshold values to help interpret metric observations. The Lorenz and Kidd suite did experience a degree of recognition in academic circles with a large number of citations (Nesi and Querci, 1998) but, with few validation studies (Sharma et al., 2012) and no available tools to measure these metrics, it is fair to say that this metric suite has not experienced significant academic or practitioner adoption. This may be due to the fact that the metric suite is fairly basic and constitutes directly measurable attributes such as Number of Methods, Number of Public Variables and Number of Variables which has caused some doubt to be cast on its usefulness (Harrison et al., 1998b).

Table 2.2 A summary of the Rosenberg OO metrics guidelines.

Metrics	Objective	Testing Efforts	Understand-ability	Maintain-ability	Dev. Effort	Reuse
Complexity	↓	↓	↑	↑		
Size (LOC)	↓	↓	↑	↑		
Comment %	↓	↓	↑	↑	↓	
CBO	↓	↓	↑	↑		↑
LCOM	↓		↑	↑	↓	↑
RFC	↓	↓				↑
WMC	↓			↑	↓	↑

2.3.5 Interpreting CK metric values

At a time when OO metrics were a relatively new field of study, Rosenberg proposed that metrics without interpretation guidelines are of little value. She concluded that, although some numeric thresholds were suggested by developers, there was little to justify specific values (Rosenberg, 1998). She proceeded to harness experiences within the NASA Software Assurance Technology Centre (SATC) to apply a common sense approach to the formulation of interpretation guidelines of individual OO metrics including most of the CK suite. These findings are summarized in Table 2.2. The table shows the objective - the direction of trends associated with favourable outcomes - and the associated impact on the external attributes. For instance, it was concluded that developers should attempt to attain low values of LCOM (the objective), which will result in a higher degree of understandability, maintainability and reuse, while reducing development effort.

However pertinent and useful, Rosenberg's research was based on the knowledge and experience within SATC, and was not an empirical treatment of software metrics. Shatnawi moved this area of metrics research forward by establishing CK metric threshold values at a number of risk levels representing probabilities of error proneness (Shatnawi, 2010). Oliveira et al. worked to devise a technique to establish relative thresholds across a corpus of 79 projects programmed in Pharo and Smalltalk, identifying those projects that violated thresholds for a higher percentage of metric observations in order to find projects which would be expected to exhibit lower maintainability (Oliveira et al., 2015). Hussain et al. used logistic regression to identify thresholds above which classes exhibit greater fault-proneness (Hussain et al., 2016).

Chidamber et al. researched the question of the interpretation of the CK metric suite for managerial use and concluded that 'outlier' metric values indicate a level of complexity that would require management action (Chidamber et al., 1998). Chidamber et al. continued to suggest that a useful method to identify such 'outlier classes' is by applying Pareto's 80/20 principle and selecting classes which exhibit metric values from the 80th percentile for further attention - for example assigning a higher skilled developer to that implementation or assigning extra testing resources to that component.

Basili et al, motivated by the objective to leverage structural metrics to provide guidance to the areas of a system where testing efforts are best spent, established the utility of the Chidamber and Kemerer suite as a predicator of fault-prone software classes (Basili et al., 1996). This was achieved by assembling eight software development teams and using regression analysis to establish relationships between OO metrics and observed defects.

These are, by no means, the only studies of this nature. Subramanyam and Krishnan conducted similar work with access to a large number of in-house developed codebases, controlling for programming language and software size, confirming the results obtained by Basili et al (Subramanyam and Krishnan, 2003). These results were further validated in a number of similar studies, each adding its own unique contribution (El Emam et al., 1999; Tang et al., 1999; Cartwright and Shepperd, 2000; El Emam et al., 2001; Subramanyam and Krishnan, 2003; Gyimothy et al., 2005; Xu et al., 2008; Malhotra and Jain, 2012; Okutan and Yıldız, 2014; Song et al., 2018). Table 2.3 surveys the empirical approach within this research.

Saberwal et al. developed logistic regression models relating CK metrics to bad code smells driven by the desire to guide refactoring efforts to where they are most needed (Saberwal et al., 2013). More recently, this work was validated by Tufano et al. (Tufano et al., 2017) using linear regression models. Badri et al, using similar techniques, concluded that a correlation exists between LCOM and unit test coverage, validating the use of OO metrics as a predictor of the testability of classes (Badri et al., 2011).

2.3.6 CK metrics and Maintainability

Santos et al and Ernst independently identified a number of issues with threshold values, foremost among them that these values make generalised statements across projects (Santos et al., 2017; Ernst, 2018). Structural metric values depend heavily on complexity and size, and

Table 2.3 A survey of the research modelling fault-proneness as the dependent variable and CK metrics as the independent variables.

Study	Confounding Variable(s)	Data Set	Analysis	Results
Basili et al, 1996	None factored in	8 student projects written in C++	Univariate linear regression. Multivariate logistic regression	Finds that LCOM is not a significant predictor of fault-proneness while the remainder of the CK metrics are.
Tang et al., 1999	None factored in	3 small/medium commercial systems written in C++	Logistic regression	RFC and WMC strong predictors for fault-proneness
Emam et al., 1999	Class size (LOC)	1 medium-sized commercial project	Logistic regression	After controlling for size, only CBO was an indicator of fault-proneness
Cartwright and Shepperd, 2000	None factored in	One large commercial project	Linear regression	Found the inheriting classes were more defect prone (identified as classes having a DIT or NOC > 0)
Subramanyam and Krishnan, 2003	None factored in	1 large commercial project written in C++ and Java	Linear regression	CBO, DIT, WMC predictive of fault-proneness
Ferenc et al., 2005	None factored in	1 large C++ FLOSS project	Linear regression, decision trees and neural networks.	CBO and LOC predictive of fault-proneness.
Xu et al., 2008	Class size (LOC)	1 medium-sized government project written in C++	Neural networks	CBO, RFC and WMC are reliable metrics for defect estimation finding that overall
Malhotra and Jain, 2012	None factored in	1 medium/large-sized FLOSS project written in Java	Logistic regression and machine learning techniques	Machine learning models comparable in performance to linear models. Found that CBO, LCOM, RFC and WMC not to be significant predictors of fault-proneness. The rest of the CK metrics were indicators.
Okutan and Yıldız, 2014	None factored in	9 open-source Java projects.	Bayesian networks	RFC is the most reliable predictor of fault-proneness. LCOM and WMC are less effective while NOC and DIT have limited effect.
Song et al, 2018	None factored in	106 publicly available data sets.	Machine learning algorithms	CK metrics, used alongside network and process metrics, were found to enable defect prediction.

therefore a single threshold value will not necessarily hold true across a diverse set of projects. Moreover, establishing a threshold across a corpus of projects using the techniques devised by Oliveira et al. is arguably of limited value to this research given the objective of establishing inferences from metric trends on how software maintainability is generally impacted by team factors. For this reason, this section focuses on surveying the research that empirically establishes a relationship between CK metric values and externally observable attributes of software in order to expand upon (and detail the empirical validation of) Rosenberg's interpretation of the relationship between CK metrics and maintainability.

As established in the previous chapter, maintainability is comprised of four sub-attributes - analysability, changeability, stability, and testability. Correia et al, through a survey-based study present the opinion of software quality experts that a number of structural attributes drive the sub-attributes of maintainability (Correia et al., 2009), the consensus being that size, complexity, coupling, cohesion and test quality are all key factors. Chong and Lee developed a technique to visualise the structural attributes of codebases using a weighted complex network in order to capture its structural characteristics, with respect to its maintainability and reliability (Chong and Lee, 2015). They broadly observe that high coupling and low cohesion are associated with lower maintainability, confirming the consensus of the software quality experts. Li and Henry conducted one of the first studies to determine if CK metrics could be used as a predictor of maintenance effort, concluding that DIT, LCOM, NOC, RFC and WMC all predict maintenance efforts beyond what can be predicted for size alone (Li and Henry, 1993).

The next section surveys research modelling the relationship between CK metrics on particular sub-attributes of maintainability and each of the sub-attributes of maintainability.

2.3.7 CK metrics and the Sub-Attributes of Maintainability

Bruntink and van Deursen used correlation analysis to study the relationship between CK metrics and the testability of software using a data set of five projects (including one open-source project) (Bruntink and van Deursen, 2006). Using the lines of test code and the number of test cases in the unit tests as a proxy for testability, they find that DIT, LCOM and NOC are predictors of testability. It is attributed by the authors to be due to the developers choosing not to re-test inherited behaviour from the parent class within each child class. Badri et al. furthered this research by testing a series of metrics capturing the structural attribute of cohesion for correlation (of which LCOM was one) against testability (Badri

Table 2.4 A reproduction of Boehm's software understandability model.

	Understandability				
	Very Low	**Low**	**Nominal**	**High**	**Very High**
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling	Strong modularity, information hiding in data/control structures
Application Clarity	No match between program and application world views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self-Descriptiveness	Obscure code; documentation missing, obscure or obsolete	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentations.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well-organized, with design rationale.

et al., 2011). Confirming the results of Bruntink and van Deursen, LCOM was found to be a significant predictor of testability.

Harrison et al. used a similar statistical approach to confirm a negative correlation between WMC and the time to create automated tests for software (Harrison et al., 1998b).

Harrison et al. also broadened the scope of their research to cover understandability and changeability. To measure software understandability, a model formulated by Boehm et al. is used which rates software qualitatively on its structure, application clarity and self-descriptiveness (Boehm et al., 1978). A simplified version of this model is replicated in Table 2.4. Harrison et al. measure the time to implement modifications as a proxy to measuring changeability. WMC was found to be negatively correlated with understandability. Both WMC and LCOM were negatively correlated with changeability.

Elish and Rine conducted a study to determine if CK metrics could be used as a predictor of the stability of software (Elish and Rine, 2003). Their research calculated the class-level stability through an algorithm that determined the likelihood that the class would be change-prone as a result of a class-level change elsewhere in the design.

CBO, DIT, LCOM, RFC, and WMC were all found to be negatively correlated with stability. In particular CBO and RFC were strong predictors of stability. A high CBO indicates that a

Table 2.5 A summary of established associations between CK metrics with the sub-attributes of maintainability.

		Sub-attributes of Maintainability			
Structural Attribute	**Rosenberg Objective**	**Testability**	**Understand-ability**	**Change-ability**	**Stability**
CBO	↓				↑
DIT	↓	↑			↑
LCOM	↓	↑		↑	↑
NOC	↓	↑			
RFC	↓				↑
WMC	↓	↑	↑	↑	↑

class depends on many other classes or that many other classes depend on it, increasing the likelihood that change ripples through to the high CBO class. Similarly, a class with a high RFC indicates a higher number of internal and external methods that may impose change on the class.

Tables 2.5 and 2.6 provide a summary of this survey. This will be drawn upon later in this thesis to draw insights from observations on structural metrics trends in the context of their impact on maintainability.

2.4 Mining Software Repositories

Mining Software Repositories is a term that refers to the extraction, inspection, and analysis of artefacts produced through the software development process in order to deduce useful information about software projects. The intention is often to make this information available both to researchers to build upon and to industry practitioners to better inform decision-making (Hassan, 2008).

This section reviews the related work in four areas. The general motivations for mining software respistories is first outlined. A survey is provided of the prior research mining and analysing forges - those centralised platforms such as SourceForge, GitHub and GoogleCode which provide the ecosystem to facilitate distributed development. Then a survey of the tools that have been developed to support the activity of mining software repositories is presented and evaluated in the context of its applicability to this research. Finally, the challenges and pitfalls associated with mining individual project repositories or entire forges are then

Table 2.6 A survey of the research establishing associations between CK metrics with the sub-attributes of maintainability. No confounding factors are controlled for.

Study	Dependent Variable(s)	Data Set	Analysis	Results
Li and Henry, 1993	Maintainability (measured by LOC changed)	Two commercial systems written in Classic-Ada	Linear Regression	Concludes that NOC, LCOM, RFC, WMC, DIT all predict maintenance efforts beyond what can be predicted for size alone.
Harrison et al., 1998	Size (LOC), testability (time to create automated tests), changeability (time to implement modifications), understandability (Boehm measures)	Five small projects written in C++	Correlation analysis	No results on DIT and NOC as no inheritance in data set. Negative correlation between WMC and the time to create automated tests for software. WMC was found to be negatively correlated with understandability. Both WMC and LCOM were negatively correlated with changeability.
Elish and Rine, 2003	Stability	Three medium-sized FLOSS projects written in Java	Correlation analysis	CBO, DIT, LCOM, RFC, and WMC (particularly CBO and RFC) were all found to be negatively correlated with stability.
Bruntink et al., 2006	Testability	Five medium/large-sized projects written in Java	Correlation analysis	Using the lines of test code and the number of test cases in the unit tests as a proxy for testability, they find that only DIT and NOC are predictors of testability.
Badri et al., 2011	Testability	Two medium-sized FLOSS projects written in Java	Correlation analysis and logistic regression	Found a correlation between LCOM and unit test coverage, validating the use of OO metrics as a predictor of the testability of classes
Saberwal et al., 2013	Bad code smells	One medium-sized FLOSS project written in Java	Logistic regression	RFC, LCOM, NOC and WMC found to be useful predictors of bad code smells.
Tufano et al., 2017	Bad code smells	Two hundred FLOSS projects	Linear regression	RFC, LCOM and WMC found to be predictors of code smells.

discussed with an emphasis on aspects of network analysis and accurately determining authorship.

2.4.1 Overview

A software repository is built on a Version Control System (VCS), such as CVS (CVS, 2018) or GIT (GIT, 2018), which is used to manage change in source code. These repositories come with a great deal of data that can be mined and subsequently analysed. Each act of file creation, deletion, or edit is represented within a 'commit'. Meta-data associated with every commit lists the paths of files that have been modified, committer details and a date. Snapshots of the source files themselves can be retrieved in their present state or at any point in their history - typically either for manual qualitative analysis or more often by machine-driven quantitative analysis. By mining software repositories, the evolution of software can be observed.

Researchers typically inspect particular characteristics of a system throughout its evolution and observe trends and relationships. Kagdi et al. classify these types of studies into one of two categories (Kagdi et al., 2007). The first category of investigations observes the changes in properties through multiple versions of a system - for example, defect density or software complexity (Yamashita et al., 2017; Agrawal et al., 2018; Tian et al., 2018). The second category of investigation is more interested in the mechanics of the changes in artefacts - for example who is revising source files, how often, and the drivers of change (Lee et al., 2017; Ortu et al., 2018). This research is focused on mining software repositories to study changes in properties.

Studying aspects of software engineering, and indeed social science, through the mining and subsequent analysis of data from FLOSS repositories is a well-trodden path with a large volume of academic studies leveraging this approach (Hassan, 2008; Hemmati et al., 2013). There are a number of reasons why this approach is widely adopted:

- **Depth:** There are several million FLOSS projects available in the public domain - with thousands being added on a daily basis. Although a large number of projects never reach maturity (Comino et al., 2005), this is still an extremely rich resource to mine (Deshpande and Riehle, 2008).
- **Access:** Under the GNU license under which FLOSS projects are typically distributed, no restrictions apply to extracting, analysing, and publishing data derived from the

publicly available source code or associated meta-data (GNU, 2007). In contrast, access to source code for proprietary commercial software is often restricted to the relevant in-house development team only for security, compliance and competitive reasons.

- **Rich data:** Open-source forges such GitHub (GITHub, 2018) or GoogleCode (GoogleCode, 2018) come with a great deal of data that can be mined and analysed. In addition to the VCS commit data, there is also project level information that is made available including project categorisation, activity, artefacts, and bug reports. The FlossMole project was established to extract, normalise and publish project-level meta-data available across forges in unified format (Howison et al., 2009). Some of their artefacts are used in this research.

2.4.2 Forges

There have been a number of studies where a multitude of repositories have been mined within a broader forge. These studies have typically focussed on analysing project artefacts to study the impact of developers, the forge, or to otherwise facilitate the process of FLOSS adoption.

Eilhard and Ménière conducted an empirical study of 10,533 projects on SourceForge assessing the productivity of development team members, finding that volunteers tend to score lower than corporate developers (Eilhard and Ménière, 2009). Similarly corporate developers are found to benefit more from 'knowledge spillover' - the positive exchange of information between individuals within an organisation. Capiluppi and Beecher carried out a comparative analysis between two large forges - Debian and SourceForge - to assess whether the decay of software architecture is impacted by the forge (Capiluppi and Beecher, 2009). While Debian was found to host more complex projects, it also exhibited greater 'anti-regressive' work to reduce this complexity over time. Capiluppi et al. also attempted to identify whether the forge could have an impact on the structural metrics of the developed software (Capiluppi et al., 2009). They find no significant difference between metrics on the KDE forge (which specifically reinforces coding standards) compared with SourceForge (which does not do so).

Bagnato et al. state that assessing if a FLOSS project meets the requisite standards for business adoption is a non-trivial task and requires analysis of multiple project artefacts including source code, documentation and issue trackers (Bagnato et al., 2017). To support

this process, they developed an IDE plug-in called CrossMiner which extracts information from these various data sources for a given project and presents it within a single screen. In a similar vein, Wasserman et al. developed a methodology and a tool to assess the 'business suitability' of FLOSS projects by, again, mining these data sources and rating projects on a series of criteria including functionality, documentation and adoption (Wasserman et al., 2017). More recently Tamburri et al., also motivated by the need to provide greater rigour around the process of FLOSS adoption developed a tool called 'Yoshi' to analyse the open-source community and categorise it into one of a number of known organisational patterns (Tamburri et al., 2018). Applying their analysis to 25 projects from GitHub, they assert that they find value in measuring and monitoring these key organisational aspects.

2.4.3 Mining Tools

With the advent of open-source repositories, researchers acquired access to an large and rich data set. This led to an improvement in the tooling used to mine repositories. German (German, 2004) documented the challenges involved in mining the CVS repositories of the GNOME project. German et al. (German et al., 2005) followed this up with a review of some tools the mine repositories and suggested a comparison framework to support this activity. The available tools were generally found to be disparate and fulfil the relatively narrow requirements of the research groups that developed them. More recently Tiwari et al. proposed an 'app store' model for packaging and distributing software repository mining tools utilising their platform 'Candoia' (Tiwari et al., 2017). While this platform has not seen significant adoption, the concept is undoubtedly a step forward. In the next chapter a review is provided of the tools of direct relevance to this work and the potential for employing these tools is evaluated in the context of the data extraction and analysis requirements of this research.

2.4.4 Pitfalls

As will be discussed during the course of this thesis, when mining a significant amount of repository data, there are a number of pitfalls that, where ignored, can constitute a serious threat to the validity of the research. This is an interesting stream of research in its own right and there are a number of studies that typically focus on either a particular VCS or particular challenges that exist across VCS. This is covered in the next sub-section titled 'Data

Extraction and Entity Reconciliation'. Through the course of this research the pitfalls around conducting network analysis in a forge containing forked projects has been a particularly significant challenge and will emerge as dominant theme later in this thesis. Prior research in this area is documented in the sub-section titled 'Forking and Cloning'.

2.4.3.1 Data Extraction and Entity Reconciliation

German did some early work in the field of mining software repositories and raised practical concerns with the volumes of data involved in mining a single significant CVS repository, proposing a graphical tool to help visualise large data sets (eventually becoming the SoftChange tool) (German, 2004). Bird et al. conducted a similar study against the GIT version control system bringing out some its particular idiosyncrasies, particularly around the pervasive nature of branch development and the implications that this has on how to interpret revision history (Bird et al., 2009).

Also relevant to this work is the research that focuses on the pitfalls associated with mining data from forges. Iqbal et al., attempting to solve for the challenge of integrating data across multiple FLOSS code forges propose the use of so-called 'semantic web' technologies to represent the meta-data contained therein (Iqbal et al., 2012). They argue that linking the various developer aliases across forges can produce a holistic picture of their activity, unlocking in the process some useful analytics. To do so they propose an email similarity algorithm which is not fundamentally dissimilar to that used within this research and described in Section 5.2.1 (Iqbal, 2015). Goeminne and Mens also highlight the challenges in reliably establishing committer identities across repositories, conducting a survey of a variety of algorithms available to help mitigate this (Goeminne and Mens, 2013). Their work was built upon by Xiong et al. who employed Natural Language Processing to perform identity reconciliation across GitHub and Stackoverflow, the popular developer community-based knowledge base (Xiong et al., 2017; GITHub, 2018; StackOverflow, 2018). In a similar vein, Squire tackled the specific problem of identifying projects that reoccur across forges and presented a method to score similarity between project pairs (Squire, 2009).

Howison and Crowston documented the promises and perils of mining SourceForge, documenting the practical challenges of data extraction, data analysis and research design (Howison and Crowston, 2004). More recently Kalliamvakou et al. conducted a similar study against GitHub finding that many projects were inactive or have very few commits and most repositories were for individual development (Kalliamvakou et al., 2014). Some of their work informs the forge and repository analysis documented in Chapters 3 and 4 in an effort

to understand the extent that the trends observed in SourceForge and GitHub apply to the forge that is the subject of this thesis: GoogleCode.

2.4.3.2 Forking and Cloning

Forking refers to the process of creating an alternate and independent software development stream from an existing project, often retaining the original VCS revision history of the parent. When mining data from projects within open-source forges it is crucial to reliably establish the unique commit activity on a given project and the fact that the forking process duplicates the revision history across multiple projects introduces an avenue of potential distortion to the raw results. When conducting social network analysis, it is essential that each committer's contribution is accurately and reliably identified without threat of misreporting child project engagement when there was only engagement in the parent project. This is particularly true in the case of this research where a particular form team stability is measured by observing the number of previous projects that committers have partnered in. This will be covered in greater detail in Chapter 5.

Nyman and Mikkonen conducted research to establish the most common motivations for forking within SourceForge (Nyman and Mikkonen, 2011). The methodology to identify forked projects was to execute a keyword search within project descriptions to find references to forking. Although this approach suffices when attempting to locate a statistically significant sample to study, relying on developers to specifically declare a project as 'forked' in the description does not help us identify the full set of forked projects.

Robles et al. (Robles et al., 2006) suggested a fairly manual approach for locating significant software forks that involved searching Wikipedia using the term 'software fork' and manually navigating to the project homepage to extract key information ahead of a study on the motivations and outcomes of forking. This is an adequate approach when attempting to extract a sample of forked projects for further study but cannot be applied to the large-scale mining of software forks within open-source forges. More recently, Jiang et al conducted similar research against the GitHub forge and they were able to directly use project meta-data made available by the forge (a recent innovation) to identify forks (Jiang et al., 2017).

As part of the process of maintaining a forked project, it is often desirable or indeed necessary to import changes from the master project. Ray et al. developed a tool called REPERTOIRE to automate the identification of common commits between known forked projects through comparison of source files but it does not attempt to identify forked projects in a wider open-source forge (Ray et al., 2012). This is the most sophisticated approach to detecting forks that is noted in the prior literature. In this general research field there have also been

a number of efforts to automate the identification of 'cloned code' i.e. source code that is duplicated within a single project or across projects within a forge. This field of research has drawn some attention due to the potential negative impact that code cloning has on maintainability (Lozano et al., 2007). Lozano and Wermelinger developed a prototype tool called 'CloneTracker' and applied it to the study of changeability within software containing clones (Lozano and Wermelinger, 2008). Clones within a wider forge could indicate the presence of forking, making it valuable input for a heuristic that identifies such projects. Gharehyazie et al. developed a tool called Clone Huntress which identifies code clones in GitHub with the motivation of simplifying 'code foraging' - the process of discovering code within a broader repository that may be of relevance to the 'forager' (Gharehyazie et al., 2018).

Schwarz et al. (Schwarz et al., 2012) developed a set of lightweight techniques based on hashing algorithms to identify cloned code in a way that, in theory, could scale up to an entire forge. Lee et al. (Lee et al., 2010) developed similar techniques to support instant cloned code researches (albeit designed to work within a single repository only) based on a more sophisticated multi-dimensional indexing algorithm. This is a particularly active research area with recent efforts to identify code clones using an array of novel techniques including image processing and machine learning (Ghofrani et al., 2017; Ragkhitwetsagul et al., 2018).

This research builds upon earlier work by developing a framework that employs heuristics, including mining version control histories and project meta-data with the objective of identifying forks to enable accurate network analysis across a large forge.

2.5 Chapter Review

This chapter provided a literature review in three fields that are directly related to this thesis. First, the impact of the development teams size and stability on key project attributes was considered. The work of Nagappan et al., Mockus, and Caglayan et al. was discussed, finding a correlation between team size and fault-proneness (Nagappan et al., 2008; Mockus, 2010; Caglayan et al., 2015). Similarly, the research of Huckman et al. and Gardner et al. was covered, establishing a relationship between team stability and fault-proneness as well as overall client satisfaction (Huckman et al., 2009; Gardner et al., 2012). The second strand of research that was discussed was the impact of CK metrics on the external attributes of software, particularly the sub-attributes of maintainability; testability, stability, changeability,

and stability. Tables 2.3 and 2.6 documented the empirical approaches to the prior research along with the key findings. Finally, this chapter provided a review of most relevant research in the field of mining software repositories focusing on FLOSS forges previously mined and the pitfalls associated with these mining efforts.

The next chapter covers the methodological approach to this research covering team size and stability definitions, the forge mining toolchain, and the process of selecting a metrics suite, crucial to the empirical measurement of structural attributes.

Chapter 3

Methodology

3.1 Introduction

This chapter is divided into three sub-sections documenting the methodological approach to each of the key challenges in observing the impact of team factors through mining software repositories. As reflected in Figure 3.1 outlining the structure of this chapter, these challenges are broadly in the categories of Definitions, Mining and Metrics. In order to begin to model the relationship between team factors and the internal attributes of FLOSS software, it is first necessary to define team size and stability and establish an empirical approach to their measurement. This is the topic of the first section in this chapter. The second section is concerned with the identification of an appropriately rich data set to study which, fortunately, open-source repositories provide with few technical limitations. The criteria of selecting a forge to mine will be articulated and a mechanism to mine this data and extract information in a consistent, reliable and repeatable way is described. In its final section, this chapter details the approach to measuring the pertinent internal structural attributes of the code through the evolution of the project.

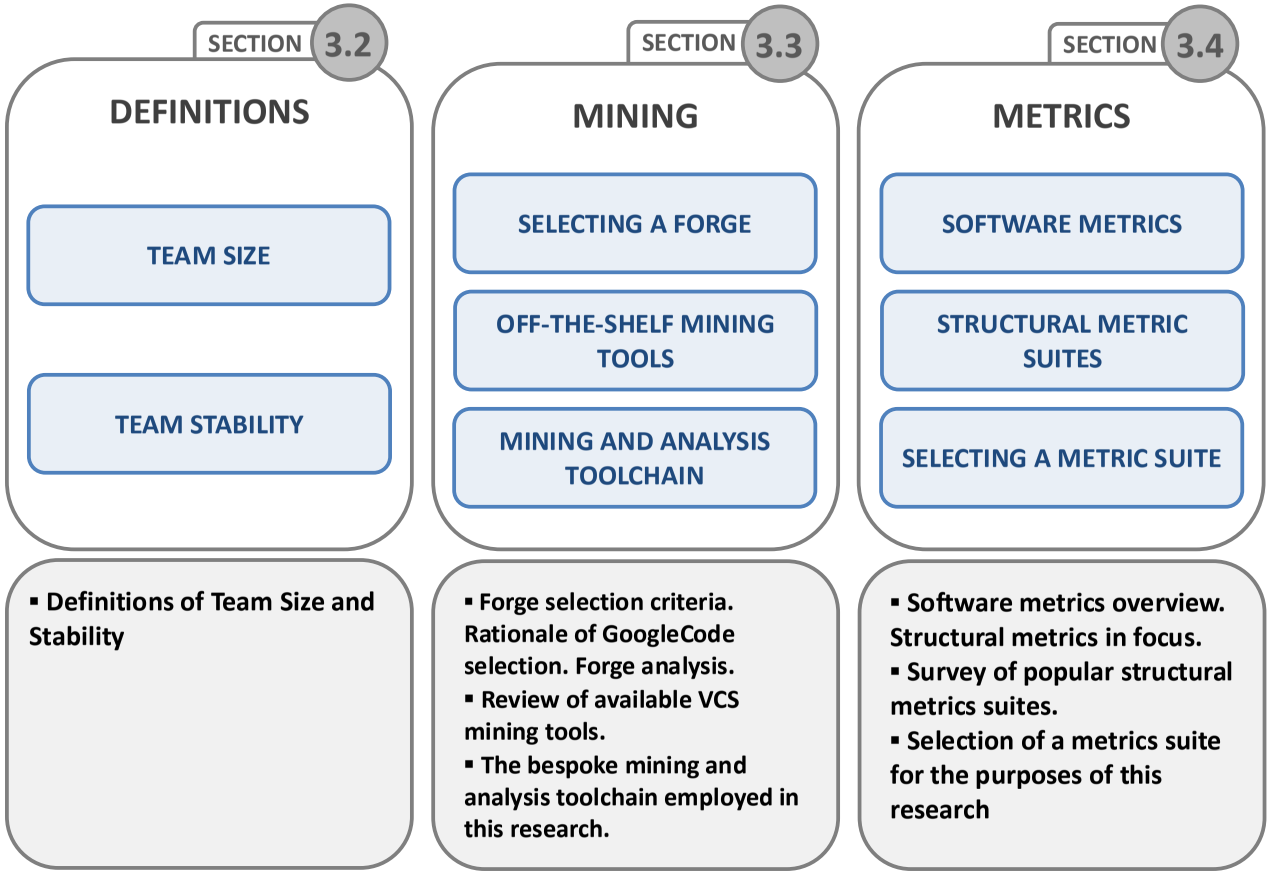


Fig. 3.1 Chapter 3 outline providing an overview of the contents of each section.

3.2 Definitions

3.2.1 Development team size

There are a number of possible definitions of a software development team. Both Capra and Wasserman and Smith et al. consider a team to consist of all developers to have worked on a codebase for any length of time (Smith et al., 2001; Capra and Wasserman, 2008) while Nagappan et al. (Nagappan et al., 2008) consider the development team to also include management, administration and operations personnel. Given the context of mining open-source repositories, there is a preference to define team size as the cumulative total of all unique committers present in the revision history in the version control system of a given project. This is the definition that is adopted by this research as it is consistent with the prior literature, simple to measure and reproduce, and elegantly enables the representation of the plurality of the unique development design approaches that may have influenced the evolution of a codebase. Independently, Rempel and Mader developed the same approach to measuring development team size through their work studying quality factors in FLOSS repositories (Rempel and Mäder, 2017).

There are some potential limitations to this approach, most notably that we do not distinguish between frequent committers and causal (infrequent) committers. As will be discussed later in this thesis, while it is true that the majority of commit activity takes place by a minority of committers, nonetheless the majority of committers do make a significant contribution and cannot be discounted, hence this is not regarded as a threat to validity. Similarly, when defining the team, the time during which developers contributed to the project is not considered. While it could be considered counter-intuitive that developers making contributions without any time overlap be considered part of a single team, analysis of the data sample shows that this is by far the exception in the GoogleCode forge, with the vast majority of developers making contributions in overlapping time windows with their fellow team members.

3.2.2 Development team stability

There has been relatively little empirical work that investigates the stability of development teams and its impact on aspects of the software development process. This could be due to the fact that it is a non-trivial task to capture a measure of stability. In the prior literature, team

stability is also referred to using the terms team fluidity or team familiarity (Huckman et al., 2009) and care is taken to distinguish this concept from 'team tenure'- i.e. the cumulative programming experience of the individuals on the team (Hackman, 2002). To measure team stability a similar approach to Huckman et al. (Huckman et al., 2009) is adopted, defining this as the cumulative time that each member has worked with every other member of the team. This approach is consistent with the limited prior research in this field and offers a simple, easily understood measure. Chapter 5 expands on this definition and proposes a methodology to calculate stability as it accrues within a project team.

3.3 Mining

3.3.1 Selecting a metrics suite

The research questions in this thesis mandate a number of key criteria of the metric suite chosen for this work. The first main requirement is that the suite is comprehensive in capturing the key attributes in Object Oriented Programming, described earlier in this chapter, of coupling, cohesion, complexity, and modularity. The second key criterion, necessary for the validity of this research, is that the metric suite be credible and empirically validated in an existing body of research. This means that the metrics must be proven reliable indicators of external quality attributes.

For this work to maintain relevance to its intended audience, which encompasses both researchers and practitioners, it is helpful to employ a metrics suite that can be well understood by practitioners, with individual metrics directly mapping onto the internal attributes. Furthermore, the chosen metric suite is that it should have good tool support as eases the development of a toolchain, enhances the repeatability of this work, and also serves as a gauge of its popularity amongst researchers and practitioners alike.

The matrix in Table 3.1 below details how each of the three metric suites considered match up to this criteria. The CK metrics suite is the only suite that is an adequate match - particularly with respect to empirical validation, relevance to practitioners and tool availability. For these reasons the CK metrics suite forms the basis for the empirical approach to this research.

Table 3.1 A comparison of the three metric suites considered for this research against the stated criteria.

Criteria	CK Metrics	MOOD Metrics	Lornez & Kidd Metrics
Empirically validated	**Yes** – the most cited OO metric suite in academic research with plenty of work correlating values to external attributes of fault-proneness, maintainability, testability, and more (Kitchenham, 2010).	**Limited** – A fraction (less than 5%) of the citations of the CK metric suite despite being published in the same year. Limited empirical validation work from Baroni et al (Baroni et al., 2003).	**Limited** – Nesi et al. did conduct some validation (Nesi et al., 1998) but overall there is a lack of empirical validation to support the use of this suite (Sharma et al., 2012).
Relevance to practitioners	**Yes** – metrics are simple and capture clearly understood design attributes.	**Limited** – system-wide measures appeal to project managers but lose a level of granularity that would be of interest to developers relative to CK Metrics (Harrison et al., 1998b).	**Limited** – As the metrics are fairly basic, they require a degree of further analysis before they are meaningful to practitioners.
Tool availability	**Yes** – Plenty of tools are available to practitioners to measure and monitor these metrics.	**No** – very few tools calculate MOOD metrics and they are not suitable for Java codebases (Abounader and Lamb, 1997).	**No –** No available tools that measure these metrics.

Table 3.2 Popularity of languages in top 3 open-source software forges (reproduced from Redmonk (O'Grady, 2011))

Rank	Github	SourceForge	GoogleCode
1	Ruby	C++	Java
2	Python	Java	XML
3	Javascript	C	C++
4	C++	XML	Python
5	C	Python	PHP
6	XML	PHP	Javascript
7	Java	Javascript	C
8	PHP	Shell	C#
9	Perl	C#	Shell
10	Shell	Perl	SQL
11	C#	Ruby	Perl
12	SQL	SQL	Ruby
13	Assembler	Assembler	Assembler

3.3.2 Selecting a forge to mine

There are a number of popular open-source project hosts or 'forges'. According to research in 2011 (at the outset of this research's data collation activities) in terms of the number of commits, GITHub was the most popular, followed by SourceForge and then GoogleCode (O'Grady, 2011). Each of these forges attracts has a unique and varied make-up of languages making up its project population - see Table 3.2.

The practicalities of constructing a toolchain that is capable of mining data from a VCS and subsequently conducting static code analysis to extract the pertinent internal attributes of the software meant that it was going to require significant additional effort to accommodate more than one programming language. Given that one of the aims of this work is to maintain the relevance of this work to both the research and practitioner communities, it was logical to select a programming language with a high degree of popularity. For this reason, the Java was chosen as it consistently rates as the programming language with the highest adoption rates (TIOBE, 2017). This also has the fortunate consequence of meaning that in most large forges there are a large number of Java projects available for study. Furthermore, from a static code analysis tooling perspective, Java is very well supported.

GoogleCode was the open-source forge selected for its popularity and high level of Java adoption rates. Another strong advantage in favour of GoogleCode was that they allowed project administrators to choose from among three available version control systems - Subversion, GIT, and Mercurial - on which to host their source code meaning that the mining toolchain would be sophisticated enough for others to reuse to mine GitHub (which uses GIT as its underlying VCS) or SourceForge (which uses Subversion and Mercurial). This is helpful as it means that the toolchain developed for this research can have utility across a broad set of forges. This is particularly pertinent given that GoogleCode announced that it was shutting down their service and entering 'archival mode' - meaning that any future research on active projects will not happen on the GoogleCode forge.

At this point it is also worth noting that, since the selection of the repository to study, the landscape has significantly changed and new repositories such as Assembla (Assembla, 2018), and Gitlab (GITLab, 2018) have established a strong presence - both of which use GIT as the underlying VCS - helpfully ensuring that the toolchain remains current.

3.3.3 Overview of the GoogleCode forge

As of May 2012, GoogleCode hosted 236,787 projects of which a significant number has seen sustained developer activity. GoogleCode projects are assigned 'labels' by the project administrators to assist in categorising the project. Many projects are 'forks' from popular projects - often where developers choose to take the project evolution in a slightly different direction or choose to 'experiment' on the codebase in their own sandbox. Forked projects retain the revision history of the original project which creates a challenge in ensuring that a forked project with a handful of commits isn't mistaken for the more popular parent project. As this is a particularly difficult challenge for which a unique solution was devised, a section is devoted to this topic later in this chapter.

The GoogleCode repository was started on the 27th of July 2006 (Shankland, 2006) and the decision was announced nine years after its inception for its shut down (DiBona, 2015). This offers a unique opportunity to observe a forge throughout its lifespan. As we study commit patterns, it is notable in Figure 3.2 that the cumulative number of commits across all projects grows from the low hundreds at the forge initiation to a peak of over 10K commits per day. It is also notable that the committer activity begins to steadily tail off towards the end of 2010 as projects migrate to more popular repositories such as GitHub. In the first half of 2011 the commit activity on GitHub surpassed GoogleCode, SourceForge and Microsoft

Codeplex combined (RedMonk, 2011). This shift led industry commentators to observe that in 2011 GitHub had become the major centre of gravity within the open-source space. This is likely attributable to the collaborative nature of its offering which pushed the boundaries of 'social coding' by providing transparency on contributors activity and allowing them further their skills and manage their reputation (Dabbish et al., 2012). As GitHub continued its exponential growth over the subsequent years, GoogleCode did not compete in a meaningful way by enhancing or rebooting its offering. In this context, it is easy to rationalise Google's decision to shut down the repository, closing the forge to new project creation in 2015.

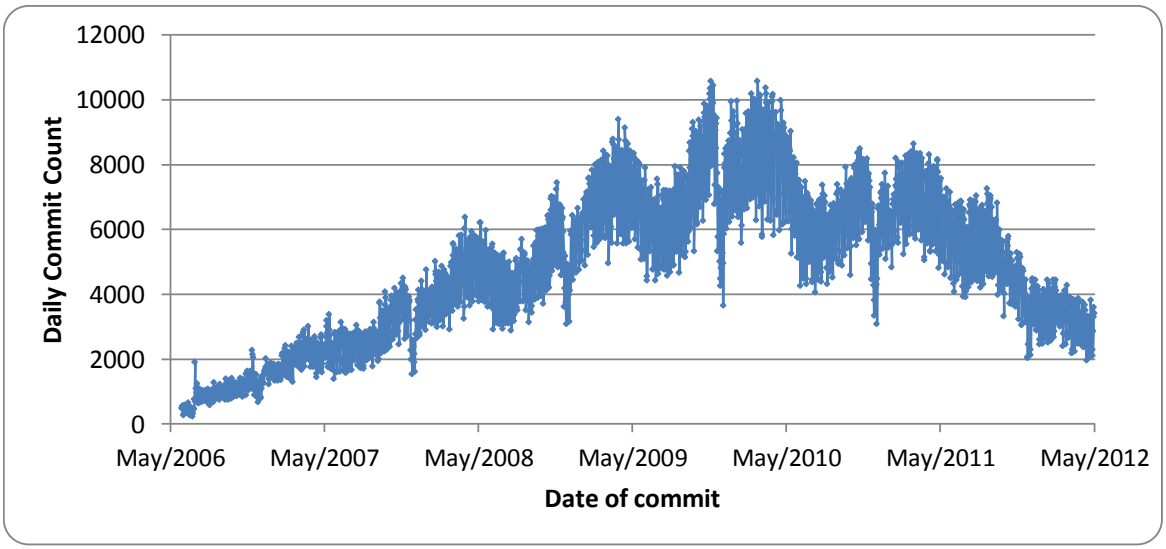


Fig. 3.2 Commit histogram showing daily activity levels across the entirety GoogleCode.

In the data analysis and visualisations throughout this thesis a fixed upper bound is defined for the time period of study. This was necessary as mining such large volumes of data can take multiple weeks to successfully extract. The majority of this data mining effort took place in the years 2012-2013 and for this reason, it was chosen to only study data up to the end of May 2012. This date was also chosen as it coincides with the availability of then up-to-date FLOSSMole artefacts which, as outlined later in this chapter, facilitates the data mining effort.

Figure 3.2 gives an indication of the activity trends but doesn't give an insight into whether this is down to an increase in the number of projects or whether this is attributable to greater activity on existing projects. Figure 3.3 helps establish that this commit activity is, indeed, visually correlated with an increasing number of projects. Clearly observable is an initial spike of project creation on the go-live date of GoogleCode. In October 2011, a significant drop in project creation is evident, decreasing to an average rate of 3 projects per day. There

is no immediately obvious reason in the publicly available records to explain this drop but it is clear that, even prior to this drop, there was a general downwards trend.

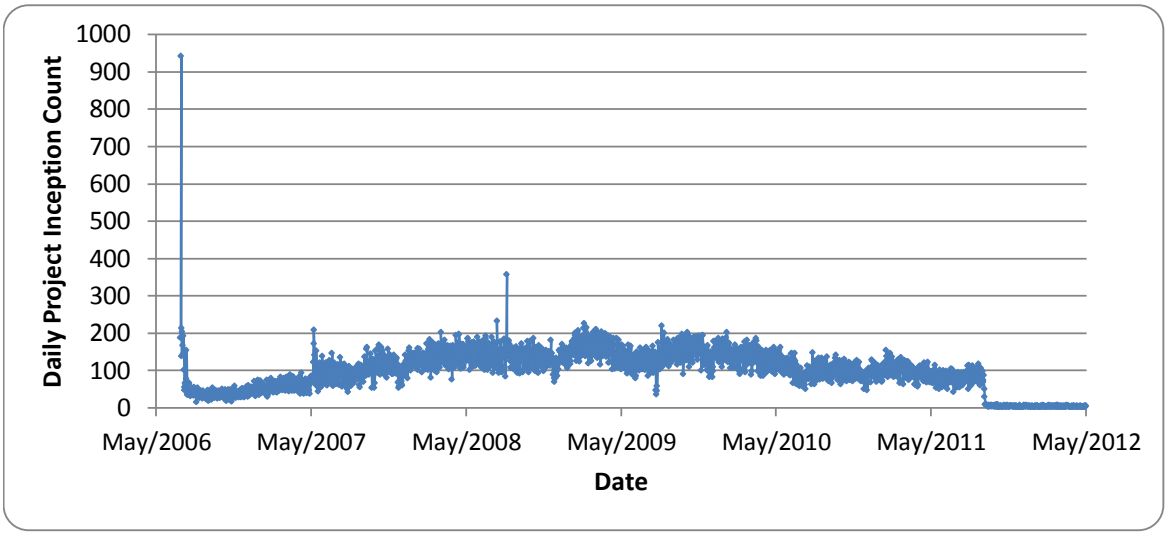


Fig. 3.3 A histogram depicting the daily rate of new project creation.

From a total project count of 236,787 only 95,490 have a single commit (corresponding to the initial project creation). Squire postulates that this is due to 'bots' creating projects although the motive for this activity remains unexplained (Squire, 2017).

3.3.4 Toolchain requirements

This research presented a number of challenges to mine the commit details for all the projects in GoogleCode's repositories and consolidate this information within a single data model and, ultimately, in a relational database for further analysis.

Given that this research tackles the study of team stability, it is necessary to mine the entirety of the chosen repository in order to derive team stability analytics based on the number of times each committer has worked with every other committer. This research clearly does not stop at an analysis of commit data. Given the focus on the internal attributes of software it is essential to inspect the source files for each project chosen for study, from which structural metrics can be extracted. This data, when juxtaposed with the relevant commit data, allows the trends surrounding structural metrics to manifest as projects evolve and mature. Given the sheer size of GoogleCode this is no small task and it will involve hundreds of gigabytes

of data transfer, multiple gigabytes of storage and many days of processing to produce an accurate representation of individual contributor behaviour.

The requirements of the toolchain can be articulated as follows.

- **Mine VCS logs and software metrics:** Mine VCS revision history and conduct static source code analysis against each snapshot of source code.
- **Breadth and scale:** Capability of mining VCS revision history of thousands of repositories across SVN, Mercurial and GIT.
- **Queriable data sets:** Store revision history in normalised queriable form.
- **Joinable data sets:** Join VCS revision history data with software metrics mined from snapshots of source code.

3.3.5 Open-Source Tools

Table 3.3 provides an overview of several relevant open-source mining tools - Softchange (German, 2004), Hipikat (Čubranić and Murphy, 2003), Dynamine (Livshits and Zimmermann, 2005), Kenyon (Bevan et al., 2005) and CVSAnaly (Robles, 2004) - along with a brief summary of their limitations with regards to this research.

Of these tools CVSAnaly was considered the more versatile to mine, normalise, and store revision history. The main functional limitations associated with CVSAnaly in the context of this research are as follows.

- **Missing support for the Mercurial VCS:** This is one of the GoogleCode supported VCS systems alongside GIT and Subversion - both of which are supported by CVSAnaly.
- **Missing support for Static Code Analysis:** Although there was no out-of-the-box support for Java static code analysis, there is support for 'extensions'. For this approach to be successful, such an extension would need to check-out each version of the codebase, execute static code analysis, and commit the results to a version of the CVSAnaly schema.
- **Restricted Database Schema:** Integrating static code analysis would necessitate further building out the CVSAnaly schema to store the mined metrics

Table 3.3 A comparison of a number of version control mining tools

Tool	Summary	Limitations
Softchange (German, 2004)	Extracts historical data from CVS repositories and defect trackers and joins both sources of information. It can also do static analysis of C++ and Java source code.	Only capable of extracting data from CVS. Source code analysis is syntactic in nature only. No support for structural metrics mining.
Hipikat (Cubranic and Murphy, 2003)	Similar to Softchange in that it is designed to join multiple sources of open source project data. Supports CVS, Bugzilla, Newsgroups, and mailing list archives.	Only capable of extracting data from CVS. No static source code analysis.
Dynamine (Livshits and Zimmermann, 2005)	This tool correlates repository revision histories with snapshots of source code to identify common code change patterns.	Language and VCS independent but fairly limited in that it is designed with specific use cases in mind – namely to identify common commit patterns across individual repositories.
Kenyon (Bevan et al., 2005)	Re-usable framework to extract data from multiple VCS systems and store it to disk in a customisable file format.	No static source code analysis. No out-of-the-box support for database persistence.
CVSAnaly (Robles, 2004)	Extracts information from revision history logs, storing in a database. Supports multiple VCS systems. Active research community.	No out-of-the-box support for static Java source code analysis.

Consideration was given to committing to enhancing CVSAnaly with the aforementioned functionality - an undertaking that would require that the CVSAnaly codebase be sufficiently understood in order to correctly identify the integration points for various new functionalities. Given the complexity of this task along with the limited existing functional scope, the balance ultimately tipped in favour of creating a bespoke toolchain for data mining.

3.3.6 Toolchain

This section documents the toolchain developed to mine and analyse the GoogleCode forge. Particular attention is devoted to those components that are common to both team size and team stability analysis: those components that are particular to either type of analysis are discussed in Chapters 4 and 5 respectively.

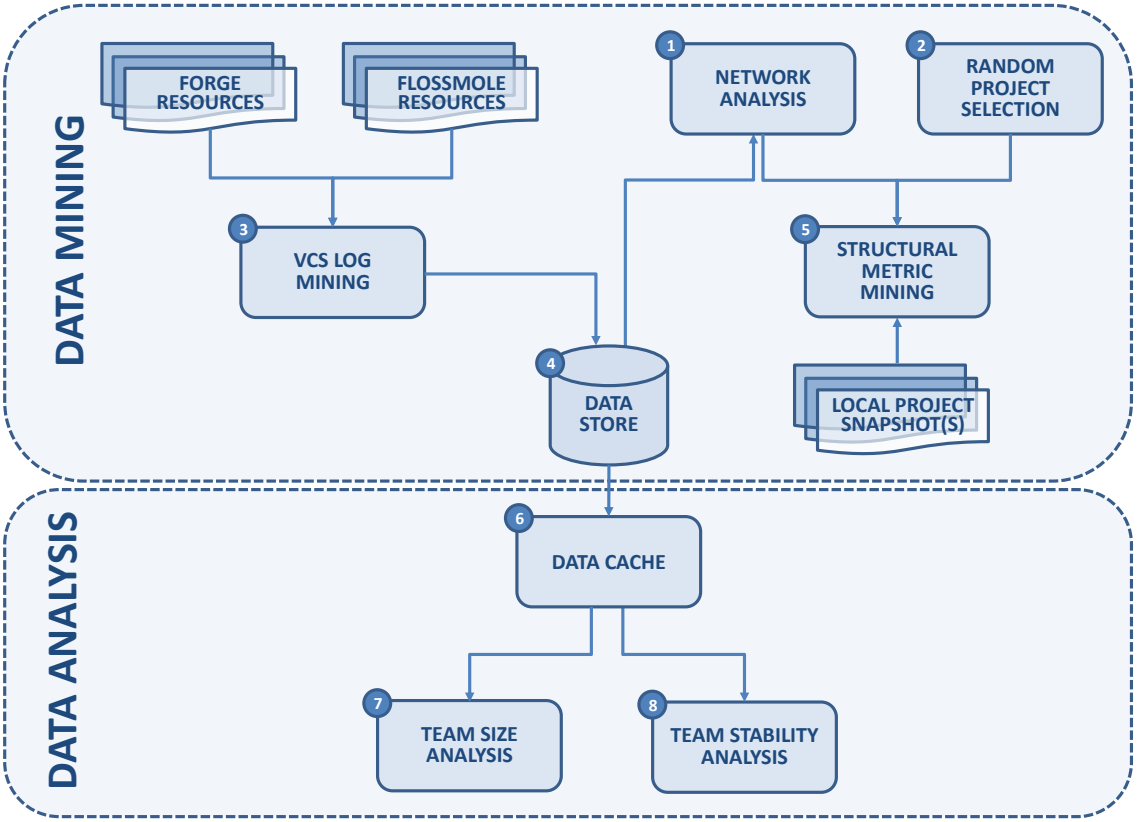


Fig. 3.4 Toolchain to mine and analyse the GoogleCode forge.

- **1. VCS Log Mining:** The FLOSSmole project makes available the raw data describing, at a project level, details of all projects hosted in GoogleCode. This, along with the

GoogleCode project webpages and the associated repositories formed the input into the bespoke toolchain illustrated in Figure 3.4. At its essence, the VCS mining component is a collection of shell scripts, run on a UNIX platform and driven by the FLOSSMole flat files and the GoogleCode project webpages, designed to retrieve, parse and persist the full revision history of all the projects in the forge. To mine this data, these scripts access all project repositories in the forge to retrieve the full revision history, parsing it accordingly and storing it in the database to then drive subsequent analysis. To achieve this, the scripts make use of command line tools made available by the VCS to retrieve meta-data about each project revision. Spinellis and Gousios adopt a similar approach to mining VCS meta-data, discussing its necessity in detail (Spinellis and Gousios, 2018).

These scripts support three version control systems: Subversion, GIT, and Mercurial. In the case of Subversion and GIT, the revision history can be directly queried through the appropriate binaries (albeit through different syntax and requiring individual log parsers) while in the case of Mercurial it is necessary to clone the repository before retrieving the history. The revision history extraction scripts use the repository URI to determine the type of version control system and extract the data appropriately.

- **2. Network Analysis:** The network analysis conducted in this research is focused on accurately mapping committer activity and project engagement in the context of the activity of the broader development team. This is used to calculate the stability of a team through the evolution of the project as well as understanding the change in team composition from one project to the next. This analysis also enables the mining of the full data set for projects that fulfil the criteria necessary for team stability analysis; for instance, pairs of projects where the development team remained stable. This will be discussed further in Chapter 5 as the team stability analysis is detailed.

- **3. Project Sampling:** This is a Java component designed to extract, in a reproducible fashion, a representative sample of projects from the full data set of 236,787 projects in the GoogleCode forge. There is functionality to identify and select projects programmed in a particular programming language (Java) to simplify structural metrics mining. This component is covered in detailed in Section 4.2 of the next chapter.

- **4. Structural Metric Mining:** This component comprises of, again, UNIX shell scripts responsible for checking out each version of the project, handing over the heavy lifting of metrics generation to an out-of-the-box metrics generation tool. Using an existing tool for structural metrics mining was favoured over writing a bespoke tool given that, unlike VCS mining, metrics calculation is complex and potentially error

Table 3.4 A comparison of a number of structural metrics mining tools

Tool	Category	Description
CKJM (Spinellis, 2005)	Open source standalone	This is a simple tool that calculates CK metrics by inspecting the compiled bytecode of the compiled java code. CKJM has hooks into Ant or Maven, two popular software build tools typically used for Java systems, and generates data in an XML format. CKJM is distributed under a 'Creative Commons' license which does not limit its usage for academic or commercial purposes.
Understand (SciTools, 2017)	Commercial standalone	Understand is a commercial tool, created and distributed by SciTools, which is freely available for academic use. Having gone through many release versions, Understand is a mature tool which is used by numerous multinational firms and large governmental institutions. Understand covers all the CK metrics in addition to size metrics and McCabe's complexity metrics.
Krakatau metrics (PowerSoftware, 2017)	Commercial standalone	Krakatau is another commercial tool, developed and marketed by Power Software, and with a much smaller user base. Again with support for all CK metrics in addition to size metrics. A key drawback is that it is not available on a free or academic licence.
Metrics (MetricsProject, 2017)	Open source IDE Plug-in	Metrics is an open source metrics generation tool designed to be a plug-in to the Eclipse IDE. Useful for ad-hoc use by developers, unfortunately it does not have the requisite integration points to be useful to this research.
nDepend (nDepend, 2017)	Commercial IDE Plug-in	This is a powerful commercial tool designed to integrate into the Visual Studio IDE and into Continuous Integration build processes. Metrics coverage is comprehensive and includes all CK metrics. Not available on a free or academic licence.

prone. This meant that creating a bespoke tool would require significant build and validation effort. Fortunately the landscape of available tools was sufficiently rich such that a suitable tool could be identified with ease and without the need for extensions or enhancements. Table 3.4 shows a comparison of the available metric generation tools categorised by the distribution license (commercial or open-source) and deployment (standalone or integrated into an IDE) (Spinellis, 2005; SciTools, 2018; PowerSoftware, 2018; MetricsProject, 2018; nDepend, 2018). 'Understand' by Scientific Toolworks Inc. was chosen as it is amongst the most reputable and widely adopted, was available on academic license, and offers a command line tool that generates metric reports in an easily parsable format. The report created by Understand is then passed through a Java Parser which extracts the information that is pertinent to this research and stores it in a format appropriate for use by the metrics analysis component. The calculations used by Understand to generate metric values are documented in Table 3.5.

- **5. Data Store:** Raw metrics and analysis should be stored in a queriable format to facilitate further data analysis - particularly statistical analysis - to identify trends and

Table 3.5 A description of how CK Metric values are calculated for Java classes by Understand.

Metric	Full Name	Calculation for Java classes in 'Understand'
CBO	Coupling Between Objects	Number of other classes invoked from this class. The following do not contribute towards the CBO calculation: - Coupling to classes in external dependencies. - Classes wired through dependency injection (e.g. using frameworks such as spring). - Coupling to interfaces. - Inner static or non-static classes where are coupled to the outer classes.
DIT	Depth of Inheritance tree	Number of parent classes in total. The following do not contribute towards the DIT calculation: - Implemented interfaces. - The inheritance tree of extended external dependencies. For example my.CustomException extending java.lang.Exception extending java.lang.Throwable will return a DIT value of 2 for CustomException rather than a value of 3 which would otherwise be returned if the entire dependency tree was traversed.
LCOM	Lack of Cohesion of Methods	For each member variable calculate the percentage of methods which do not access that variable. Average the percentages to determine LCOM. Visibility modifiers and the static keyword do not affect the calculation here.
NOC	Number Of Children	Count of other classes that directly extend it. The restrictions listed on DIT apply here.
RFC	Response For a Class	Number of total methods including all methods in parent classes (regardless of invocation or visibility).
WMC	Weighted Methods per Class	Count of all methods in that class only (regardless of invocation, visibility, and instance or static).

correlations. A MySQL database was created with a schema reflecting the data model outlined in Figure 3.5 to store the results of the VCS log mining and the structural metric mining.

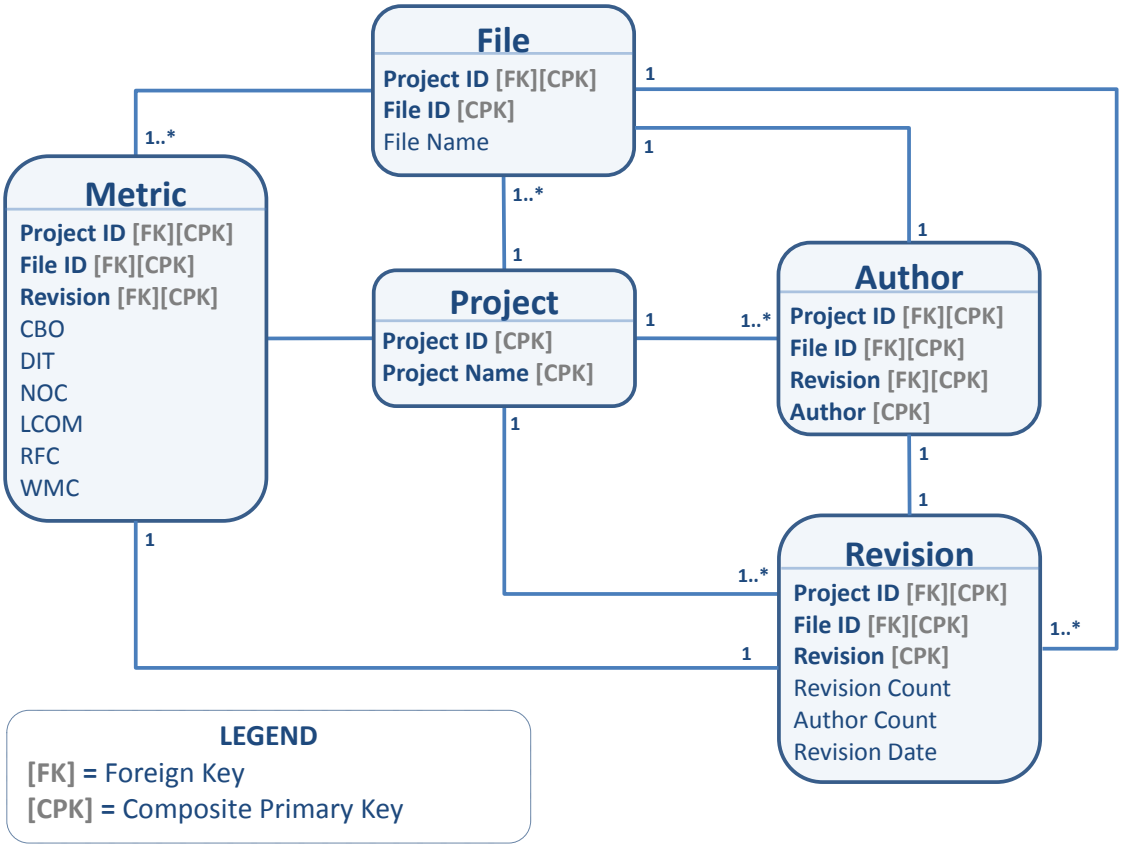


Fig. 3.5 A simplified ER diagram depicting the data model used by the data store. This is closely reflected by the object model underpinning the analysis components.

- **6. Data Cache:** When conducting analysis - and in particular network analysis - it is necessary to traverse a large data set multiple times in order to make meaningful observations on the data.

  For example, to establish which committers have common project engagements, it is necessary to traverse the data set and categorise project engagements by committer. To calculate the stability of a team within a project, a traversal of the data for that project is required to determine the overlapping committer engagement periods. For this reason, it is efficient to load entire commit data into an in-memory data structure within the Java Virtual Machine running the team size and stability analysis to facilitate rapid access and flexible indexing. In practice this is implemented using hash maps keyed by author and project as appropriate for the requisite type of analysis.

- **7. Team Size Analysis:** This is the subject of Chapter 4 so will not be discussed at length here. However, as a brief overview it is worth mentioning that a Java component queries the database and joins structural metrics with commit-level information in order to produce an aggregated result set. This result set is then analysed using the Anaconda Data Science workbench (Anaconda, 2018) running Python 'notebooks', the source code for which is available on GitHub in the location previously specified in the chapter titled 'Publications'. Figure 3.6 illustrates the approach to data analysis across both team size and stability.

- **8. Team Stability Analysis:** Similarly, this analysis is the subject of Chapter 5 but suffice to say that the design is similar to the previous component with a focus on reading commit data and its associated meta-data to calculate team stability measures. This then drives the categorisation and statistical comparison of structural metrics in order to ascertain the impact of stability on metric values.

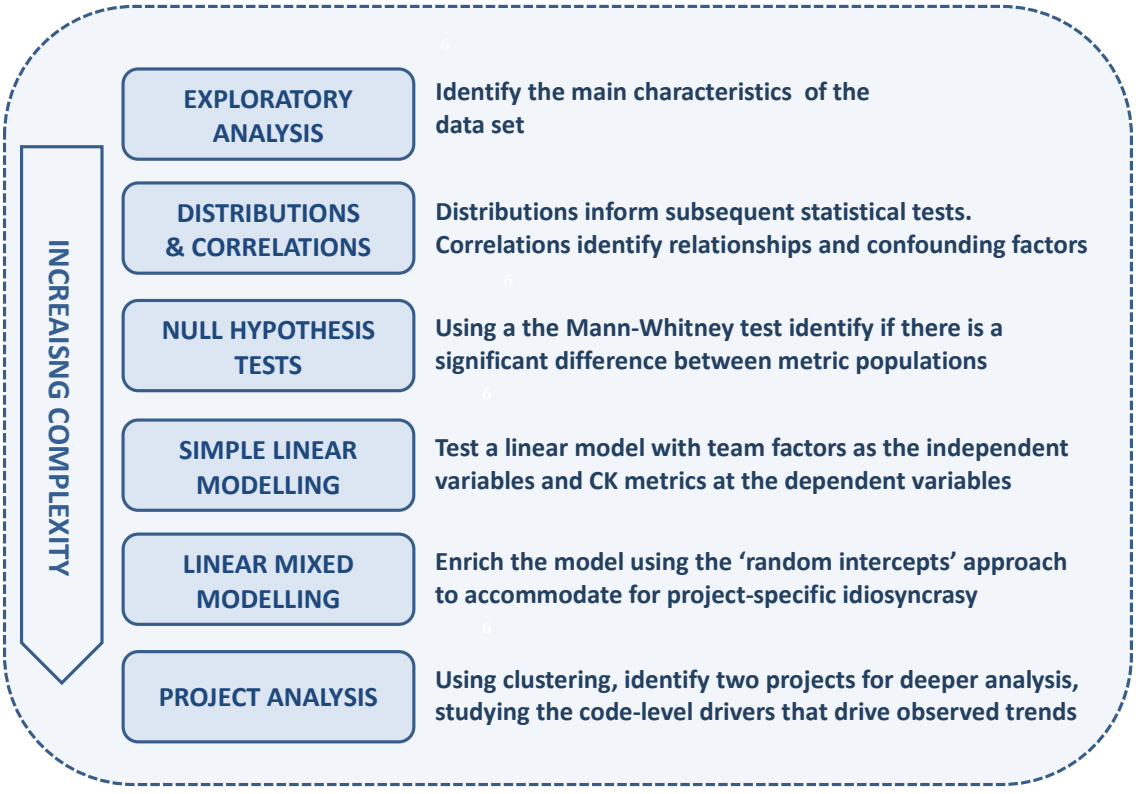


Fig. 3.6 A summary of the common approach to data analysis across both team size and stability.

3.3.7 Validating the Toolchain

The toolchain described bears a basic high-level similarity to one independently developed by Ludwig et al. to extract metrics of maintainability from GitHub repositories (Ludwig et al., 2017). One challenge involved in developing a bespoke toolchain was the requisite effort to ensure its validity. Whenever writing code, typically good engineering practices such as ensuring adequate unit test coverage should be a given (Tosun et al., 2018). The approach taken to development of components for this research was no different. This gives some level of confidence that individual elements in the toolchain have been correctly implemented.

However, to get adequate confidence that the complete toolchain is functionally sound from an end-to-end perspective, it was necessary to manually validate the toolchain results against a small project. To ensure that the validation process was not overly taxing yet suitably thorough, the project to validate was chosen to have a limited number of project revisions yet have multiple authors including multiple revisions and authors on the same source file. The project selected for this purpose was zsea-planetwars as it has 8 revisions and 3 authors. The project was downloaded in each of its 8 snapshot states and imported within the Eclipse IDE. For each file within each snapshot, the metrics were derived by hand and validated successfully.

3.3.8 Toolchain Performance

The toolchain uses standard programming languages and does not require any specialised hardware or operating systems. However, as mining processes are network bandwidth intensive, the software was deployed to a commercially hosted Virtual Private Server (VPS) as this guaranteed faster internet connectivity than could be achieved with a typical domestic connection. The VPS account was an entry-level single core deployment with 512MB RAM, 25GB of data storage. The Linux distribution was CentOS 6.8.

The time taken to download VCS logs is dependent on the volume of revision history for a given project but is generally in the single digit seconds. Mining structural metrics is a more time-consuming process and can take around several minutes for a single 'snapshot' of the source code. This is a roughly equal distribution between the time taken to download the source code and the time taken to run static code analysis. Medium sized projects can have hundreds of revisions which leads to a total analysis time in the hours. As a rough guide, mining structural metrics for a 1000 project sample takes about 300 hours. Even with

the limitation of a single core, this process can be made much faster by running parallel processes (multi-threading). 25 threads was found to be optimal and resulted in a sample processing time of approximately 24 hours.

Once stored in the MySQL relational database the data associated with all revisions of a single project within the sample takes up approximately 4-5MB of disk space. While this may present a technical challenge when mining the entirety of the forge for structural metrics, this was not a concern when limiting the static analysis to a representative sample of projects.

3.4 Chapter Review

This chapter covered three fundamental aspects to the methodological approach of this research. First, measurable definitions of team size and stability were established, laying the groundwork to the empirical work in this thesis. The FLOSS mining toolchain was then detailed, covering aspects of performance and validation. Figure 3.6 illustrated the general approach to data analysis across the team size and stability analysis that will follow in the coming chapters. Finally, structural metrics are covered in detail and the process leading to the selection of the CK metrics for this research is covered.

The next chapter uses the methodology described in this chapter, along with a number of linear models, to establish a relationship between team size and the structural attributes of software.

Chapter 4

The Impact of Team Size on Structural Metrics

4.1 Introduction

The first research question (RQ1) seeks to establish '*the impact of development team size on the internal structural attributes of software projects and the implications on its maintainability*'. In practice this means first establishing if populations of structural metrics of software, when grouped by team size, exhibit statistically significant differences. Where such a difference is observed, the nature of that difference should be ascertained.

This chapter follows a methodical approach of initially conducting foundational work to facilitate that statistical analysis. The subsequent sub-sections articulating the analyses are of incrementally increasing complexity as outlined in Figure 4.1. Section 4.2 is titled 'Data Mining' which details efforts to extract a data sample from the broader forge and to mine the sample for structural metrics. Section 4.3 covers an initial 'Exploratory Data Analysis': efforts to understand the underlying trends in the mined data from a one-dimensional perspective in addition to the distribution and correlations within. Section 4.4 documents an initial, 'Univariate Analysis' to establish a relationship between team size and structural metrics - simplistic as the confounding impact of revisions (potentially a proxy to functional complexity) means that this analysis will not yield clear or reliable results but will serve to establish the foundation on which the further analysis will take place. These confounding factors will be the subject of Section 4.5 which factors in the impact of revisions to observe the impact

of the development team size, alone, on structural metrics. In Section 4.6 the results are presented in the context of two individual projects and qualitative code-level observations are made to highlight the reasons that drive the quantitative forge-level observations. Finally Section 4.7 provides a summary of the results and relates the observed relationship between team factors and CK metrics to the likely impact on maintainability. Figure 4.2 highlights the aspects of the toolchain that will be subject to further discussion in this chapter.

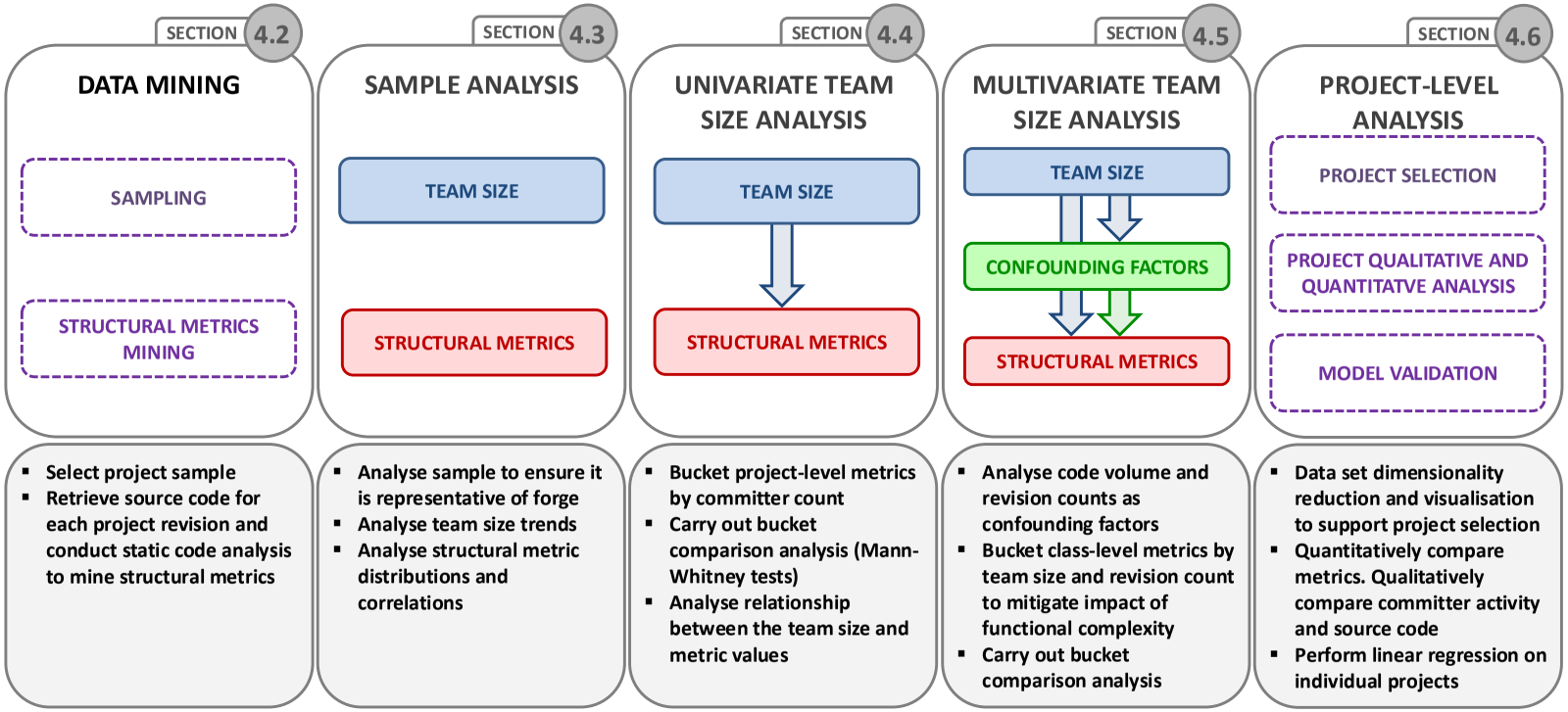


Fig. 4.1 Chapter 4 outline providing an overview of the contents of each section.

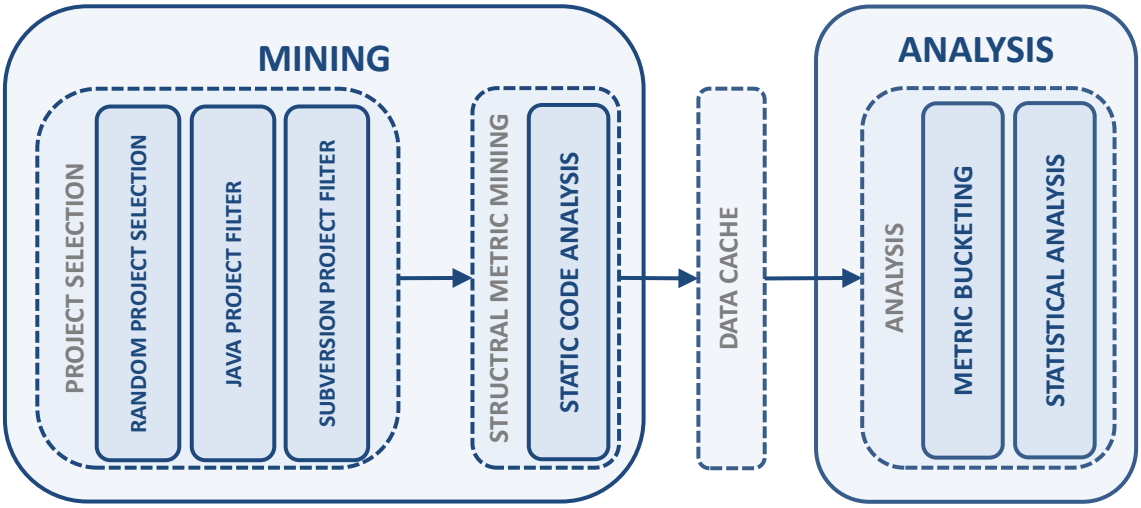


Fig. 4.2 Some aspects of the toolchain pertinent to team size analysis. Project Selection and Structural Metric Mining are both the subject of discussion in section 4.2. Analysis is the subject of the remainder of the chapter.

4.2 Data Mining

When analysing basic committer activity within a forge it is possible to learn a substantial amount through mining the revision logs across all the repositories within the forge. Although requiring significant processing, given modest resources it is computationally feasible to mine the revision history of all projects in a forge made of up several hundred thousand repositories. It is necessary to mine the entire forge when conducting network analysis (as discussed in detail in the next chapter). This, however, is not the case when conducting the detailed structural metric based analysis contained within this chapter. To mine structural metrics necessitates network, processor and storage intensive processes to execute the retrieval and static analysis of the source code at each revision for each project studied. As described in section 3.3.8 in the previous chapter this is still a very resource intensive task and it is not practical or necessary to run this type of analysis across the entirety of the forge. For the purposes of this research it is most appropriate to restrict this type of in-depth analysis to random sample of projects.

With this context in mind, this section covers two key aspects of the data mining approach that are foundational to the team size analysis: Project Selection and Structural Metric Mining.

4.2.1 Sampling

The project selection component was outlined in section 3.3.6 of the previous chapter and is a Java component taking, as its input, flat files made available by FLOSSmole providing a full listing of all projects hosted by GoogleCode. Its first responsibility is to identify, within this population, a sample of projects programmed in Java. As discussed earlier, Java was chosen as it increases relevance to the academic and practitioner community and restricting to projects of a single language greatly simplifies the data mining by not requiring support for static code analysis across multiple languages. While restricting the analysis to a single language aids practicality, it also presents a threat to validity when generalising the applicability of any results to projects in any other language, as was highlighted in early research (Basili et al., 1996). While some similarities carry across object-oriented languages, earlier work has highlighted some of the structural differences between Java and C++ projects (Subramanyam and Krishnan, 2003; English and Mc Creanor, 2009) finding that Java lends itself to greater maintainability. These threats to validity are the subject of further discussion in Chapter 6.

While creating projects in GoogleCode the project administrator can associate 'tags' with the project. This is essentially meta-data enabling the categorisation of projects for the purposes of searching or browsing projects. FlossMole provides this meta-data in flat file format. Analysis of this meta-data shows that 204,918 projects (87%) have at least one tag and on average each project has 4 tags defined. Given that tags are free text, there are a large number of unique tags (145,600). The top 250 most frequent tags were extracted and manually categorised accordingly. The relative occurrences of each category was then calculated on the basis of the 250 most frequent tags and depicted in Figure 4.3.

The results in Figure 4.3 demonstrate that tagging projects by programming language is fairly common. It follows that identifying a substantial set of Java projects is achievable through tag analysis alone. Table 4.1 shows Java as the most popular language tag associated with 22,594 projects (making it also the most popular tag in any category). This set of projects is considered the *population*.

GoogleCode allowed project administrators to choose from among three available version control systems - Subversion, GIT, and Mercurial - on which to host their source code. Each of these necessitate a different mechanism to mining data. In the case of Subversion and GIT, the revision history can be directly queried (albeit through different syntax and requiring individual log parsers) while in the case of Mercurial it is necessary to clone the repository

before retrieving the history - an altogether slower process. As this research is concerned with building a toolchain that can efficiently mine data from each repository type, it is necessary to understand the distribution of projects across types. Table 4.2 shows that Subversion was overwhelmingly the most popular repository type. This reflects the relative popularity of these version control systems at this point in time as GIT was later to see greater adoption in the industry, eventually becoming the most popular VCS by web searches according to Google Trends (RhodeCode, 2016). De Alwis and Sillito attribute this transition to the decentralised nature of GIT which allows enhanced developer access, greater support for experimental changes and improved workflows around branching and merging (De Alwis and Sillito, 2009). While the revision history mining effort spans all projects (necessary for comprehensive network analysis in the coming chapter), the static code analysis effort is simplified by exclusively selecting Subversion projects to mine for structural metrics.

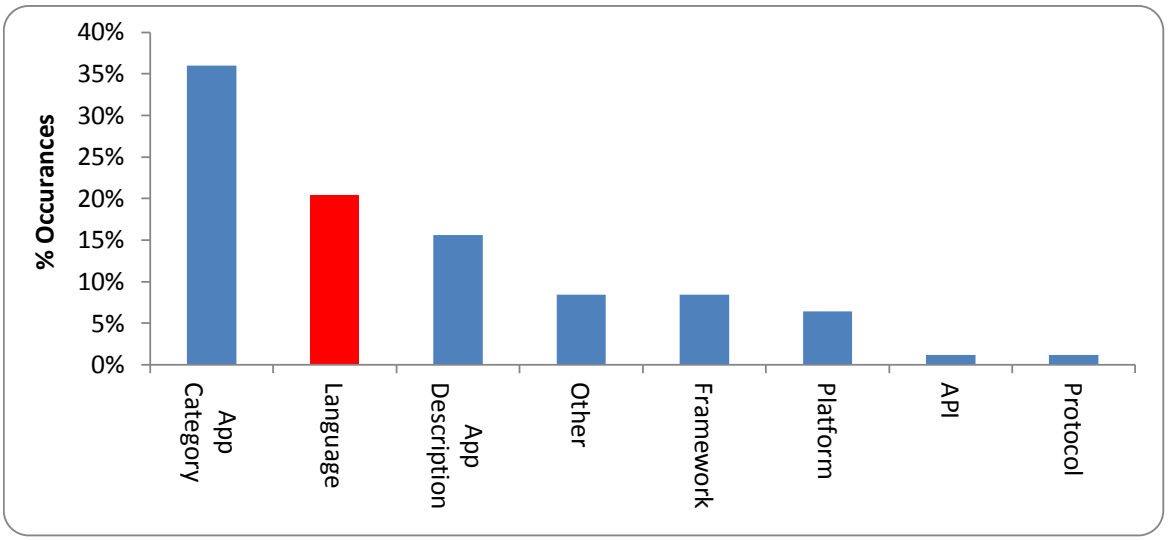


Fig. 4.3 A representation of the categorisation of tags and their relative occurrences across the forge

A *sample* is extracted from the *population* of 22,594 projects by iteratively invoking a simple pseudorandom function (the 'random' method in the Java Utils API which has an approximately uniform distribution) to select a number between 0-1 to be multiplied by the total number of projects available until the sample is extracted of an appropriate size. The selection algorithm discards from consideration any projects with no revision history as they represent projects which never saw committer activity beyond initial project creation and therefore have no relevance to this study. With a confidence level of 99% and a confidence interval of 5%, a minimum sample size of 646 was calculated. The actual sample size used was arbitrarily higher than the minimum sample size at 658 projects.

Table 4.1 The top 5 tags within the language category showing the 'Java' tag the most popular of all.

Programming Language	Occurrences (Projects)
Java	**22,594**
Python	13,525
PHP	12,180
C	8,144
GME	7,858

Table 4.2 The repository counts for each version control system across the forge.

Version Control System	Repository Count
Subversion	**207,390**
GIT	1,643
Mercurial	2,064

Once the sample projects are selected, a utility in the toolchain allows for the automated parsing of the repository URL from the relevant project's page on the GoogleCode website. The project list with the associated URLs are then consolidated in a single file output which is used to drive the metric mining process.

4.2.2 Structural Metrics Mining

The broad approach to mining project repositories for structural metrics is documented in section 3.3.6 of the previous chapter. This section builds upon that foundation to detail the aspects of mining that are specific to the particular analysis that forms the latter half of this chapter. For the initial univariate analysis, static code analysis is conducted on the latest version of the code of all selected projects within the sample. However, the later multivariate analysis, as will be covered in section 4.5, the identification of revision counts as a confounding factor will necessitate the analysis of every revision of each project and subsequently the static code analysis to mine for structural metrics - again for each revision. This creates a requirement to store structural metrics for each Java class file in each project for every revision of that project. Given that there can be hundreds of class files and thousands of revisions, this is a fairly large data set. In order to be able to query this data effectively

there is a reliance on the database model documented in Figure 3.5. This is in contrast to the univariate analysis where only a single set of structural metrics is captured for the entire project. Figure 4.4 illustrates the difference in these two approaches. On the left, there are three revisions and three classes: class A being revised three times and class C revised just once. On the right the contrast is shown between the univariate analysis (where only the final revision is considered) and the multivariate analysis (where all three revisions are considered).

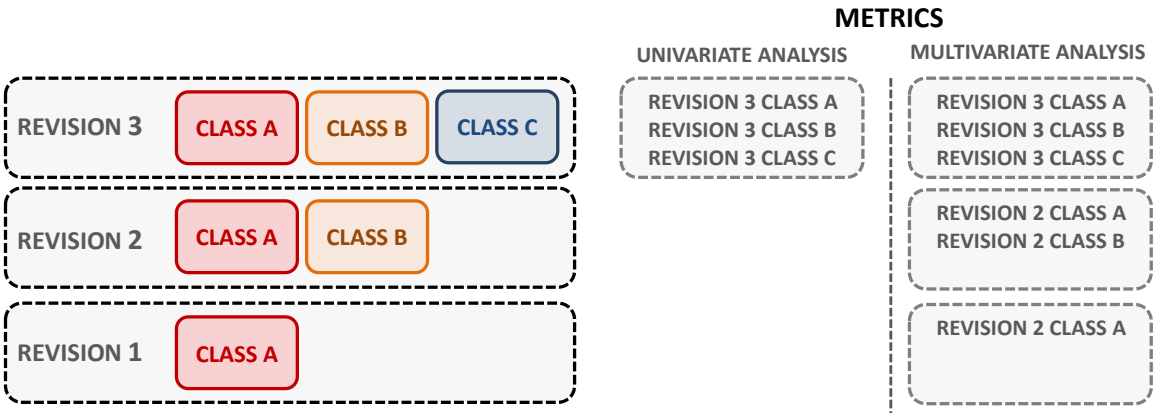


Fig. 4.4 An illustration of the difference between data sets for each of the univariate and multivariate analysis.

4.3 Sample Analysis

Ahead of conducting a detailed analysis of the *sample* or carrying out a regression analysis to answer the research questions, it is standard practice to conduct an initial exploratory data analysis. This is a typical approach in this type of research (Grechanik et al., 2010; Cartwright and Shepperd, 2000) as it enables for a dissection of the dataset allowing it to reveal its underlying structure without prior assumptions or biases. It is through that initial analysis that potential complexities or threats to validity can be identified mitigated (Tukey, 1977). This section details that exploratory data analysis as applied to the *sample* of 658 projects and presented against similar analysis in the *population* of 22,594 projects. This analysis will show that the sample is broadly representative of the wider forge.

Table 4.3 File Extensions: top five cumulative occurrences.

Extension	Count
JAVA	311,252
XML	38,089
HTML	37,817
PNG	33,141
GIF	27,082

Table 4.4 The number of projects within the sample of 658 projects, grouped by team size.

Team Size	Project Count
1	465
2	102
3	37
4	21
5	21
> 5	12

4.3.1 Exploratory Data Analysis

First, the extensions of files contained within the commits were mined and analysed with the results confirming a very heavy bias towards the Java file extension alongside other file extensions usually associated with Java web-based projects. This is as expected given the project sample selection criteria which exclusively selected 'Java' tagged projects. The results of this analysis are depicted in Table 4.3. The table reveals a substantial proportion of XML files (typically used in Java projects for configuration) and HTML and graphics files (mostly used in web-based projects).

Next, the cumulative number of project committers was analysed. This is of relevance to this analysis as it is used as a measure of team size. Figure 4.5 shows that the sample follows a similar profile to the forge-wide analysis. Table 4.4 summarises the project team sizes in greater detail. The analysis reveals that more projects exist on lower committer counts; a fact that will have a bearing on the upper limit imposed on team size throughout the analysis in this chapter to help ensure a substantial population of data points for each team size that is included in this study.

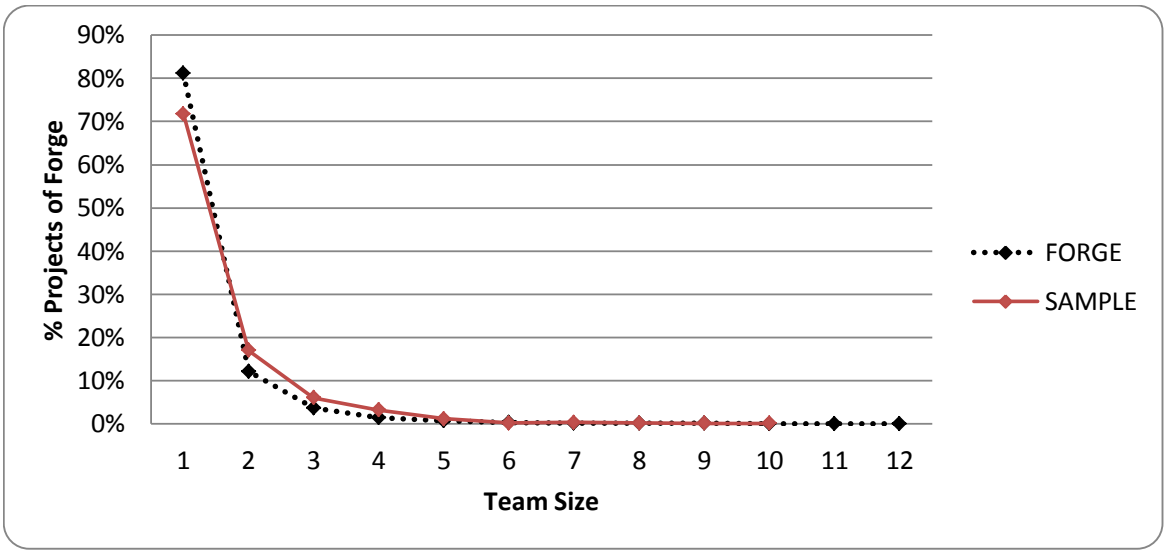


Fig. 4.5 The number of projects with a team size of 1, 2, 3 through to 12.

Committer behaviour was also analysed by investigating the cumulative number of commits grouped by committer and project, depicted in Figure 4.6. It is notable that the broader forge shows a majority of committers only ever contributing a single commit when participating in a project. The sample deviates from this trend exhibiting a more even distribution across committer activity levels. This is attributable to the fact that the sample constituting only those projects tagged with the appropriate meta-data and, hence, are more likely to be active projects showing more sustained committer engagement. These trends will bear relevance to the multivariate team analysis later in this chapter where structural metrics are grouped by the number of revisions that files undergo.

The final aspect of sample analysis concerns the nature of the individual commits that make up committer activity. An individual commit can have any number of affected files. A commit could be a single file or path modification or, on the other extreme, the check-in of large mature codebase. Figure 4.7 shows the general trend that smaller commits are more frequent than larger ones. Any file-level analysis of commit information will necessitate the joining the structural metrics of potentially multiple affected files with the commit-level data. This will become apparent through the course of the multivariate team analysis.

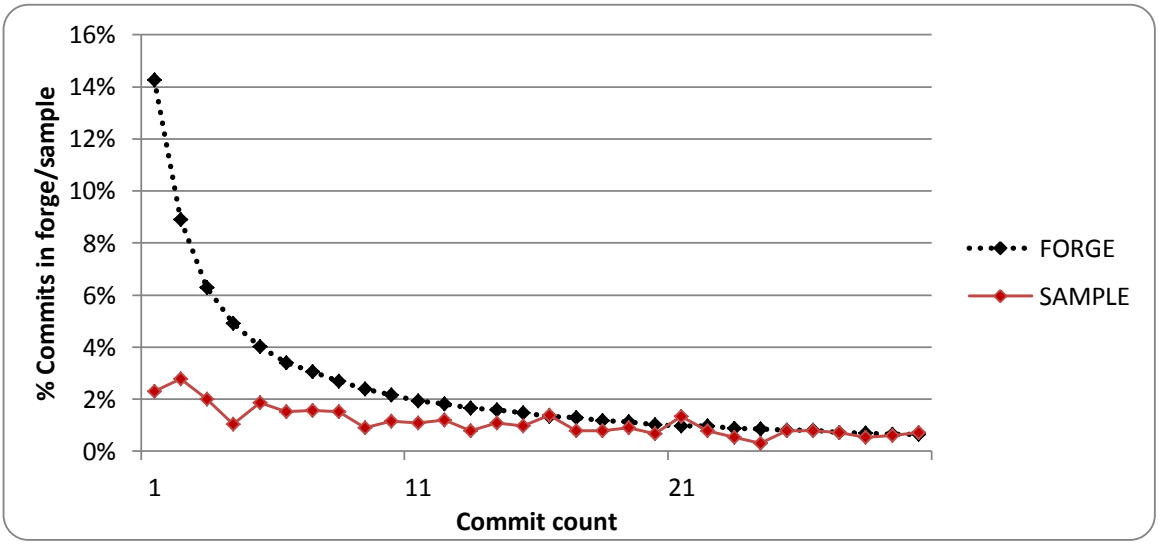


Fig. 4.6 Sample analysis of the number of commits that each committer makes within individual projects. The chart shows x-axis shows the number of commits that a committer contributes to a project and the y-axis shows the frequency of that level of project engagement.

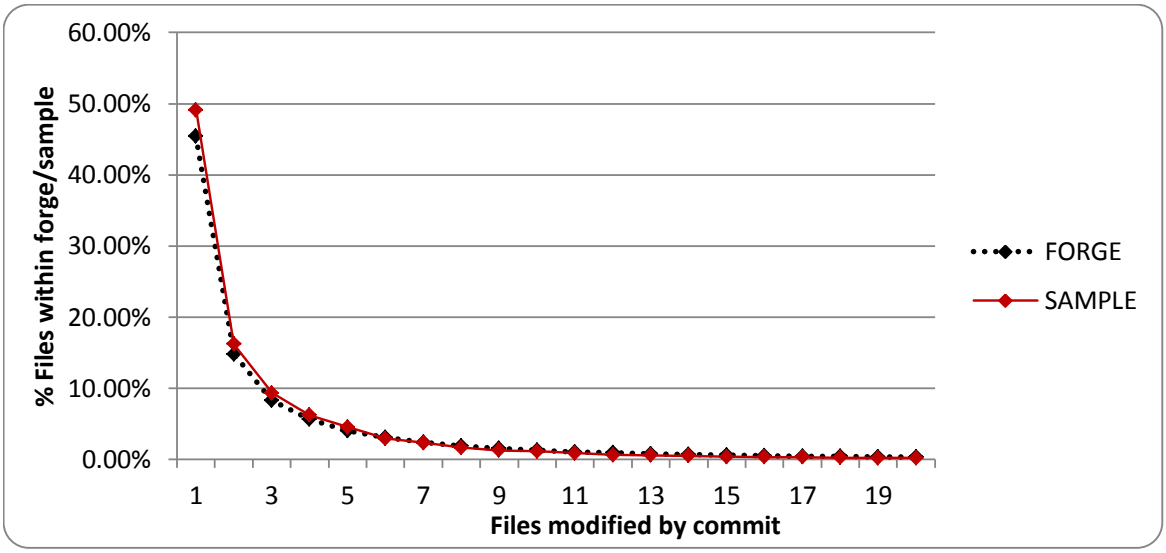


Fig. 4.7 The frequency of commits grouped by the number of files affected.

Table 4.5 Results of Kolmogorov-Smirnov tests comparing the distribution of each metric against a 'reference' normal distribution.

	CBO	DIT	LCOM	NOC	RFC	WMC
Test Statistic (D)	0.62	0.84	0.68	0.50	0.87	0.84
Significance (p-value)	0.00	0.00	0.00	0.00	0.00	0.00

4.3.2 Distributions and Correlations

To conduct robust hypothesis testing and apply linear regression techniques, it is necessary to understand how the key parameters of the data set distribute relative to a normal distrubtion. An in-depth qualitative analysis of the distributions of CK metrics is not the focus of this work and the reader is referred to the work of Succi et al. (Succi et al., 2005) and Basili et al. (Basili et al., 1996) for an analysis into the typical distribution of CK metrics for Java and C++ software respectively. For the purposes of this research the Kolmogorov-Smirnov test was used to compare the population of values of each CK metric against a normal distribution (Kolmogorov, 1933; Smirnov, 1948). This test makes no assumptions of the distribution of those data sets being compared and produces a p-value and a D-statistic as an output. Table 4.5 shows zero p-values allowing the rejection of the null hypothesis of normality. The D-statistic indicates the ratio of the data sets that exists outside the normal distribution. This insight will influence the choice of statistical methods later in this thesis.

To analyse the relationship between the metrics within the sample, the Spearman correlation coefficients are analysed in Figure 4.6. The Spearman measure of correlation was chosen as it makes no assumptions of the normality of the distribution of the data being analysed (Spearman, 1904). The correlation matrix shows positive correlations between CBO, LCOM, RFC and WMC. Weak correlations exist DIT, NOC and the remainder of the metrics. These observations are in-line with prior analysis by Succi et al. (Succi et al., 2005). Team size is found to be correlated to CBO and, to a lesser extent DIT, NOC and RFC. It is worth stressing that, for reasons explained in the next section, team size is measured at a project level while the metric values are at an individual class level. This initial result implies that greater team sizes cause metrics to trend counter to the objective as articulated by Rosenberg and others (detailed earlier in Chapter 2) - something which has been empirically associated with degraded external attributes. This adds further weight to the empirical evidence of prior research associating larger team sizes to lower productivity (Pendharkar and Rodger, 2009) and greater fault-proneness (Nagappan et al., 2008; Caglayan et al., 2015).

Table 4.6 Spearman correlation matrix for Team Size and the CK metrics within the sample.

	TEAM SIZE	CBO	DIT	LCOM	NOC	RFC	WMC
TEAM SIZE	1						
CBO	0.25	1					
DIT	0.13	0.32	1				
LCOM		0.24	0.14	1			
NOC	0.14	0.46	0.57		1		
RFC	0.12	0.52	0.38	0.53	0.50	1	
WMC		0.40	0.14	0.64	0.16	0.87	1

KEY	+VE STRONG	+VE MODERATE	+VE WEAK	NOT SIGNIFICANT
	0.7 - 1.00	0.40 - 0.69	0.01 - 0.39	

4.4 Univariate Team Size Analysis

4.4.1 Defining the Team Size

For the purpose of this research, team size is defined as the cumulative total of all unique committers present in the revision history in the version control system of a given project.

As alluded to in the previous section, while it is reasonable to suggest that such a definition would be an oversimplification as some committers could contribute to the codebase during widely varying time windows, it was posited that the majority do commit in overlapping time windows. This is borne out in analysis of the project sample that found that 83% of committers to a project contribute in overlapping time windows to their fellow contributors, an example of which is shown in Figure 4.8. This fact, alluded to the reality that those committers who contribute outside the time window of their peers nevertheless leave an impact to the codebase which cannot be discounted, reinforces the argument in favour of a simple cumulative measure of team size.

Mockus et al. studied an Apache project and observed that the majority of development was attributable to a minority of 'core' developers (Mockus et al., 2000) who commit frequently to the codebase. It could be suggested that the measure of team size proposed in this research does not distinguish between this frequent core committers and infrequent 'peripheral' committers. As Figure 4.9 illustrates, a significant amount of activity takes place by those committers who contribute with a low number of commits, hence infrequent committers

cannot be excluded from any analysis without losing a key influencing factor on the codebase. Indeed Terceiro et al. studied the contributions of core and peripheral committers across 7 projects concluding that peripheral committers contribute a disproportionately high amount of structural complexity to the codebase further strengthening the argument that a measure of team size should include all committers (Terceiro et al., 2010). Figure 4.8 highlights the simple and intuitive nature of a team size measure derived from the cumulative committer count as contrasted with a count of the number of committers with overlapping committer activity.

It is also reasonable to suggest that, given CK metrics are measured at an individual class level, so team size could also be measured at a class level (perhaps as the total number of committers to modify a class). This runs somewhat counter to the intention of making this research relevant to practitioners (particularly middle-management) which, the author argues, would relate much more to a measure of team size at a project level than at a class level.

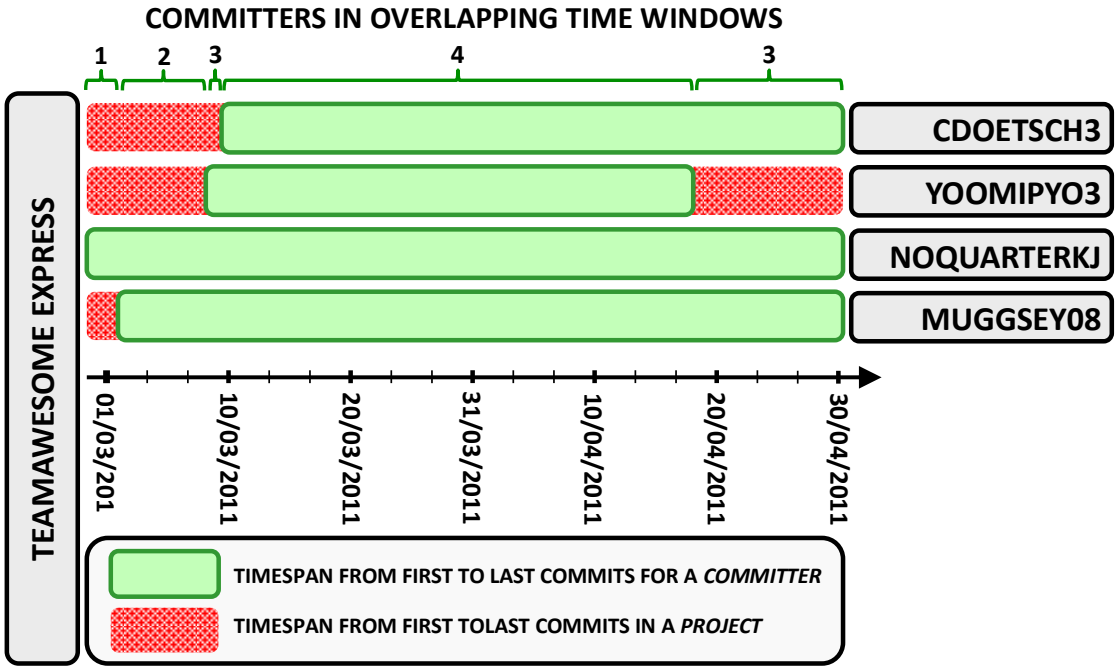


Fig. 4.8 The committer engagement timelines for project 'TeamAwesomeExpress'. The diagram depicts the count of the number of committers with overlapping timelines of activity.

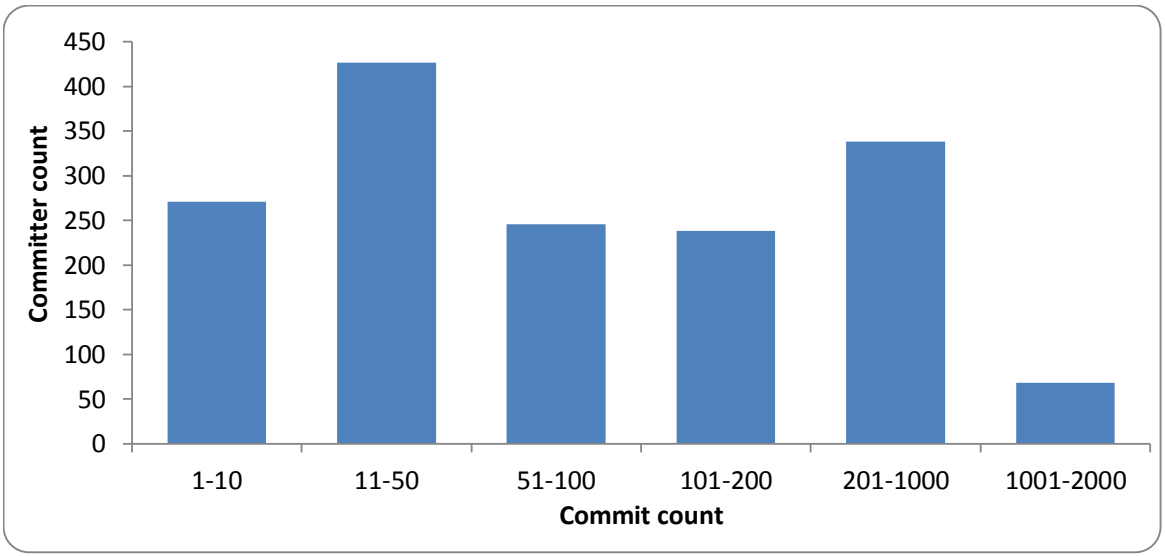


Fig. 4.9 The number of commits that each committer makes within individual projects.

4.4.2 Analysis

The analytical approach to the 'univariate' team size analysis uses, as its structural metric data set, the results of a static analysis of the projects codebases snapshotted at their final revisions. This is depicted in Figure 4.10 and illustrates the approach of overlooking the evolutionary path that a project took to reach its final (snapshotted) end state.

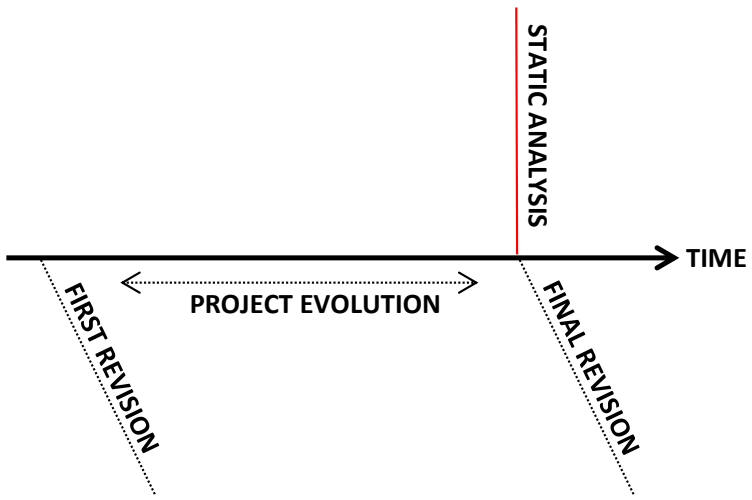
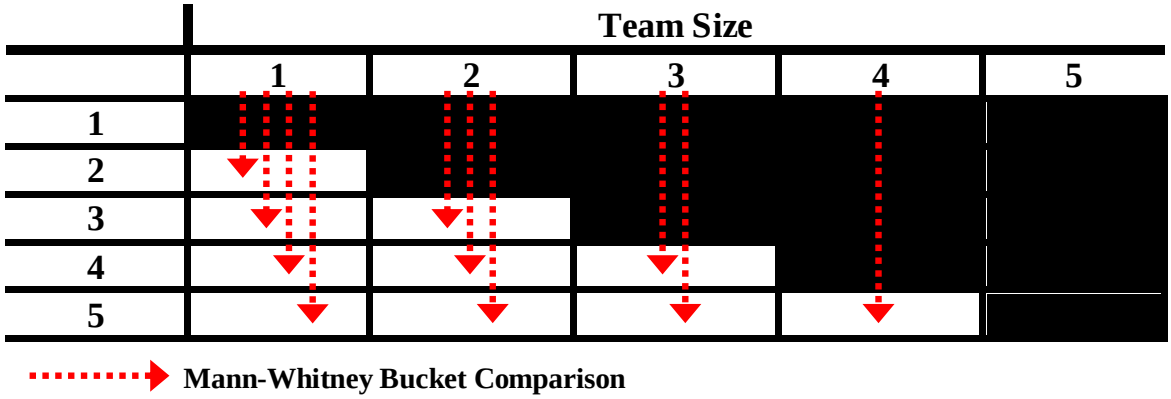


Fig. 4.10 The evolution of project along the axis of time overlaid with a depiction of the point at which static analysis is conducted.

Table 4.7 The bucketed metric comparison strategy.



Armed with these structural metrics, project metrics are then grouped or 'bucketed' according to the cumulative number of committers to the originating project. By way of example, should the total number of committers for a project number *n*, then all the class-level structural metrics for that project would move into the *n* committers bucket. This is illustrated in Figure 4.11.

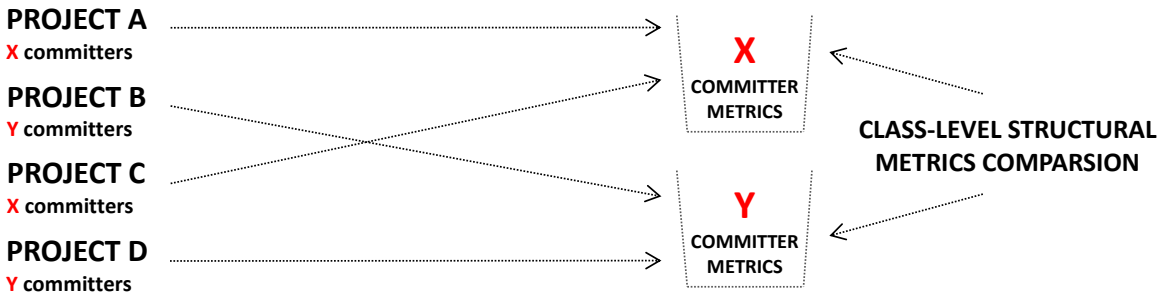


Fig. 4.11 An illustration of our bucketing strategy categorizing the class-level structural metrics of a project according to the cumulative committer count of that project.

As shown in the sample analysis in Figure 4.5 earlier, there is a dramatic drop in the proportion of projects that have a higher committer count compared with those projects with a lower committer count. In the sample this drop is particularly dramatic from 5 committers to 6 committers with counts of 12 and 3 projects respectively. For this reason, the statistical analysis excludes buckets for 6 or more committers. As will be discussed in Chapter 6, this can present an external threat to validity generalising some of the findings in this research to substantially larger team sizes.

Table 4.7 illustrates the approach of comparing each bucket to every other. When comparing metrics populations bucketed by committer count, the Mann-Whitney test is ideally suited as

all metrics populations are independent and consist of continuous non-normal data (Mann and Whitney, 1947). Unlike the t-test, the Mann-Whitney test is a null hypothesis test that makes no assumptions of the distribution of the data. Null hypothesis H0,1 anticipates no significant difference between bucket populations of varying committer count and is rejected where the p-value is less than a certain threshold denoted by α. With a target confidence level of 95% this would imply a p-value cut-off (α) at 0.05. However, a number of hypothesis tests are conducted, increasing the likelihood of observing at least one significant result at our chosen value of α. It is therefore necessary to apply the 'Bonferroni correction' which sets the significance cut-off at the product of α and n where n represents the number of tests (Bonferroni, 1936). In this analysis there are 10 hypothesis tests for each metric giving a corrected α value of 0.005. Where the null hypothesis is rejected, the mean value of the bucketed metric populations are compared to determine in which direction the metric value is trending. Table 4.9 shows the mean and median metric values and observation counts by bucket.

Table 4.8 summarises the results of this analysis, based on which, two immediate observations can be made.

- **Rejecting null hypothesis H0,1:** The null hypothesis can be rejected in the case of all metrics with the exception of NOC as p-values are lower than the α for most of the remaining metrics comparison tests. NOC excepted, 44 out of 50 comparisons show p-values under the α threshold.
- **Metrics not overwhelmingly trending in a one direction:** Based on the analysis of simple means, there is a roughly even split between metric values increasing in value by committer count and those decreasing in value. Looking at the trends in more detail, DIT and RFC bucket comparisons show a dominant trend of decreasing metric values with increasing committer count. However, DIT shows the opposing trend where 1 committer bucket is compared to 2, 3, and 4 committer buckets. Similarly, the LCOM bucket comparisons show a dominant trend of increasing metric values with increasing committer count.

While some trends do emerge from this analysis, it is important from the perspective of the validity of this research to ascertain whether there are any potentially confounding factors that may be impacting these results. That is the subject of the next section.

Table 4.8 Tabular summary showing the results of each bucket comparison within the sample analysis.

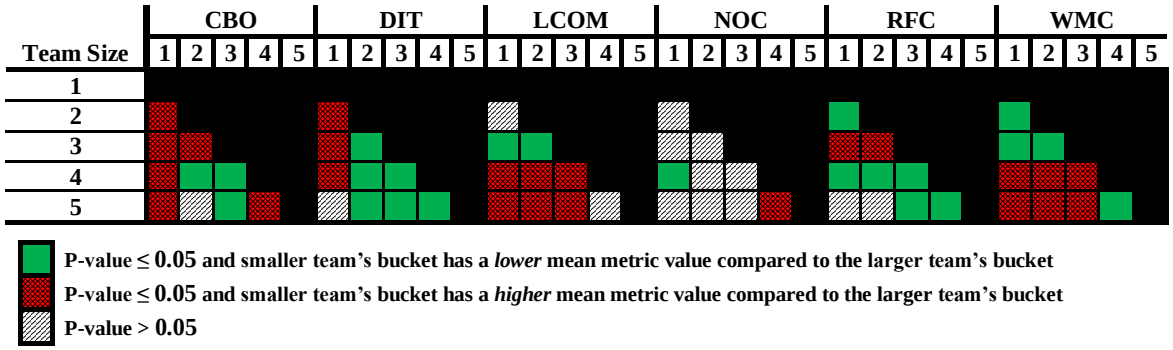


Table 4.9 Metric mean and median values for each metric bucket within the sample analysis.

		Mean CK Metric Values					
Team Size	**Count**	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
1	68087	2.76	1.76	41.73	0.27	12.34	7.16
2	28734	3.58	1.71	43.58	0.32	12.69	7.24
3	17296	3.98	1.68	46.21	0.35	12.81	8.15
4	106966	4.45	1.66	48.14	0.30	13.00	9.00
5	8523	4.90	1.64	49.77	0.28	13.46	9.85
		Median CK Metric Values					
Team Size	**Count**	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
1	68087	2	1	57	0	8	5
2	28734	2	1	62	0	8	5
3	17296	2	2	57	0	8	6
4	106966	3	2	44	0	8	5
5	8523	1	2	58	0	8	6

4.4.3 Confounding Factors

Confounding factors are those that influence both the dependent and independent variables within a model causing an association to be made which may not be genuine. In prior research modelling the impact of structural metrics on the externally observable attributes of software, class size was found to be one such factor. Emam et al. found that class size had a confounding impact on fault-proneness (El Emam et al., 2001). They suggested that earlier models which had established the predictive power of CK metrics over fault-proneness were largely (but not entirely) driven by class size and therefore not controlling for this confounding variable was a significant threat to validity. Their work was very recently corroborated by Gil and Lalouche (Gil and Lalouche, 2017) who argued that this threat to validity also extends to other OO metrics. In a similar vein, Zhou and Leung found that class-size was a confounding variable to CK metrics models where change-proneness was the dependent variable (Zhou and Leung, 2006).

This research differs from the prior literature in that the CK metrics are essentially the dependent variables rather than the independent variables. For the purposes of the validity of this analysis, it is necessary to establish those factors that could potentially confound models that use CK metrics as the dependent variables.

4.4.3.1 Class Size

The typical measure of class size is Lines of Code (or LOC). This is a simple measure of the number of lines within a class to the exclusion of comment lines (Nguyen et al., 2007). An analysis of the Spearman correlation p-values show no correlation between class-level LoC, team size and CK metrics. This is not surprising in an object-oriented language where one could reasonably hypothesise that a larger team is likely to work on a codebase which has more class files rather than larger class files per se. Similarly, LOC shows very weak negative correlations to the DIT and NOC metrics with almost no correlation to the rest of the metrics. On this basis it is accurate to say that class size has no relationship to either the dependent or independent variables in the model that was established in the previous section and therefore cannot be considered a confounding factor that should be controlled. Figure 4.12 depicts how CK metrics trend against project size showing the mean metric values across individual projects against their aggregate Lines of Code (LOC) count. These charts show that at a project level there is no obvious relationship between code volume and metric values.

4.4.3.2 Revision Counts

When using version control systems, it is usual to build functionality iteratively through

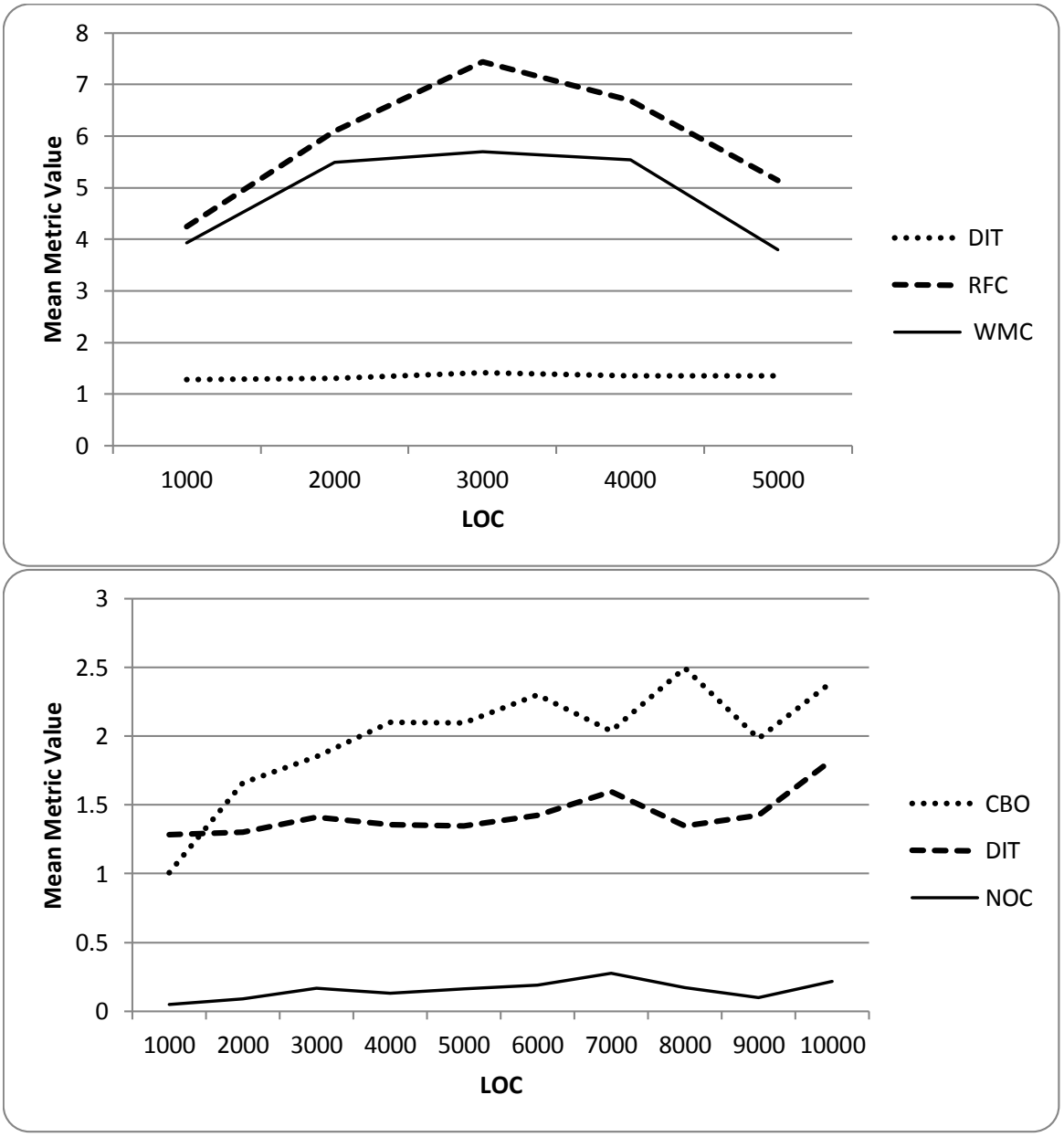


Fig. 4.12 Sample analysis of the mean project-level structural metric values plotted against the cumulative Lines of Code for that project.

repeated modification of source files. This represents the development iterations driving the evolution of the codebase. It has been proven that, as software projects evolve, iterations of a codebase tend to exhibit both growing size and complexity (Prather, 1984). Each time a file (or group of files) is edited and re-committed this is considered a single revision. Revisions are tagged with commit comments which often refer explicitly to additional functionality that relates to the commit. An example of this can be seen in Figure 4.13 which is an excerpt from the revision logs from a project called 'Precise', a requirements management tool, which will be studied in more detail in a later chapter.

oranvik	Oct 17, 2006	396	Added remaining services to the new remote service calls approach
oranvik	Oct 13, 2006	395	Improved test to include real DB calls.
oranvik	Oct 12, 2006	394	Design/Implementation of centralized service dispatching with authentication and
oranvik	Oct 9, 2006	393	Added status/user/role/service mappings. Separated out Bpot-specific concepts.

Fig. 4.13 An excerpt of the revision log from the 'precise' project showing commentary explaining the addition of functional complexity with each revision.

To date in this thesis, references to complexity have, in fact, alluded to structural complexity. This was defined earlier as the measure of the degree of interactions between components in a software system. This is in contrast to functional complexity which has no single definition but generally refers to the degree of sophistication in the logic encoded within a software system. Revision counts are an important factor to study as, this research argues, it represents a proxy to functional complexity (albeit a flawed one). While measuring structural complexity is a fairly straightforward task captured by RFC and WMC metrics (to make no mention of Halstead or McCabe's complexity metrics which are specifically designed for this purpose) capturing functional complexity is notoriously more difficult. Several measures have been proposed, generally with a tendency to conflate functional and structural complexity through attempts to track the nature of control structures within the source code or interactions between components. It is understandable that code inspection would be the default approach to measuring functional complexity as, where requirements are documented, they are often fragmented and therefore difficult interpret in an automated fashion.

Revision counts cannot be considered a perfect proxy to functional complexity as revisions will not exclusively be associated with the addition of new logic. Refactoring activities and bug fixes both necessitate revisions which would not add to the functional complexity. The efficacy of revision counts as a proxy to functional complexity is beyond the scope of this work but, nonetheless, revision counts are an important factor worthy of further study for its potential confounding effect.

Table 4.10 Sample analysis of the Spearman correlation coefficients for revisions against team size and each CK metric.

	TEAM SIZE	CBO	DIT	LCOM	NOC	RFC	WMC
REVISION	0.47	0.29	0.12	0.12	0.14	0.21	0.16

KEY	+VE STRONG	+VE MODERATE	+VE WEAK	NONE	-VE WEAK
	0.7 - 1.00	0.40 - 0.69	0.01 - 0.39	0.00	-0.2 – -0.01

Table 4.10 shows the Spearman correlation coefficients for the relationship of revisions to team size and also CK metrics. It is notable that revisions do have a marked positive correlation to team size (0.47) as well as to the CK metrics; particularly CBO and RFC (0.29, 0.21 respectively). While these correlations are weak (under the arbitrary threshold of 0.40) they are nonetheless significant and imply a linear relationship between revisions and both the independent and dependent factors in our earlier model; namely team size and CK metrics respectively.

This relationship between revisions and team size is confirmed at a project level with Figure 4.14 showing that the cumulative number of committers to a project trending positively against the number of revisions that the project has undergone. Furthermore, Figure 4.15 demonstrates a clear positive correlation between class revision counts and all CK metrics with the exception of DIT and NOC. These metrics capture a very specific facet of structural complexity - inheritance complexity - which the results imply is not correlated with functional complexity. These results are consistent with the work of Johari et al. (Johari and Kaur, 2012) who studied the relationship between CK metrics and revision counts on an open-source project.

4.5 Multivariate Team Size Analysis

4.5.1 Data Analysis Approach

Having established revision counts as a confounding factor, in order to accurately study the impact of team size and produce reliable comparisons between metric populations of varying committer count it is necessary to control for revision counts. Continuing with the same sample as previously described, the data set takes the form of a class-level and revision-level

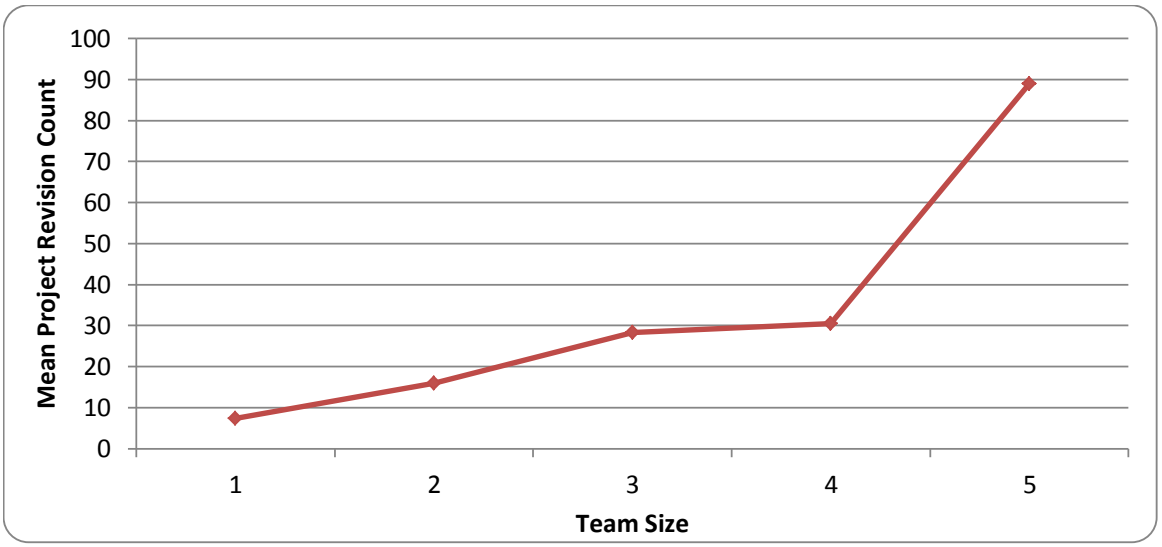


Fig. 4.14 Analysis of the sample projects showing a clear upwards trend of the project revision count against committer count.

CK metrics along with date and committer associated with each file revision as previously illustrated in 4.13. This allows the determination of the number of unique committers to an individual project which is, as defined earlier, the project development team size. The approach of conducting static data analysis at each revision of the projects in our sample is illustrated in Figure 4.16. This figure depicts the evolution of a project revised through its timeline of development with each revision the subject of static code analysis.

Given this enriched data set, metrics can be grouped together by project team size - irrespective of the individual project from which they came - and considered distinct populations. For example, if project X and project Y each had the same number of unique committers, all metrics belonging to each file within both projects would reside in a single bucket. Using this bucketing approach, a set of distinct metrics populations can be compared using statistical techniques. From the meta-data associated with each file, the number of revisions any one file has undergone can be ascertained. This data will feed into the bucketing process where the population of metrics within a particular bucket only contains those metrics belonging to projects with a particular team size and only from files that have been modified a particular number of times. This approach, illustrated in Figure 4.17, will give confidence that, when comparing our bucketed metric populations, any statistically significant differences are attributable to team size without the confounding impact of revisions.

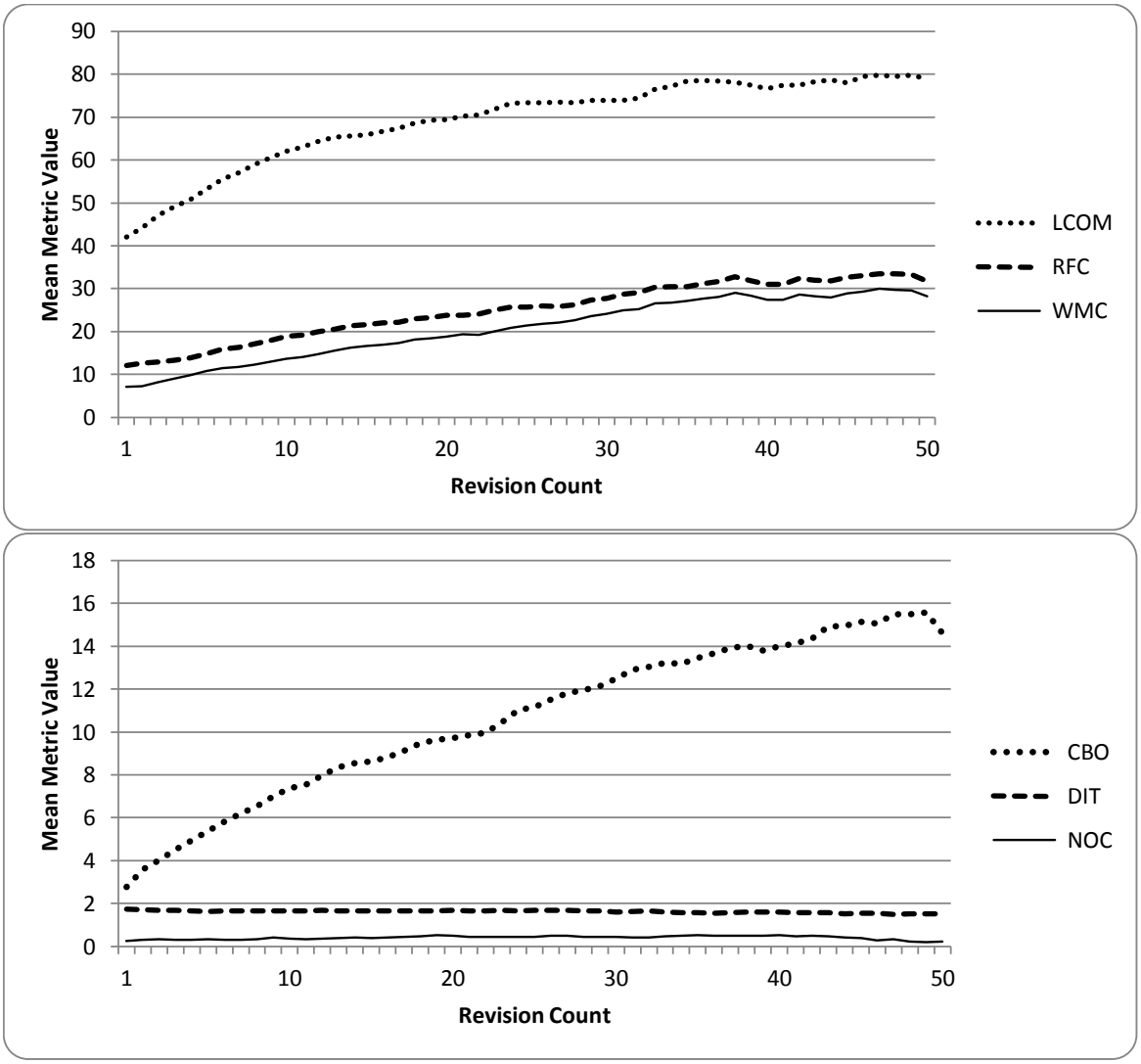


Fig. 4.15 Mean metric values at a class level plotted against the last revision count for all files within the sample.

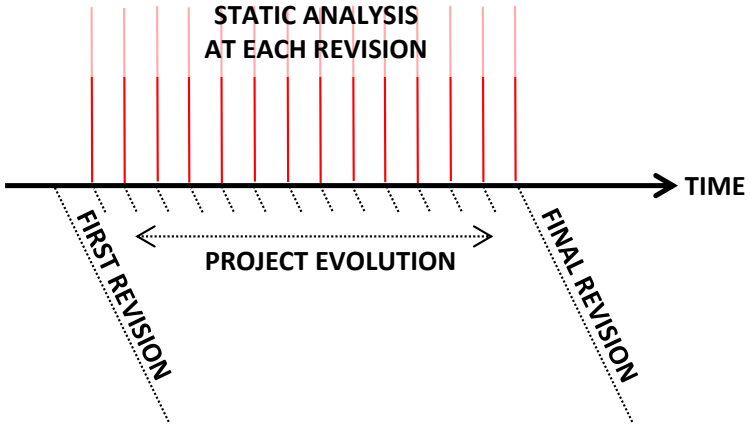


Fig. 4.16 Static analysis is conducted at each revision of the project evolution.

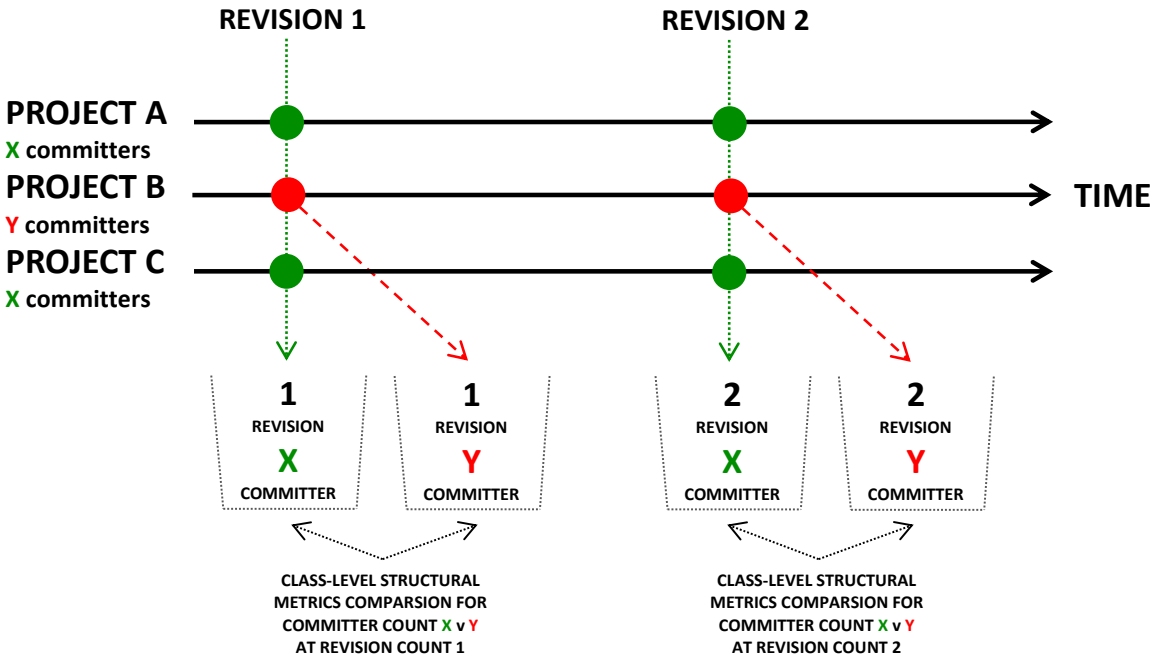


Fig. 4.17 An illustration of the bucketing approach used to categorise metrics for comparison.

Table 4.11 Bucket population sizes where each file within the sample is inspected at its final revision.

	Team Size				
	1 Committer	**2 Committers**	**3 Committers**	**4 Committers**	**5 Committers**
1 Revision	39,460	12,524	4,372	2,060	4,962
2 Revision	14,236	5,552	2,630	2,169	2,316
3 Revision	8,291	3,662	1,758	981	1,509
4 Revision	5,743	2,395	1,282	695	973
5 Revision	4,024	1,705	995	515	687
6 Revision	2,985	1,311	786	393	528

Consistent with the univariate team analysis in the previous section, as the dimensions in the data are not normally distributed, the Mann-Whitney test is again used to compare bucketed populations of metrics. Once again null hypothesis H0,1 is the subject of this analysis. H0,1 is rejected where the bucketed populations are found be independent where the p-values indicate a 95% confidence level. As 50 hypothesis tests are executed for each metric, once again the Bonferroni correction need apply. This sets the p-value threshold of α at 0.001.

It is logical that there would be a greater number of metric results pertaining to the lower revision counts as, by necessity, for a file to be revised x times, it would have x-1 prior revisions (where x>1). However, it is perfectly normal for a file to only have very few total revisions. This is an important consideration as those buckets with metric populations of higher revisions and committer counts have diminishing populations. Figure 4.18 and table 4.11 show this effect. For this reason, and to ensure suitable metric populations in each bucket, in this analysis only buckets belonging to team up to 5 committers with a maximum of 6 revisions are considered. A substantial drop in bucket population is notable beyond these thresholds.

4.5.2 Comparing Bucket Populations

Table 4.12 shows the results from the statistical tests run across each team-size comparison, similar to the previously discussed Table 4.8 but also grouped by revision. By way of explanation, the first set of rows are bucket comparisons for the CBO metric. The revision columns relate to the number of times that the classes (from which the metrics were extracted to make the bucket metric populations) have been revised. The committer count columns and

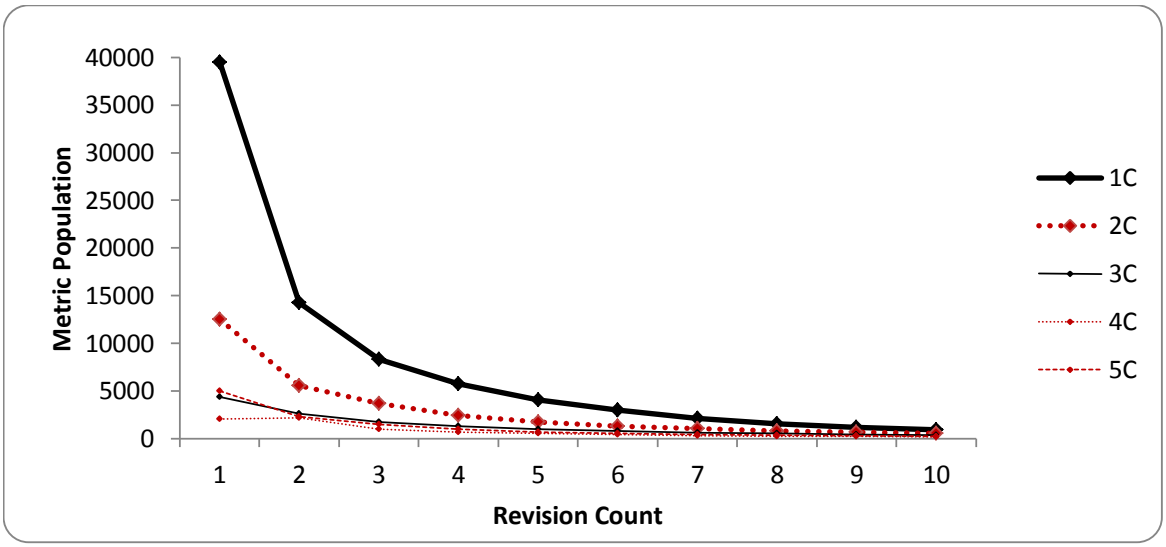


Fig. 4.18 Analysis of mean metric values against revision count where each file within the sample is inspected at its final revision.

rows refer to the number of unique committers to the project from which the class originates. For completeness, those tests that do not meet the Bonferroni corrected α cut-off but do show significance at the 95% confidence level are depicted accordingly.

Where the bucket populations are found to be significantly independent the bucket population metric means are compared. Based on this, a determination is made as to whether increasing committer count will result in a rise or a fall in the metric value.

From these results a number of notable observations can be made.

- **Rejecting null hypothesis H0,1:** After controlling for revisions it is still possible to reject null hypothesis H0,1 for all metrics but NOC and RFC. NOC showed no impact from team size before or after controlling for revision counts. However, RFC did earlier show a negative association with team size but this analysis has shown that this was due to the confounding impact of revision counts. It is evident that a large number of buckets that show statistical significance across CBO, LCOM, WMC and, to a lesser extent, DIT.
- **DIT and LCOM show a positive relationship with team size at higher revisions:** The results for DIT and LCOM show a dominant trend of increased committer count resulting in increased metric values. In the case of DIT this trend is stronger at higher revisions of 4 and above. However, DIT does the opposing trend where 1 committer

bucket is compared to 2, 3, and 4 committer buckets. Similarly, the LCOM bucket comparisons show a dominant trend of increasing metric values with increasing team size.

- **WMC shows a negative association with team size:** WMC shows a trend with a decrease in metric value accompanying an increase in team size.

4.5.3 Simple Linear Models

While the previous analysis focussed on holding revision counts constant, it is appropriate to model revision counts and committer counts as two independent variables with the CK metric values being their dependent variables, using simple linear regression to analyse their impact. Table 4.13 shows the output of this linear regression using the Ordinary Least Squares (OLS) regression method. One of the considerations when using this method for multivariate regression is that collinearity between the independent variables can lead to misleading coefficient estimates. The Variance Inflation Factor measures the increase of the variance of the parameter estimates if an additional variable is added to the linear regression. This helps ascertain the impact of collinearity of parameters on the validity of an OLS regression. At 1.65, this measure is significantly less than the 'rule of thumb' threshold of 5 (Menard, 2002), hence the collinearity between revisions and team size is not a concern.

The linear models capture the linear relationship between team size, class revision count (as independent variables) and CK metrics (as dependent variables) expressed as the following equation where team size and revision count are multiplied by their respective coefficients β, γ is the intercept, and ε is the standard error:

$$MetricValue = \beta_{TS} TS + \beta_R rev + \gamma + \varepsilon$$

R-squared is a statistical measure, ranging from '0' to '1', of how close the observed data points are to the fitted regression line. A value of '1' would imply that the independent variables linearly explain all variance of the respective CK metric.

The *coefficients* are the estimated slope (that is to say the slope based on a sample of the population) of the component of the independent variable that is uncorrelated with the other independent variable. Coefficients cannot be directly compared across independent variables

Table 4.12 Tabular summary showing the results of each bucket comparison within the sample.

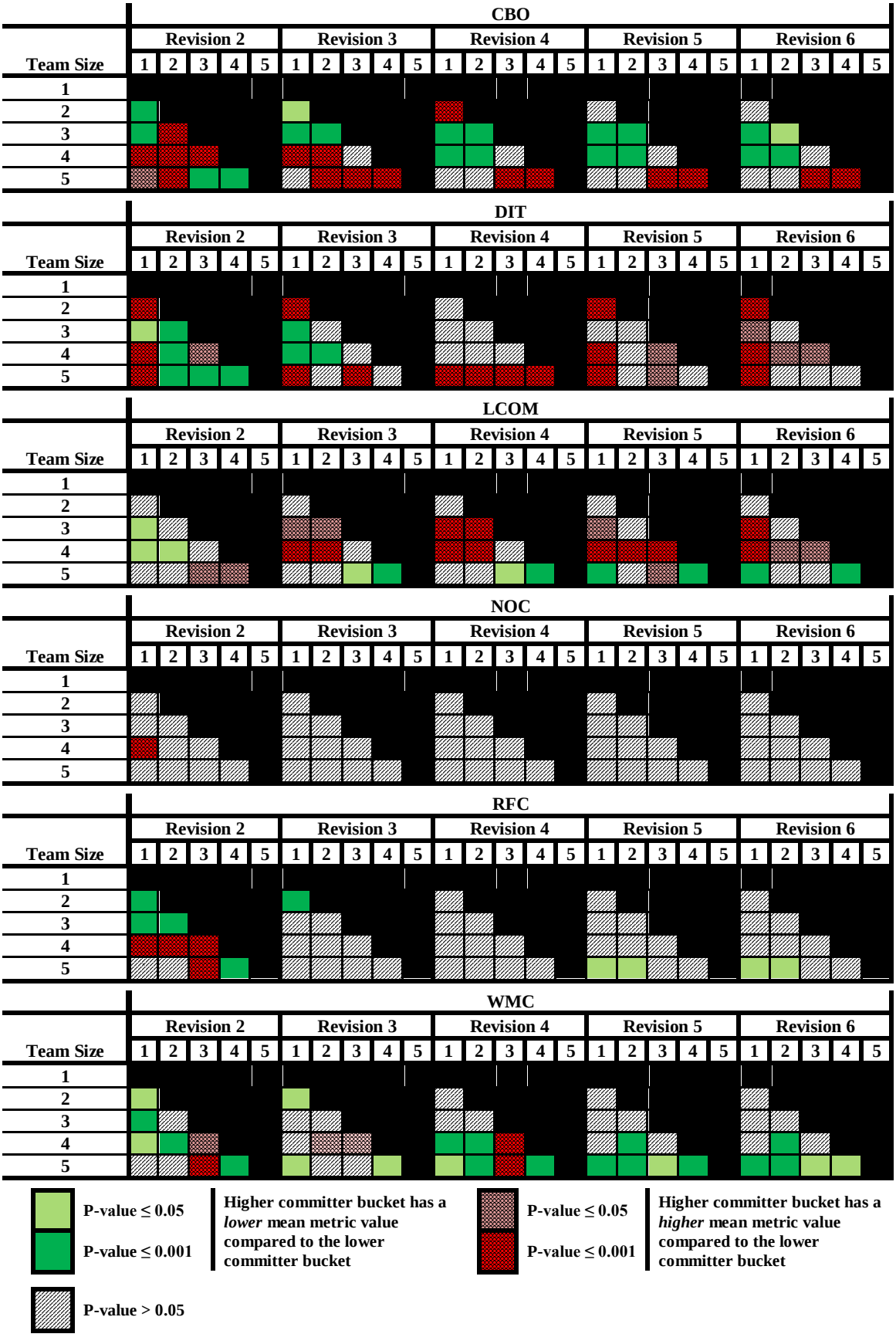


Table 4.13 The results of an Ordinary Least Squares regression using the sample dataset with committer and revision counts as independent variables.

		CBO	DIT	LCOM	NOC	RFC	WMC
Statistics	**R-squared**	0.21	0.44	0.43	0.00	0.12	0.10
Coefficients	**Team Size (β_{TS})**	0.59	0.30	7.74	0.05	2.06	1.14
	Revisions (β_R)	0.33	0.00	2.08	0.01	0.75	0.66
Residuals	**RMSE (ε)**	8.51	1.46	45.43	4.71	32.50	25.37
Intercepts	**Intercept (γ)**	2.87	1.70	45.78	0.31	10.06	6.99
Standard errors	**Team Size**	0.01	0.00	0.03	0.00	0.03	0.02
	Revisions	0.00	0.00	0.01	0.00	0.01	0.01
T-statistics	**Team Size**	91.41	275.56	225.02	15.04	83.62	59.47
	Revisions	134.90	105.05	159.80	5.35	80.41	91.35

P-values across all regressions: 0.00
Variance Inflation Factor: 1.22
Degrees of Freedom: 173,189

on the same model as they exist on different scales. For instance, if team size had the same impact on a metric value as revisions, it would still be expected that the coefficients of revisions would be lower given that they possess a higher range. Similarly it is not possible to directly compare coefficients of a specific independent variable across multiple CK metrics as here the dependent variables are on differing scales.

The *residuals* are a measure of the distance between the actual observations and the model prediction. Table 4.13 shows the Root Mean Squared Error (RMSE) which is a measure of residuals in units of the respective CK metric. Residuals are difficult to interpret in isolation but will become useful when comparing the accuracy of these models against the mixed linear models in the next section.

The *intercepts* are the values at which the fitted regression lines cross the y-axis. It is worth noting that the intercept point where team size and revisions are at '0' is purely a theoretical construct. It is useful, however, as it does indicate the degree to which the regression line is shifted upwards and will bear some relevance in Section 4.7 where intercepts of individual projects are compared.

Standard errors indicate the degree to which the coefficient estimates may vary from the sample to the full population. This is a function of the *R-squared* and the variance in the sample. *T-statistics* are the ratio of the coefficient estimate and the standard error. *Degrees of freedom* are the number of independent observations available to establish a regression model.

The simple linear models detailed in Table 4.13 show a number of key results outlined as follows.

- **Further rejection of null hypothesis H0,1:** In producing coefficients for team size and revisions, the linear modelling process first runs a hypothesis test to prove the rejection of the null hypothesis of coefficients of zero (i.e. the hypothesis that the independent variables have no effect on the dependent variables). This test returns highly significant p-values (approximated to 0.00) across all CK metrics.
- **Team size and revisions explain a substantial degree of variance, particular for DIT and LCOM:** It is notable that revisions and team size explain a substantial proportion of the variance in DIT and LCOM - 44% and 43% respectively - as represented by the R-squared values across the respective regression models. This is consistent with the results of the previous section that found that DIT and LCOM trended quite clearly positively with team size (controlling for revisions). NOC shows a R-squared value approximated to '0' which confirms the results of the spearman coefficient matrix that was presented earlier in Table **??** where no linear correlation was observed between NOC and the independent variables.
- **Inheritance complexity not impacted by revisions:** Coefficients close to zero for revisions across both DIT and NOC indicate that these metrics, both of which capture inheritance complexity, are not affected by revisions.
- **Team size trends positively with all CK metrics but NOC:** The positive estimated coefficients for team size (β_{TS}) across all regressions with the exception of NOC indicate that team size has a positive association with all but one of the CK metrics.
- **Low standard errors:** The standard errors indicate the probability that the sample mean and the population mean differ and therefore impact the estimated coefficients. The standard errors across the regression models do not rise above 0.03 indicating that, within a 97% confidence level, the coefficients would prove accurate if calculated across the full population - i.e. all projects in the Forge.

4.5.4 Linear Mixed Models

While the application of the Ordinary Least Squares regression enabled the creation of some initial models, these models attempt to fit a single regression line per metric across the entire sample, disregarding the project-by-project variation. This is at odds with prior research which tends to produce a distinct population of observations for each project and study them individually using regression techniques (for the full survey refer to Tables 2.3 and 2.5). This is somewhat intuitive as the idiosyncratic aspect of a project, driven by the nature and complexity of the functional behaviour as well as the individual programming traits of the individuals composing the team, has the potential to have a material impact on its structural metrics.

As this analysis is applied on a representative sample of an entire forge, it leads to a substantially larger data set than the 1-5 small/medium sized projects that is typical in the prior literature. As a result it is not feasible nor desirable to treat each project as an isolated data set as it would lead to a reduction in the degrees of freedom available to establish each individual linear model, consequently impacting the 'goodness of fit'.

Linear Mixed Models (LMMs) allow for a more nuanced linear regression by distinguishing between 'fixed effects' that apply to all groups and 'random effects' that apply individually to subgroups within data sets. Specifically, this approach can generate a single coefficient estimate but can allow for project-specific intercepts. This approach is useful in recognising and modelling the idiosyncratic 'project specific' impact to the metrics while using the full available data set to establish the coefficient estimates.

This is the first application of LMMs to the study of software structural metrics but there is a substantial corpus of research that uses this technique in other fields. LMMs have gained recognition in the field of genomics for allowing 'relatedness' - modelled as the random effects - to factor in genetic association tests (Zhou and Stephens, 2012). Ecologists have recently started to apply this technique to modelling random effects of space, time and individuals in the study of species diversity and extinction risk (Bolker et al., 2009).

Linear mixed models can be expressed in a similar way to the previously specified OLS regressions in the previous section where γ_p is now the *project-specific* intercept:

$$MetricValue = \beta_{TS} TS + \beta_R rev + \gamma_p + \varepsilon$$

Table 4.14 summarises the results of a mixed model linear regression using project as the grouping variable. The coefficients and residuals are detailed as previously in the OLS regression results of Table 4.13. The *sample variance* shows the the degree to which observations are spread from the mean across the sample. The *group variance* shows the same measure across individual groups and averaged across the full set of groups. This highlights the degree to which observations within a group share similar values when compared to ungrouped observations across the sample.

The following observations can be drawn from Table 4.14.

- **Sample variance is substantially higher than group variance:** With the exception of the DIT regression, the groups exhibit lower variance than overall sample. This confirms that the project-specific idiosyncratic characteristics have a significant bearing on metric values.
- **Lower Residuals:** The results from the LMMs exhibit substantially lower residuals across all metric regressions when compared to the OLS results in Table 4.13. A reduction is expected as the intercepts are defined on a per project basis explicitly to reduce residuals.
- **Higher coefficient estimates:** The higher coefficient estimates in the LMMs indicate that team size and revisions have a greater impact on metric values than otherwise apparent in the OLS regression. While all projects share the same coefficient estimate, the revised values are attributable to the greater flexibility afforded by LMMs in fitting a regression line without forcing all projects through a single intercept.

4.6 Results at a Project Level

4.6.1 Context

While the analyses of the previous sections established a relationship between team size and CK metrics, it is informative to analyse individual projects at a code level and shed light on what may be driving the broader relationships observed. This section begins with the application of dimensionality reduction to visualise the team size sample which is then used to identify two individual projects for further study. Those projects are then analysed

Table 4.14 The results of an mixed model linear regression against the sample dataset with committer and revision counts as independent variables and the project as the grouping variable.

		CBO	DIT	LCOM	NOC	RFC	WMC
Statistics	**Sample variance**	72.97	0.89	1345.27	22.13	1003.25	633.47
	Group variance	4.78	1.15	1037.15	0.07	225.69	38.41
Coefficients	**Team Size (β_{TS})**	0.66	0.53	14.26	0.06	2.92	1.88
	Revisions (β_R)	0.30	0.00	0.92	0.00	0.55	0.54
Residuals	**RMSE (ε)**	7.72	0.92	34.16	4.69	28.32	24.46

P-values across all regressions: 0.00
Degrees of Freedom: 173,189

qualitatively through observing the source code of individual class files and quantitatively through metric values and LMM regression parameters.

4.6.2 Project Selection

To drive the process of project selection it is first necessary to visualise the project sample. This would enable us to ensure that the projects selected for study are not too similar but capture the diversity within the sample. Plotting a scatterplot for a data set with two dimensions is simple as the data can directly transpose onto the axis. When there are more dimensions the process of 'dimensionality reduction' can be applied. Principal Component Analysis (PCA) is one such technique, transforming the data set to a number of linearly uncorrelated dimensions (Pearson, 1901). The 'Principal Components' are the combination of the original dimensions weighted to retaining the maximum variance within the data set. PCA does not require the individual dimensions to be of any particular distribution (Timm, 2002).

Through the application of PCA it is possible to visualise the team size data sample through two orthogonal dimensions. This process results in 'loading coefficients' which weight each dimension within the sample to derive two principal components. As shown in Table 4.15, the first principal component is weighted towards CK metrics and particularly measures of structural complexity while the second principal component shows a greater bias to cohesion and team size.

Table 4.15 Loading coefficients of the Principal Component Analysis as applied to the team size analysis sample.

	LOC	Revision Count	Committer Count	CBO	DIT	LCOM	NOC	RFC	WMC
PC-1	0.27	0.08	0.17	0.39	0.37	0.29	0.41	0.48	0.34
PC-2	-0.33	-0.10	-0.29	-0.26	-0.22	0.52	-0.25	0.18	0.56

Figure 4.19 shows a depiction of the team size sample scattered across these two principal components. No distinct clusters immediately emerge from this, but through this process it is possible to select two projects which are visually distant from one another for further study.

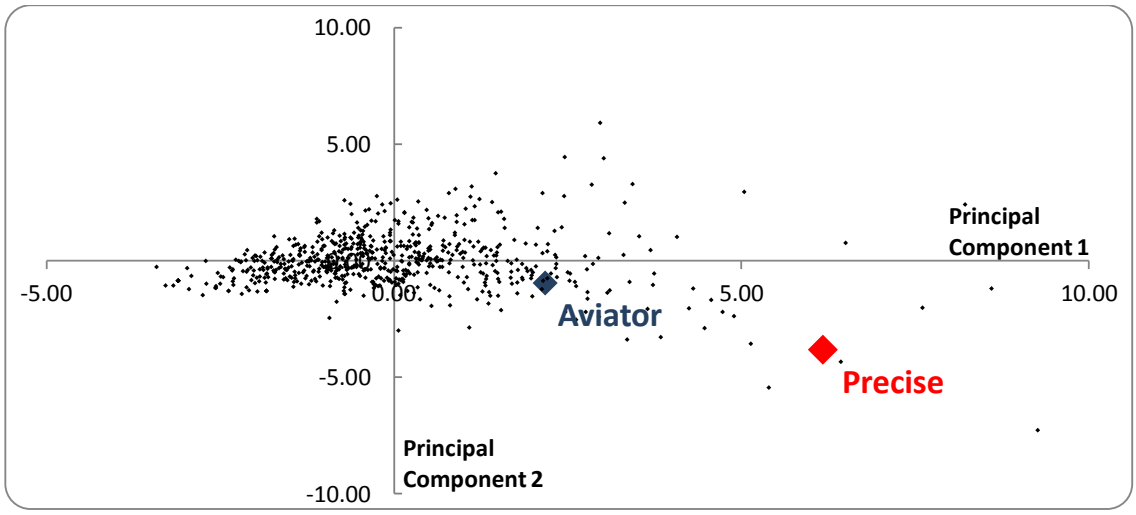


Fig. 4.19 A visualisation of the team size sample scattered across the two principal components. The selection of Aviator and Precise for further study.

Aviator and Precise are two projects that exhibit metric values that, relative to each other, are consistent with the wider forge trends for the given team sizes: one and five committers respectively. Aviator is located on the edge of the main cluster while Precise appears very much as an outlier - likely owing to its higher than mean LOC. Figure 4.21 shows a comparison of the metric profiles between these two projects plotted against revision counts. Aviator is an expression engine that dynamically compiles expressions into Java byte-code and delivers them to a running JVM. It is a single committer project with the full codebase in Java and no graphical user interface. The project comprises 233 Java class files. The committer is currently a prolific contributor to open-source projects with an active public GitHub profile. The Aviator project represented one of his early efforts started after he had accumulated roughly 2 years of development experience.

Precise is a requirements modelling and tracking tool designed to plug into the Eclipse IDE, with no dedicated user-interface as such. The project codebase comprises 524 Java class files alongside some modelling artefacts. Four distinct committers contributed to this project over a period of two years with the activity levels shown in 4.20. It is notable that one committer contributes very little commits to the project while two committers contribute the majority of commits and unique files. The definition of team size within this research does not distinguish between the activity levels of committers in assigning a value to this measure. However, the final chapter of this thesis does consider avenues of future work to assess the impact to regression models by distinguishing between core and peripheral committers in the calculation of team size.

4.6.3 Project Comparison

Figure 4.21 shows a comparison of the metric profiles between two projects plotted against revision counts. As previously, metric values are bucketed by revision count and averaged. It is notable that the observations for Precise are higher than those for Aviator for those metrics most affected by team size according to the linear regressions; CBO, DIT and LCOM. CBO and LCOM trend up with revisions, as expected given the positive coefficient estimates of the linear models. DIT does not exhibit a consistent trend against revision counts which is consistent with the earlier results which assigned a coefficient of zero to revision counts (β_R).

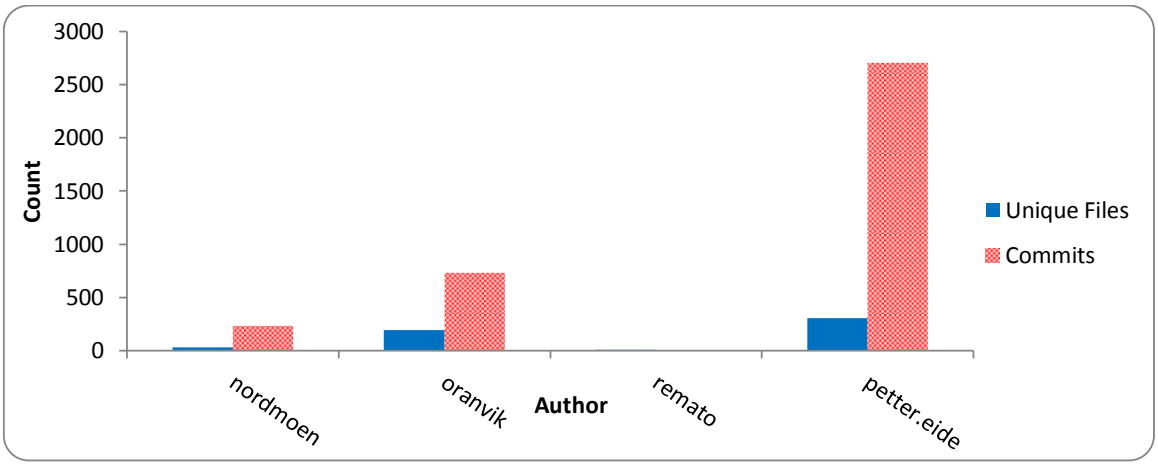


Fig. 4.20 Committer behaviour analysis for the Precise project.

Within the Precise project there is some obvious fragmentation which can be observed through a fairly dis-organised source folder and package structure. Many classes are too large

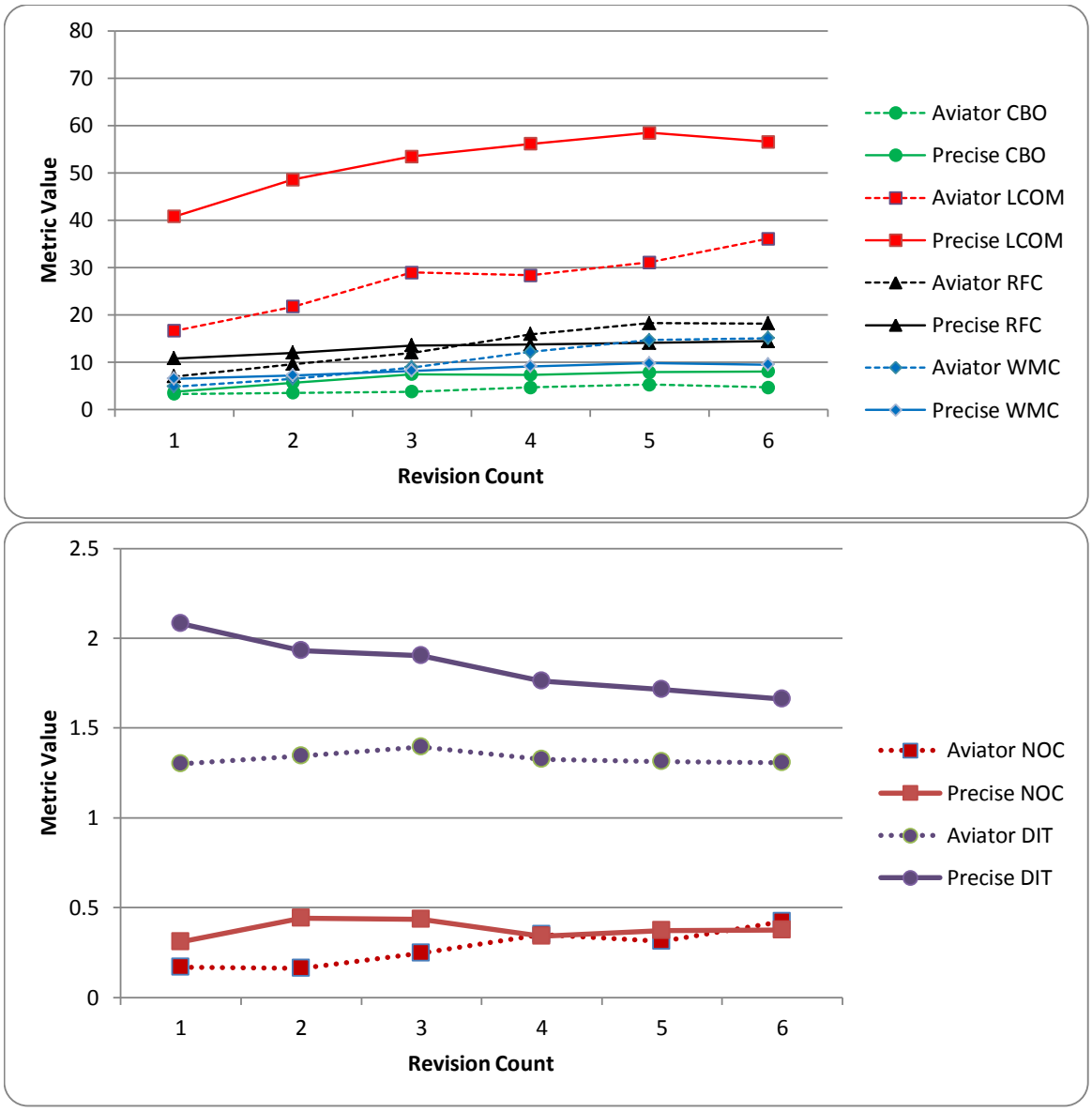


Fig. 4.21 Key structural attributes for the single contributor Aviator project compared against multi-contributor project Precise.

Table 4.16 The intercepts and residuals for the Precise and Aviator projects.

		CBO	DIT	LCOM	NOC	RFC	WMC
Intercepts (γ_p)	**Aviator**	3.13	0.79	12.23	0.08	8.02	5.93
	Precise	1.84	-0.73	-25.65	0.09	-4.38	-3.22
Residuals (ε_p)	**Aviator**	1.93	0.43	36.05	0.46	3.85	3.60
	Precise	5.71	1.05	34.82	1.19	10.58	6.69

P-values across all regressions: 0.00
Variance Inflation Factor: 2.07

with 28 classes over 250 lines long, with code violating the 'single responsibility' principle which dictates that a class should do one thing and do it well. Failure to adhere to this leads to a lack of cohesion as member variables are rarely relevant to all methods in a fragmented class. Similarly a large number of methods would lead to a high values for WMC and CBO which are capture structural complexity and modularity respectively. To illustrate, Figure 4.23 shows a snippet of code from one of the most complex classes which contains 635 lines and 35 methods, most of which contain complex functionality. In one method there are seven nested conditional blocks, indicative of inordinate structural complexity and poor modularity. The revision history of this particular class shows that it had three distinct committers and had a high degree of structural complexity from the initial creation which steadily increased over subsequent revisions.

Precise has a significantly larger and more fragmented codebase than Aviator which appears to lead to fairly large class files with multiple responsibilities and points of coupling with other classes. By way of example, the ASMCodeGenerator class has responsibilities for code generation, arithmetic operations and maintaining complex collection structures indicating poor cohesion. Furthermore, as illustrated in 4.22, it directly coupled to multiple concrete implementations within the codebase, albeit mostly functionally oriented to code generation. It is reasonable to postulate that a lack of effective coordination between committers on the Precise project lead, at least in part, to this fragmentation. This is a vicious cycle as poor structural attributes leads to further degradation as the codebase becomes more difficult to navigate and the demands for effective coordination between committers become unwieldly. All evidence is that the Precise project never made it to a fully-fledged release and ultimately failed as a project. Meanwhile Aviator continued to remain under active development after GoogleCode was decommissioned, migrating to GitHub and registering numerous releases.

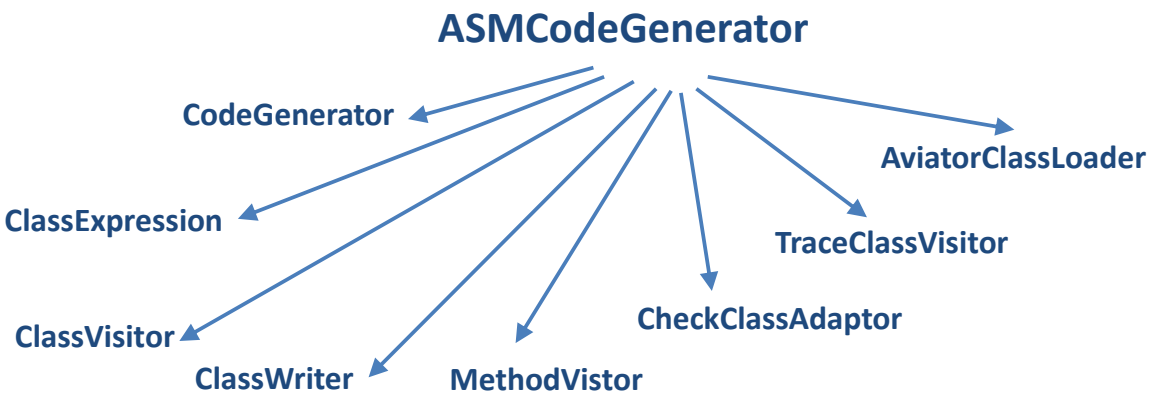


Fig. 4.22 A depiction of the external dependencies to which the ASMCodeGenerator class is coupled.

Table 4.15 shows the intercept and residual values for the Precise and Aviator projects from the LMM regression in the previous section. The residual values of Aviator and Precise are consistent with their respective positions in the scatter plot. Precise, as an outlier in the scatter plot, could reasonably be expected to have a regression line that is compromised by the nature of the majority of the projects in the sample that make up the main cluster. This is reflected in higher residual values relative to the Aviator project. Conversely, the intercept values for the Precise project are significantly higher than Aviator. This is a result of the greater distance to the y-axis when extrapolating regression lines to achieve intercepts for projects with a higher team size. Given that projects with larger team sizes will have 'further to travel' in order to intercept with the y-axis, and that all projects will have share a single gradient (the coefficient estimate) for a given metric regression, it is logical that those projects with a higher team sizes are likely to have lower (or even negative) intercepts. These observations help illustrate the strength of LMMs over OLS regression models within this context.


```
    if (null == methodsIndex ) {
        for (Class clazz : interfaces) { //TODO: check if real methods are public methods ?
            for (Method method : clazz.getMethods()) {
                MethodNamingConvention methodType = MethodNamingConvention.getMethodType(method);
                if( null == methodType)
                    continue;
                if (!MethodNamingConvention.SERVER_METHOD.equals(methodType)) {
                    Method realGetter = methodType.retriveGetterMethod(actualClass, method);
                    if( null == realGetter) // for greater flexibility regarding business methods (transient)
                        continue;
                    if (getClassMethod != realGetter) {
                        Method realSetter = methodType.retriveSetterMethod(actualClass, method);
                        if( null == realSetter) // for greater flexibility regarding business methods (transient)
                            continue;
                        allDeclaredByInterfaceToRealGetter.put(method,realGetter);
                        Method inconsistentCheck = realGetterToRealSetter.put(realGetter, realSetter);
                        if (null != inconsistentCheck) {
                            if (!inconsistentCheck.equals(realSetter)) {
                                if (inconsistentCheck.getParameterTypes()[0] != realSetter.getParameterTypes()[0]) {
                                    throw new InternalError("Inconsistency in declared methods in interfaces vs. " +
                                            "actual class. Method: "+method.getName()+", real getter: " + realGetter.getName()+ ", real setter: " + realSetter.getName());
                                }
                                log.warning("Two equal, but different setters found for the same getter." +
                                        "Method: "+method.getName()+", real getter: " + realGetter.getName()+ ", real setter: " + realSetter.getName());
                            }
                        } else {
                            if (MethodNamingConvention.isActualLinkToConcept(realGetter)) {
                                linkToOtherConcept.add(realGetter);
                            } else if (MethodNamingConvention.isActualLinkToConcepts(realGetter)) {
                                linkToOtherConcepts.add(realGetter);
                            }
                        }
                        methodType.validate(method);
                    } else {
                        log.warning("'getClass' should not be present in interfaces.");
                    }
                }
            }
        }
    }
```

Fig. 4.23 A code snippet from the DomainProxyInvocationHandler class within the Precise project. The nested iterative blocks are numerically labelled.

4.7 Summary of Analysis

To recap the analysis in this chapter, at the outset there was basic exploratory data analysis revealing underlying trends around committer behaviour and the nature of a commit as well as the distribution and correlations between the key variables in the data set. The analysis then moved on to tackling the first research question (RQ1) to establish if there was a relationship, at a forge level, between team size (the independent factor) and CK metrics (dependent factor). Then, following in the approach of the prior literature, there was a study of two factors for potential confounding impact, code size and revisions. It was determined that the former was not a confounding variable but the latter is. Revisions were then controlled for as a part of the next phase of analysis. The analysis to this point was achieved by taking observations from across projects and grouping them by team size and later by revisions, but always out of the context of the project from which they came. The latter section used the mixed models approach to linear regression to retain the project-specific idiosyncratic element to the data, re-evaluating the coefficient estimates and generate lower residuals. Finally the forge sample was visualised on a scatterplot, using PCA for dimensionality reduction, enabling the selection of two distinct projects for further qualitative and quantitative study. This helped shed light on the code-level features that drove the relationships reported by the linear models.

The results showed that projects developed by larger team sizes exhibited an increase in coupling (reflected by larger CBO values), an increase in inheritance complexity (reflected by higher DIT values) and a decrease in cohesion (reflected by larger LCOM values). This is a rejection of the null hypothesis H0,1 which anticipated no impact to any of the structural attributes of the software. Similarly, this leads us to accepting the alternate hypothesis H1,1.1, the basis of which was that prior research had linked larger teams to greater fault-proneness and, in the absence of further data, it was reasonable to hypothesise that maintainability and fault-proneness could be negatively correlated. There is consistency between the results of this team size analysis and the research of Nagappan et al. (Nagappan et al., 2008) who linked larger team sizes with increased fault-proneness given that Basili et al. had confirmed that CBO and DIT is highly correlated with fault-proneness (Basili et al., 1996); this research having also observed higher CBO and DIT values from larger team sizes.

Referring to Table 2.5 (the survey of research correlating CK metrics to the sub-attributes of maintainability), inferences can be drawn from the observed structural trends against the impact on the maintainability of the software. Bruntink et al. had observed that DIT was negatively correlated with the testability while Badri et al. noted the same relationship

between LCOM and testability (Bruntink and van Deursen, 2006; Badri et al., 2011). Harrison et al. found that LCOM was negatively correlated with changeability and Elish and Rine found that all CK metrics were negatively correlated with stability (Harrison et al., 1998b; Elish and Rine, 2003).

Based on this prior research it can be deduced from the results of this chapter that the increased coupling, inheritance complexity and decreased cohesion of software associated with larger team sizes - represented by the higher values of CBO, DIT and LCOM respectively - result in degraded levels of the maintainability sub-attributes of testability, changeability and testability. This is an acceptance of alternate hypothesis H1,1.2 and is depicted in Figure 5.27.

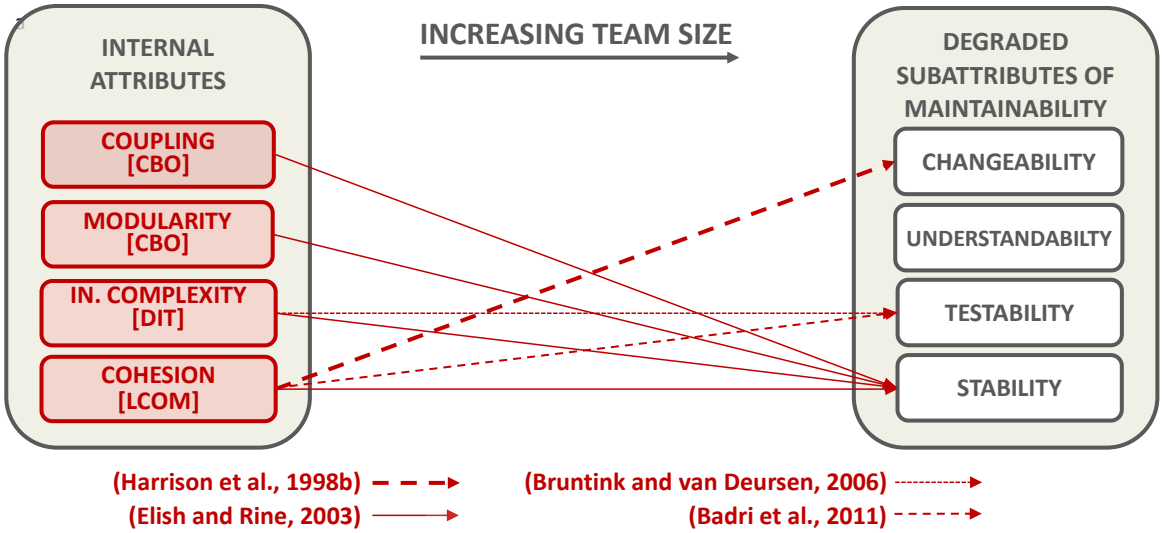


Fig. 4.24 The impact of structural attributes on the maintainability of software. Coupling, cohesion and modularity and inheritance complexity all trend in a direction that indicates an attendant degradation in testability and stability and changeability.

4.8 Chapter Review

This chapter studied the relationship between development team size and the CK metrics of the produced software. This was done by first sampling the broader GoogleCode forge followed by exploratory data analysis studying committer behaviour and comparing the sample to the broader forge. Simple linear models were produced and the confounding factor of revision counts was identified. The multivariate analysis introduced revision counts into

the linear model and, through studying individual projects in detail, the nature of the linear relationship was established.

The next chapter uses a similar approach to study the impact of team stability on the structural attributes of software. A distinction will be drawn between the team stability that is developed through the commit history of a project and the team stability that accrues across multiple projects.

Chapter 5

The Impact of Team Stability on Structural Metrics

5.1 Introduction

The previous chapter focused on impact of team size on the structural metrics of software, addressing the first research question (RQ1). In this chapter, the focus turns to the second research question (RQ2): *the impact of the development team stability on the internal structural metrics of coupling, cohesion, complexity, and modularity of software projects and the implications on its maintainability*. Consistent with the previous chapter, a representative sample is first extracted and a measure of team stability is then presented. This is then used to drive a series of statistical analyses to answer the research question. At the outset of this chapter it is useful to restate the basic definition of team stability as the cumulative time that each team member has worked with their fellow team members. Consistent with the previous chapter, the definition of the team remains the set of unique committers present in the revision history in the version control system of a given project. This initial basic definition of team stability will be developed and expanded upon through the course of this chapter.

Figure 5.1 depicts the structure of this chapter starting with an initial treatment of those aspects of data mining and analysis that are foundational the team stability analysis which is detailed in the latter sections of this chapter. Section 5.2 outlines the approach to conducting network analysis throughout the GoogleCode forge; this is necessary to calculate a reliable

measure of team stability. The pitfalls associated with forked projects and multiple committer identities are documented, along with mitigation strategies to these threats to validity. Section 5.3 is concerned with the second strand of foundational work - data mining and preliminary forge analysis to discover those basic trends that have a bearing on the latter analysis phase. Section 5.4 documents the approach to sample extraction necessary for the subsequent analysis in this chapter. Section 5.5 is a study of the impact of **'intra-project team stability'** on structural metrics; *that is assessing how the stability accrued through the course of the evolution of the project affects its structural metrics, observing the project's final archived state within the forge*. Section 5.6 assesses the impact of **'inter-project team stability'** on structural metrics; *that is the stability that is gained from retaining committers in a development team across multiple projects*. Inter-project team stability analysis factors in the chronology of projects and makes observations of the impact of stability that accrued in previous projects on structural metrics of the team's subsequent projects. Fortunately, given the breadth of the forge it is feasible to focus exclusively on the study of projects where an entire development team collaborated on a project and subsequently migrated, with the introduction of no new committers, to a later project. A comparison is carried out between the structural metrics of the chronologically earlier project against the later projects and, as in the previous chapter, functional complexity is isolated to remove any confounding impact it may have on the analysis. As in the previous chapter, section 5.7 presents the results in the context of two individual projects.

Figure 5.2 highlights, at a high level, aspects of the toolchain that are of relevance to the team stability analysis - aspects that will be expanded on later in this chapter.

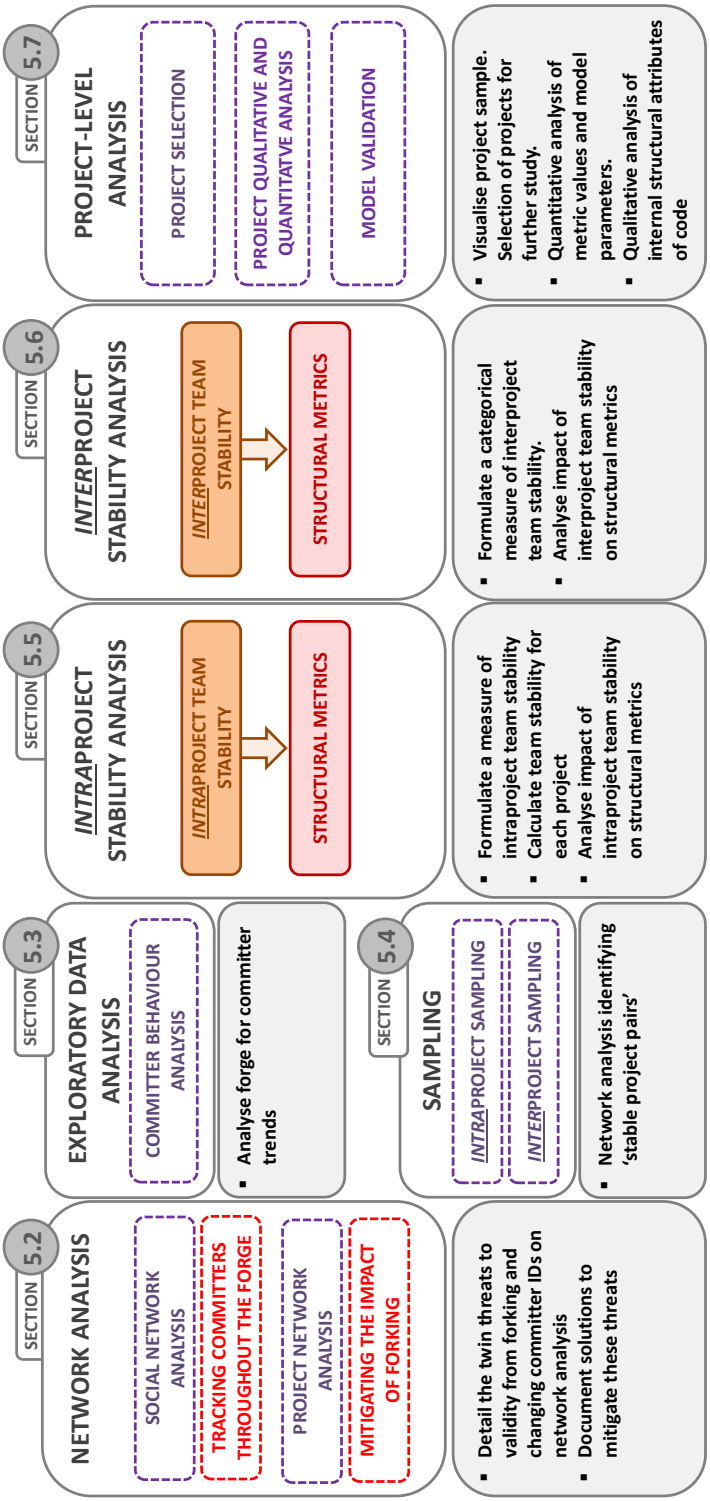


Fig. 5.1 Chapter 5 outline providing an overview of the contents of each section.

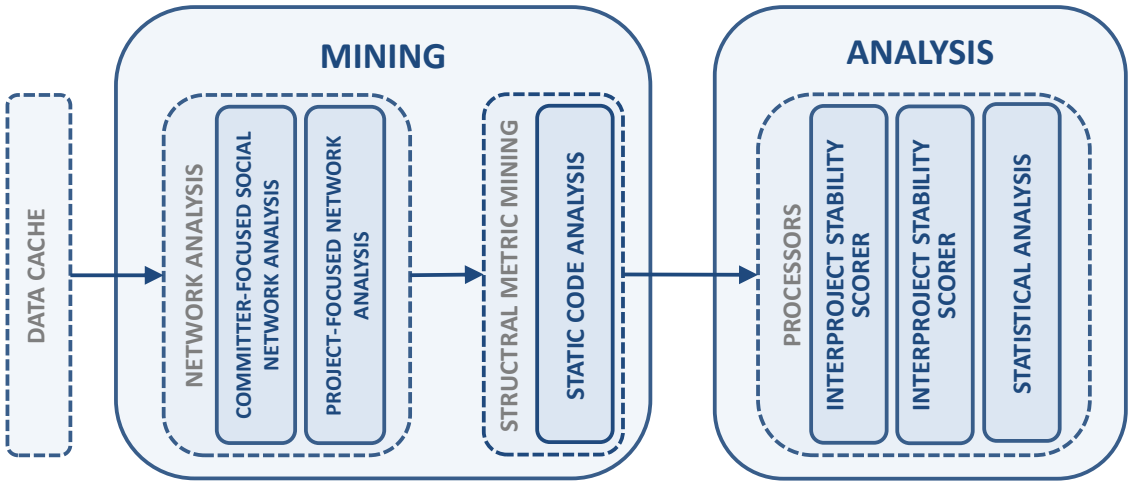


Fig. 5.2 Aspects of the toolchain pertinent to team stability analysis.

5.2 Network Analysis

Network analysis is the process of mapping and measuring relationships between entities. This is a broad subject and there is substantial related work in the field of software development centred on project contributors as the entities to be mapped, usually focussing on studying communication between entities to understand the impact of various organisational dynamics (Howison et al., 2006; Martinez-Romo et al., 2008; Conaldi and Lomi, 2013; Daniel and Stewart, 2016; Bernardi et al., 2018). This research is concerned with utilising network analysis techniques to make accurate observations of team stability that can then be used to drive comparisons between sets of structural metrics.

This research necessitates two types of network analyses illustrated in figure 5.3. The first is a social network analysis focussed on committers where every single commit is mined and mapped to its respective project and the nature of the engagement of each committer's project engagement is analysed within the context of their fellow committers. This is used to calculate the stability of a team through the evolution of the project.

The second type of network analysis - project-focused network analysis - is concerned with establishing the relationship of projects to one another. This relationship is given rise through the process of 'forking' creating, in essence, a dependency network. As will be discussed in this section, this relationship can distort analysis of committer history hence is it critical that this hierarchy is mapped out in order to eliminate this issue as a potential threat to validity.

The project-focused network analysis in this chapter does not consider directionality although this will be discussed in the final chapter as a potential future avenue of work.

This section discusses each of these two types of network analyses separately, their associated mining challenge and the strategy to solve for those challenges.

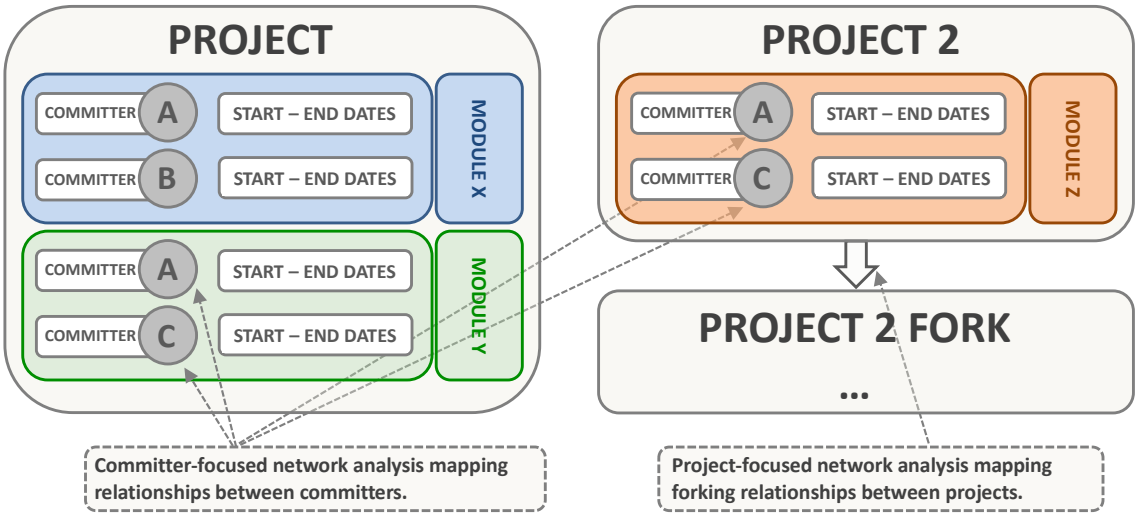


Fig. 5.3 A depiction of the network analysis conducted in the GoogleCode forge.

5.2.1 Committer-Focused Social Network Analysis

The social network analysis strand of this research is concerned with mining the commit history of every committer in the forge, identifying which projects they contribute to, capturing the detail of their project engagement and analysing each committer's project engagement within the context of their fellow committers on those projects. The project engagement detail captured comprises the sub-modules that committers contribute to and the time interval of their contributions. This analysis will provide a basis for the calculation of intra-project team stability to be discussed in more detail in section 5.4.

A challenge to this network analysis is building the capability to consistently track committers as they traverse through the forge. When determining contributor activity, it is noted that

multiple user identifiers are occasionally used by the same committer. Without rationalising these to a single identifier it is not possible to effectively track a committer's behaviour. As this research seeks to accurately establish the composition of development teams across all the projects in the forge in order to establish instances where groups of two or more committers contribute to more than one project together, it is essential to reliably identify committers across multiple projects. This is a common problem in the field of mining software repositories and has seen some earlier research efforts. Robles and Gonzalez-Barahona developed a methodology and general heuristics to identify developers across repositories (using data from VCS, mailing lists, and bug reports) (Robles and Gonzalez-Barahona, 2005). They classify email addresses as a 'primary identity' which is almost always present across diverse repositories and they present a general approach to extract identities from email addresses. Applying this approach on the GoogleCode forge, it is observed that multiple email addresses can be attributed to the same identity.

To illustrate by way of example consider the two IDs below which, for the sake of this example, appear in the same project:

ID1: Jane Doe <jane.doe@doe.com>
ID2: Jane Doe <jdoe@doe.com>

Robles and Gonzalez-Barahona argue that it is a reasonable assumption that both these user handles refer to the same committer, albeit from two separate user accounts (Robles and Gonzalez-Barahona, 2005). If this is not accounted for in the team stability analysis later in this chapter, this could result in an incorrect determination of the degree of stability attributed to a project.

Fortunately multiple user IDs for a single contributor usually carry sufficient similarity to allow automatic detection and rationalisation of these user IDs. Bird et al. propose a heuristic to rationalise user IDs when mining email social networks (Bird et al., 2006) which is partially adapted and adopted for this purpose. Specifically, there are several stages of name parsing that are applied:

- Where a name is accompanied by an email address in brackets the email address is ignored. In this case one user ID would be selected to which we assign the commits of both user IDs.
- All names are converted to lower case and all trailing spaces are removed.
- All punctuation is removed and replaced with a single space.

- Multiple spaces are replaced with a single space.

When this type of network analysis is applied to the GoogleCode forge, it is observed that 17% of unique committer identities are indeed sufficiently similar to be considered subtly different identities referring to the same committer. Naturally, activities on these alternate identities are conflated to represent a comprehensive view of the committer's behaviour.

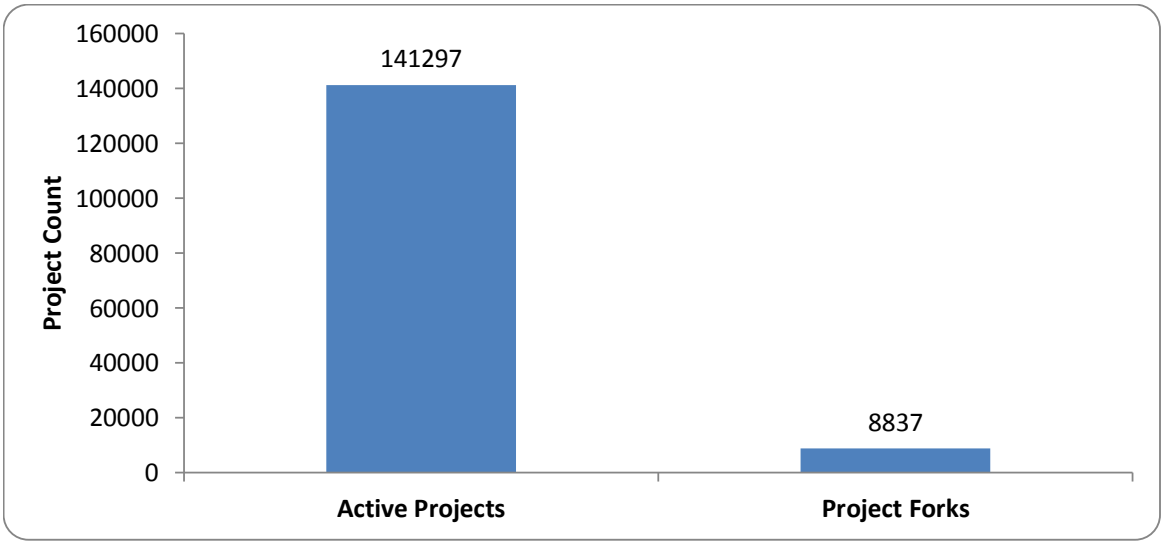


Fig. 5.4 A depiction of the number of committers in the GoogleCode forge pre- and post-analysis.

5.2.2 Project-Focused Network Analysis

The project-focused network analysis within this research is concerned with mapping out the parent-fork relationships between projects. Forking refers to the process of creating an alternate and independent software development stream from an existing project. When unmapped, these relationships can pose a real threat to validity due to the duplication of commit history in the child project which can erroneously reflect those committers contributing to the parent project also contributing to the child project. This research seeks to study inter-project stability by studying projects where groups of committers genuinely contributed to multiple projects. Therefore this type of network analysis is necessary to avoid misattributing duplicated commits from forked projects to committers who have not genuinely contributed to those projects.

5.2.2.1 Mechanics of Forking

One of the strengths of open-source software development is the ease with which one can start a new project leveraging all the tools that an open-source forge can offer. An unsuccessful project will attract few additional contributors while a successful project will build an active development community, produce artefacts and ultimately an active user base. Open-source forges also simplify the process of starting new projects based on the source code of existing projects and without affecting the original. The motivations to do so can range from discontent at the direction of development in the 'master' project through to a desire to build or experiment with new ideas within a mature project. The process of forking is named as such after the concept of forking processes to execute in parallel.

Forking projects can take multiple forms and each presents its own challenges in terms of automated detection. For the avoidance of doubt, when 'forking projects' are referred to, this is distinct to the 'fork and pull' development approach adopted by repositories such as GitHub, studied extensively by Kalliamvakou et al. (Kalliamvakou et al., 2014). 'Fork and pull' is the process of cloning a master repository into an individual contributor's personal repository which acts as a staging area before changes, after review, can be merged into the master repository. This research, however, is concerned with forked projects that represent an entirely different development stream to the original project.

The process of creating a fork of a project can take several forms. This section discusses each approach illustrating using examples from GoogleCode.

- **Copying source files or binaries** Some developers choose to fork projects by copying selected elements of the parent project's codebase. This could, for example, take the form of copying selected pre-compiled binaries or indeed the entirety of the source code. The example depicted in Figure 5.5 illustrates a snapshot of the revision history of the 'iTerm2' project - a MacOS Terminal replacement - which is a fork of the iTerm project. The second of the two revisions (r2) shows an import of the source files of the parent project. At the heart of this threat is that there is essentially no way of determining if a committer's contribution is their original work or whether it was fully or partially copied from other projects or sources without mining a host of other forges. While this is not a serious threat to the validity of this research, this could impact those researchers attempting to understand the value of developer contributions and would necessitate a mechanism to distinguish between a committer's original source code and that imported from other projects. For this reason, among others, development of techniques and software to identify 'code cloning' is an active field of

research (Roy et al., 2009). While Brixtel et al. presented a framework which could be deployed for the purposes plagiarism detection (Brixtel et al., 2010), the majority of the efforts in this field focus on gaining greater understanding on the evolution of software repositories and reducing the wasted effort arising from duplicate code. Juergens et al. created an open-source workbench for code detection research geared towards configurability and extensibility and hence designed to support code clone research (Juergens et al., 2009). Lee et al. worked designed algorithms to support scalable indexing structures on vector abstractions of code to allow for the rapid detection of clones (Lee et al., 2010). This work will be discussed towards the latter part of this thesis in the context of possible future work.

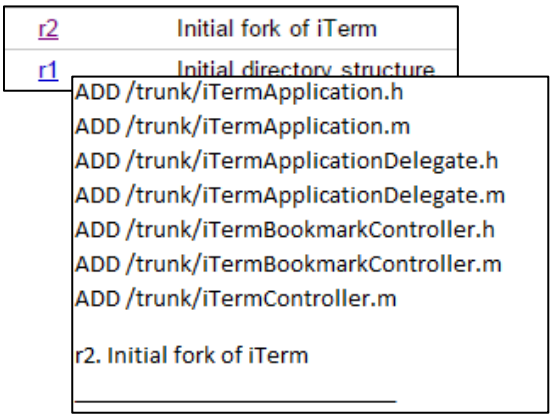
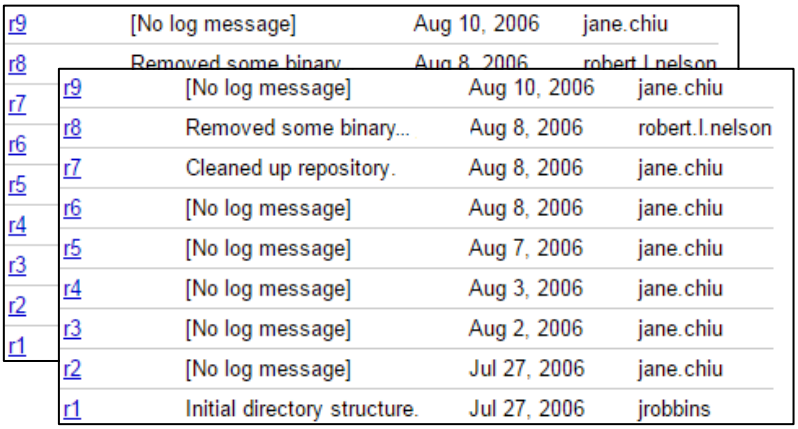


Fig. 5.5 A snapshot of the start of the commit history of the 'iTerm2' project.

- **Cloning a repository within the forge** Most popular version control systems make it fairly easy to clone a project, along with its full commit history into a new repository. Where this approach is taken to create a new fork, it will have an identical history to its parent. Figure 5.6 shows the identical VCS history of the 'hotcakes' and 'zumastor' projects - both enterprise network storage solutions. In this example both projects share the same first 239 commits, after which they take divergent development paths. When conducting network analysis across the forge, without factoring in this threat to validity, it may be falsely observed that the committers of 'hotcakes' later contributed on 'zumastor'.

- **Cloning a repository from outside forges** This approach is a subtle variation on the previous method. Figure 5.7 depicts the commit history of the 'cacheboy' project repository - a fork of 'squid' which is a webserver caching solution. Commits predate GoogleCode's launch by 9 years so clearly cannot be part of the 'cacheboy' project



r9	[No log message]	Aug 10, 2006	jane.chiu
r8	Removed some binary...	Aug 8, 2006	robert.l.nelson
r7	Cleaned up repository.	Aug 8, 2006	jane.chiu
r6	[No log message]	Aug 8, 2006	jane.chiu
r5	[No log message]	Aug 7, 2006	jane.chiu
r4	[No log message]	Aug 3, 2006	jane.chiu
r3	[No log message]	Aug 2, 2006	jane.chiu
r2	[No log message]	Jul 27, 2006	jane.chiu
r1	Initial directory structure.	Jul 27, 2006	jrobbins

Fig. 5.6 The identical VCS history of the 'hotcakes' and 'zumastor' projects.

itself. These commits do, in-fact, belong to the 'squid' project which was hosted on a repository external to the GoogleCode forge. The existence of a parent outside the open-source forge will prove a challenge to detect. The threat to the validity of the network analysis in this case is the erroneous attribution of committer activity to a fork rather than a parent that resides outside the forge. This threat is of no consequence to this research given that the parent resides outside GoogleCode, there is no threat of 'double counting' the committer contribution; the risk is solely that the aggregate activity across the two projects is assigned to the later forked project.

r9	misc	Feb 21, 1996
r8	adding	Feb 21, 1996
r7	adding	Feb 21, 1996
r6	This commit was manufactured by cvs2svn to create branch 'squid_1_0'.	Feb 21, 1996
r5	Initial checkin, duh.	Feb 21, 1996
r4	This commit was manufactured by cvs2svn to create branch 'squid_1_0'.	Feb 21, 1996
r3	This commit was manufactured by cvs2svn to create branch 'unlabeled-1.1.1'.	Feb 21, 1996
r2	Initial revision	Feb 21, 1996
r1	New repository initialized by cvs2svn.	Feb 21, 1996

Fig. 5.7 The VCS history of CacheBoy.

5.2.2.2 Identifying Forked Projects

Identifying forked projects is an area which has seen some prior academic interest. Nyman and Mikkonen conducted research to establish the most common motivations for forking within SourceForge (Nyman and Mikkonen, 2011). The methodology to identify forked projects was to search the project descriptions that referred to forking. Although this approach

suffices when attempting to locate a sample to study, relying on developers to specifically declare a project as 'forked' in the description does not help us identify a comprehensive set of forked projects to facilitate an accurate network analysis. Similarly, Robles et al. (Robles and González-Barahona, 2012) used a fairly manual approach for locating significant software forks that involved searching Wikipedia using the term 'software fork' and manually navigating to the project homepage to extract key information ahead of a study on the motivations and outcomes of forking. Again this cannot be applied to the large-scale mining of software forks within open-source forges making this approach inappropriate to identifying a comprehensive set of forks.

As part of the process of maintaining a forked project, it is often desirable or indeed necessary to import changes from the master project. Ray and Kim developed a tool called REPERTOIRE to automate the identification of commonality between known forked projects through comparison of source files but it does not attempt identify forked projects in a wider open-source forge (Ray et al., 2012). Of the approaches to automated clone detection, this is closest to the methodology adopted in this research. While REPERTOIRE tracks activity between known forked projects, this research is focussed on identifying the forked projects the co-exist within a forge. As such, this research can be considered complementary to the work of Ray and Kim.

5.2.2.3 Identification Heuristic

This research proposes a heuristic which searches for common commits across projects and identifies them as projects exhibiting a fork relationship. This heuristic searches for common commits across projects and identifies them as projects exhibiting a fork relationship. This approach is only capable of detecting forks where the parent resides in the same forge, and has been cloned, retaining the revision history of its parent. Therefore, if we capture both the parent and its forks within GoogleCode we would expect to see commits of identical lists of affected files, date and author across multiple projects. Using this approach we are able to determine networks of linked projects as the forks are identified in the context of their related projects rather than in isolation as is the case with the other heuristics.

It is important to note that very simple changelists (defined as the set of files affected by a commit) may appear across a large number of unrelated projects. For example, most Maven projects will contain changelists which contain a modification to the pom.xml file (the key configuration file in such projects). In order to eliminate these false positives, one important parameter in this heuristic is the minimum number of files which a changelist has to contain

in order to be considered for duplicate analysis. For the purposes of our work, a size of *ten* has been found to eliminate false positives without unduly restricting the result set.

Limitations to the heuristic are outlined below.

- **Coverage:** This approach is only capable of detecting forks where the parent also resides in the forge (or set of forges) being analysed; i.e. the mechanism of forking earlier referred to as 'cloning a repository within the forge'. This is not a limitation within the context of this research as this analysis is concerned with identifying those inter-project relationships that arise as a result of forking projects within the forge. This is because this forking mechanism was earlier identified at the only mechanism that has the capacity to create a distortion to the type of committer-focused network analysis conducted in this research.
- **Forks by source file import:** As detailed earlier, some projects are forked by importing binaries and source files from other projects rather than importing the version control commits themselves. These forked projects are not detected by this heuristic but may be found by the other heuristics if the project or commit description contains the term fork. Again, this is not a limitation in the context of this work as these other forking mechanisms do not pose a threat to the validity of the network analysis in the latter parts of this chapter.
- **Context limitations:** This heuristic does not attempt to establish which, within a forking relationship, which project is the parent and which is the child. While establishing this relationship can be informative, as will be outlined in the Discussion chapter, it is not necessary for the purposes of the committer-focused network analysis in this research.

Figure 5.8 shows the results of this analysis. 6.25% of projects in the GoogleCode forge are forks of parent projects with a revision history that contains commits that are duplicated from the parent project.

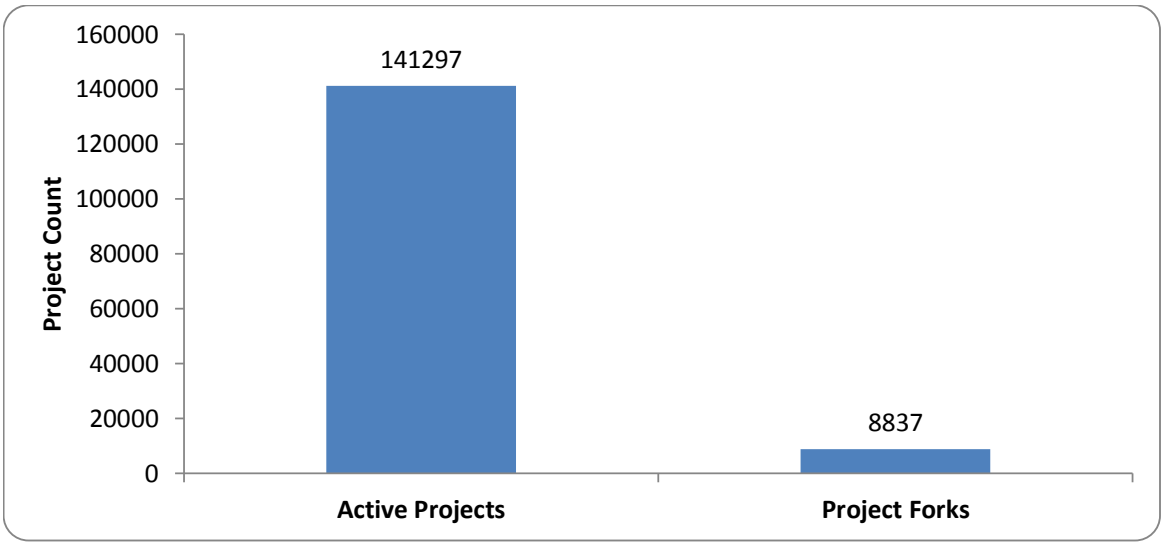


Fig. 5.8 A chart comparing the total number of active projects (i.e. those projects with commits beyond the initial repository creation commit) to the number of project forks.

5.3 Exploratory Data Analysis

This section studies basic committer behaviour in order to analyse typical project engagement. This basic preliminary analysis will be valuable when conducting network analysis to calculate team stability.

Figure 5.9 depicts the number of distinct projects that individual committers engage in, plotted against the log of the number of committers with that project engagement count. While 87% of committers engage in a single project only, there are substantial numbers of committers that engage in multiple projects. This is significant as it indicates the availability of multiple committers engaging multiple projects - something that is critical for the identification of the inter-project team stability analysis data set as discussed in the next section.

Figure 5.10 is a representation of the duration of committer project engagement. This is calculated as the elapsed time (in units of days) between the first and last commits of individual committer on a project and is plotted against the frequency of that duration across all committer project engagements. It is observed that the majority of project engagements have a duration of eleven days or more. This has implications for the accumulation of intra-project team stability and, again, makes it likely that significant time overlaps between committers within a single project will be observed. As will be discussed later in this chapter,

this is necessary for the accumulation of intra-project stability through the evolution of a project.

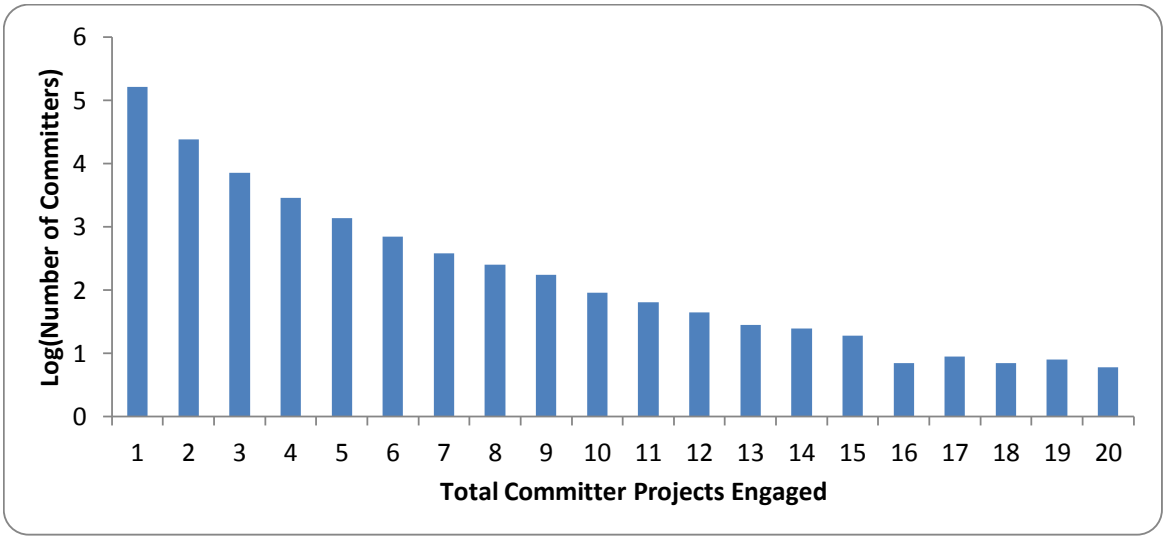


Fig. 5.9 The number of projects individual committers contribute to throughout GoogleCode.

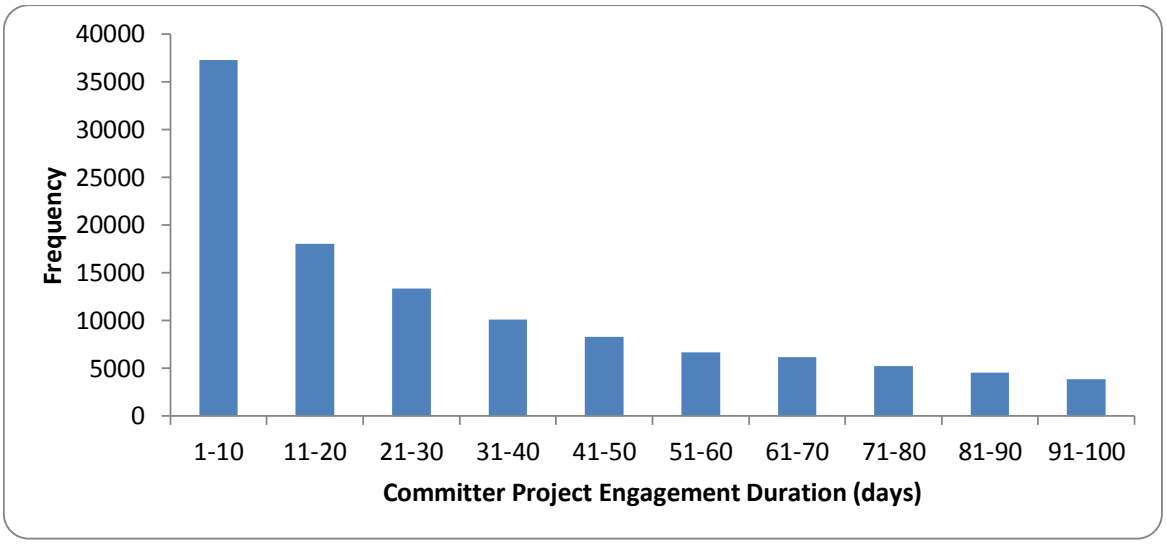


Fig. 5.10 The frequency of the timespan of project engagement - measured as the time between the first and last commits on projects by individual committers.

5.4 Sampling

The network analyses detailed earlier underpins the team stability analysis which is necessarily based on an accurate and comprehensive picture of committer project contributions and forge traversal. Where **intra**-project stability is studied, it is necessary to map the individual committer contributions at the project level. This type of analysis places limited demands on the requisite data set. Projects included in the data set should have more than one committer as there can be no meaningful analysis unless there are multiple committers. As with the team size analysis, it is possible to continue to restrict the study to Java projects only without unduly compromising the sample size.

Inter-project stability analysis places greater demands on the data data-set and essentially should comprise pairs of projects where multiple committers migrate from a project, potentially shedding committers but not gaining any new ones, onto a new project. In order to provide contrast between the later project where committers have established some stability in comparison to the earlier project where the team stability would have been less, there should be no time overlap in the commit activity of the two projects. This criteria is illustrated in figure 5.11 below. The earlier project network analysis ensures that we reliably identify these project pairs without the distortion that forking can bring.

The network analysis discussed in section 5.2 enables the accurate and comprehensive mapping of committer project engagement throughout the forge. This facilitates the identification of projects where committers have contributed to more than one project alongside the same fellow contributors. This forms what is termed in this thesis a *'stable partnership'*; i.e. two or more committers contributing to two or more of the same projects. This stable partnership is associated with a 'stable project pair'; this is to say that pair of projects where the 'stable partnership(s)' manifest. This is a simplification as there could be more than two projects where these partnerships appear. For the purposes of ascertaining the impact of stability this does not constitute a threat to validity, rather it is an opportunity for further work as will be discussed in the next chapter.

The expansive nature of the GoogleCode forge enables the application of fairly specific criteria to the identification of the inter-project analysis data set while retaining a statistically significant sample size. As illustrated in figure 5.11, only project pairs where the commit history of the earlier of the projects concludes before the later project commences are included in the data set. This is to ensure that stable partnerships are captured rather than overlapping partnerships which could detract from the impact of the team's stability. Secondly, only those

project pairs with committer populations entirely made of stable partnerships are eligible for inclusion; that is to say that the committers in the later project in the pair should be made up entirely of contributors from the earlier project. This is to avoid considering projects where the inter-project team stability is reduced by newcomers to the team. Finally, as discussed earlier in this chapter, the population sample must be cleansed of forked projects with duplicated commit history as this is essentially distorted information which would constitute a threat to validity.

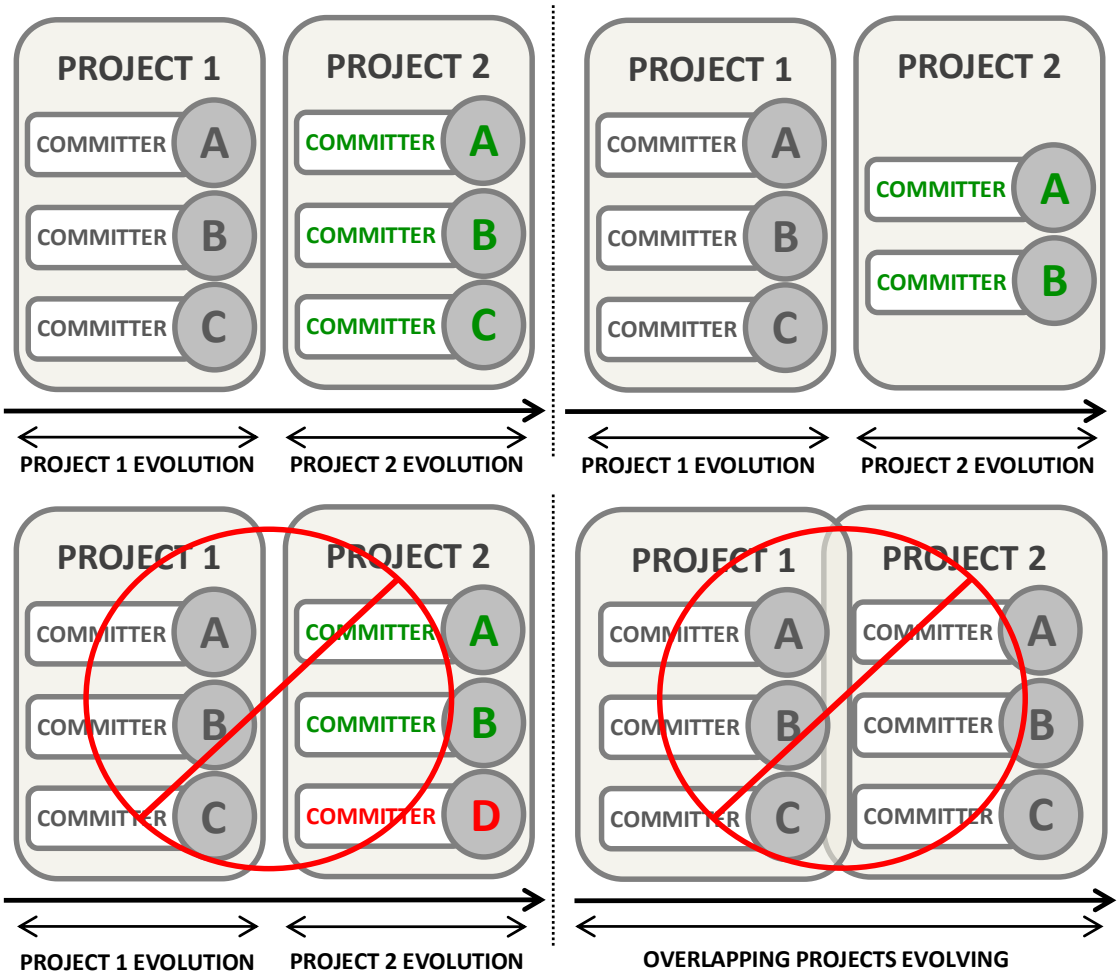


Fig. 5.11 An illustration of the 'project pairs' that are eligible for inclusion in the intra-project stability analysis. While shedding team members from one project to the next is acceptable, additional committers is not. Projects must not overlap.

This comprehensive criteria enables the creation of two distinct data-sets which can then be analysed relative to one another; projects where committers have previously partnered together 0 to n times against those later projects where they have partnered 1 to $n+1$ times.

By contrasting the structural metrics of each of the earlier projects within a stable project pair against the later project within that pair, it is possible to make some observations on the impact of team stability on these structural attributes of these projects.

For simplicity, once the data set for the inter-project stability analysis is identified, that same data set is leveraged to drive the intra-project stability analysis where projects are studied in isolation as opposed to within a their project pair.

The execution of this network analysis yields 411 project pairs - 822 projects in total. Figure 5.12 shows the committer counts across the sample, with the vast majority of projects exhibiting single digit committer counts. This trend is consistent with the observed trends in the forge analysis in the prior chapter as illustrated in figure 4.5.

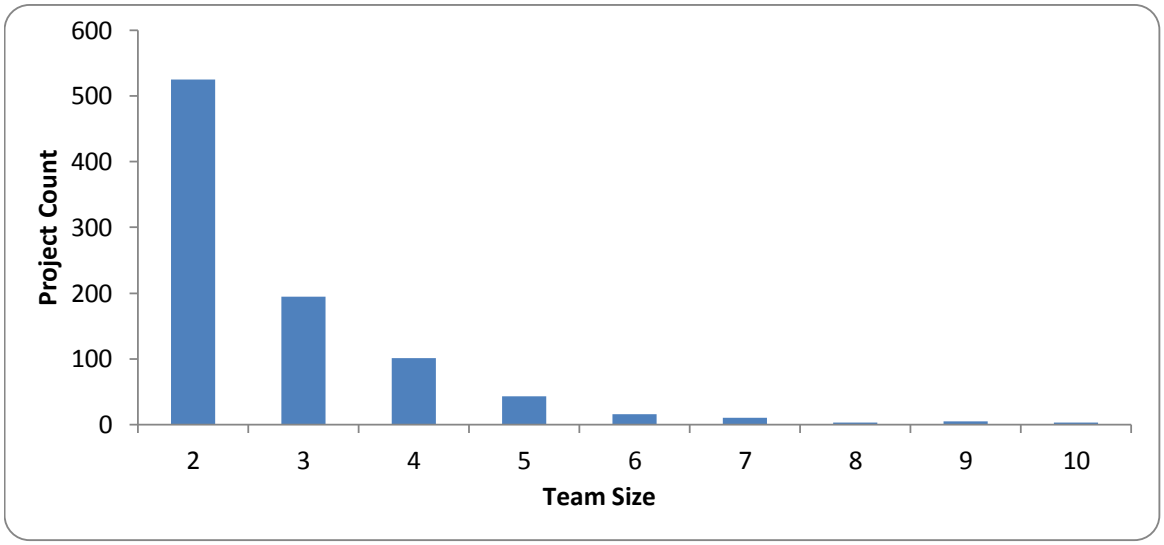


Fig. 5.12 Committer counts across the sample, with the vast majority of projects exhibiting single digit committer counts.

5.5 Intra-project stability analysis

The earlier definition of team stability - the cumulative time that each team member works with their fellow team members - is fairly broad and open to interpretation. This section provides greater precision around this definition and proposes a calculation for intra-project team stability - a measure capturing the degree of stability accrued by a team through the course of a project.

5.5.1 Determining a measure of intra-project stability

This research proposes a measure of stability, assigned at a project-level, to capture the degree to which committers on a project accrue time working alongside their peers on the project. The premise of this measure relates to the earlier definition of stability from Huckman et al. which states that the greater cumulative time that a team of committers spend working with each other, the more stable that team (Huckman et al., 2009).

Before delving into the specifics of how this measure is calculated, it is helpful to first explain that, within this model, there are two main constructs. The first is the 'time-frame of activity'. This applies both at level of the committer and the project and is defined as the window of time stretching from the first and last commits for that entity; specifically only those commits that impact Java source code, excluding for example, documentation changes. For instance, if a project records its first commit on the first day of January and its final commit on the final day of December of that same year, it would have a time-frame of activity spanning the 365 days of that year. Likewise, a committer recording commits on the first and last day of January would have a time-frame of activity spanning the 31 days of that month.

The second construct relates to how stability (or lack thereof) accrues on a project. Figure 5.13 outlines three mechanisms through which a measure of stability can be derived. The first method (labelled '1') is the most straightforward and is to directly capture the number of days in each committers time-frame of activity through the course of the project. This is the most simplest transposition of the Huckman definition onto this problem space. However, it is also a measure that will closely track team size and does not capture the degree to which time-frames of activity overlap. The second method does capture the cumulative number of days that that these time-frames of activity overlap but will also be closely correlated with team size. The third method captures the extent to which the committer time-frames of activity *do not align* with that of the project's time-frame of activity and therefore often misaligning with their peers and failing to accrue stability. For the reasons outlined below, this third method is chosen for this work and the measure is termed the 'Lack of Stability Ratio (LSR).

- **Avoiding tracking team size:** As illustrated in Figure 5.13, the first two methods risk primarily tracking team size rather than stability. LSR, however, calculates a ratio which captures a particular aspect of committer time-frames of activity and therefore avoids tracking team size.

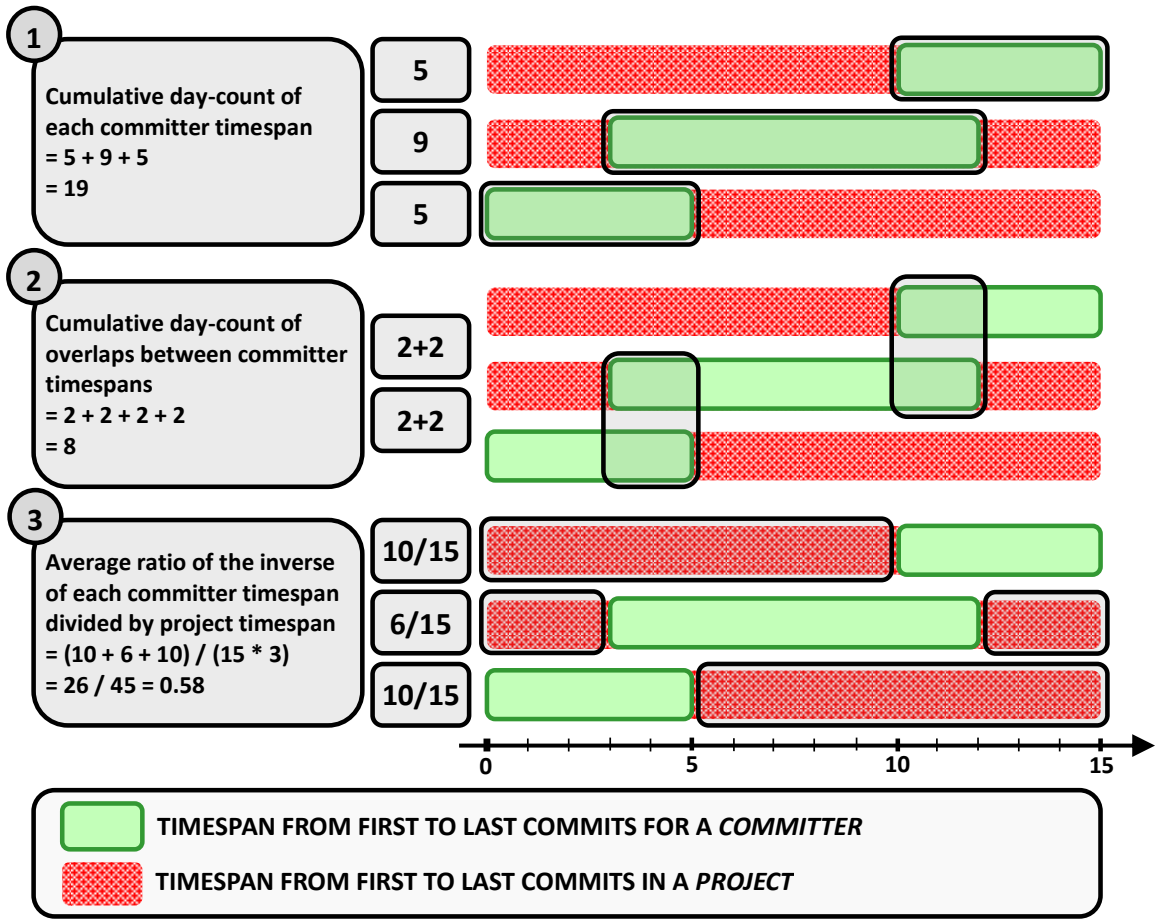


Fig. 5.13 A hypothetical example showing three different approaches to calculating intra-project stability.

- **Adhere to conventions:** By measuring 'lack of stability' rather than 'stability', LSR continues the convention set by the CK metric suite and other software metric suites that favourable measures are lower in value (Lack of Cohesion being a pertinent example).

5.5.2 Calculating LSR in practice

An example of how LSR is calculated on a project from the stability project sample is illustrated in Figure 5.14. The LSR calculation is based on the Jaccard Index where similarity is established between each and every committer time-frame of activity to the overall project time-frame of activity.

The Jaccard index is the simplest of several similarity measures and was developed at the start of the previous century to compare botanical data sets (Jaccard, 1901). The Jaccard Index has some prior use in the field of mining software repositories. Kiefer et al. and Kpodjedo et al. use this measure to calculate similarity between Java classes to observe evolution of software projects through releases (Kiefer et al., 2007; Kpodjedo et al., 2013). Jermakovics et al. used the measure to compute similarities among developers based on common file changes, constructing a network of collaborating developers (Jermakovics et al., 2011). Alternative similarity measures such as the Sørensen-Dice coefficient (Sørensen, 1948) or Euclidean distance add complexity but it isn't clear that either would better capture team stability given that they are more suited to weighting certain factors and clustering respectively. The Jaccard Index is defined as the size of the intersection divided by the size of the union of the sample sets.

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

While the Jaccard Index is designed to establish the similarity between two finite data sets, the LSR calculation is an adaptation of this approach to take into account the fact that there can be more than two committer time-frames that need to be factored into the stability ratio. It is essential that LSR is in the form of a ratio to avoid calculating a number which has a strong direct correlation to committer count which would simply lead to tracking that factor and, ultimately, confounding results. As expressed in the equation below, LSR is the inverse of the simple mean of the Jaccard Indexes of each unique committer time-frame of activity and the overall project time-frame of activity.:

$$LSR = 1 - \frac{\frac{TF_{com.1} \bigcap TF_{all}}{TF_{comm.all}} + \ldots \frac{TF_{com.n} \bigcap TF_{comm.all}}{TF_{comm.all}}}{n}$$

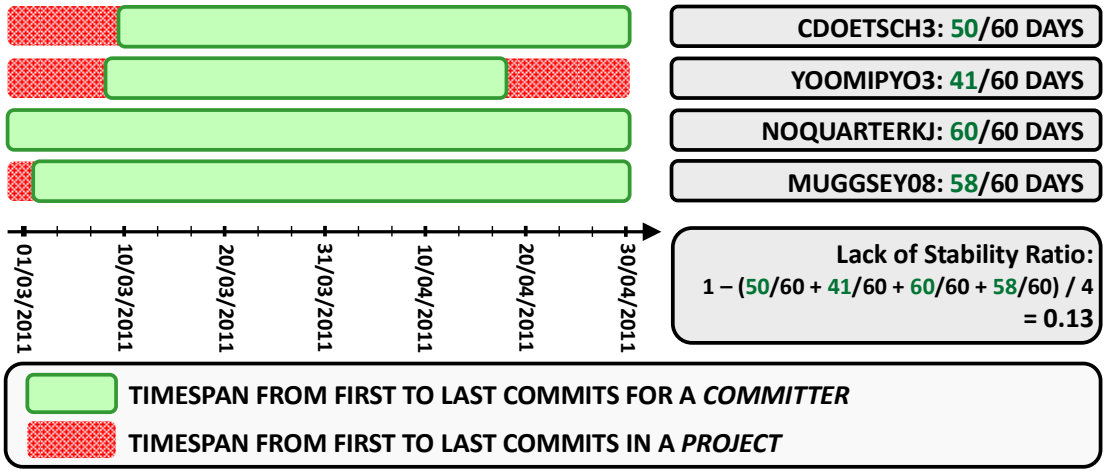


Fig. 5.14 A worked example showing the calculation of the Lack of Stability Ratio (LSR) in a project called 'TeamAwesomeExpress' from the stability project sample.

While the previous example is valid, it is also true that many software projects - particularly larger ones - are commonly divided into 'modules'; a fact that should be factored in the calculations. Each module represents a logical grouping of functionality with its source code typically residing in its own folder within the project repository. The concept of modules helps decouple sections of the codebase, providing the ability for specialisation amongst team members and reducing the need for coordination between them. Where the definition of team stability refers to team members 'working together', this should be expanded to stipulate that they work together on the same module, as otherwise there can be no assumptions on the degree of coordination between members - coordination being crucial to the accruing of stability in a team.

Figure 5.15 shows the prevalence of multi-module projects throughout the stability analysis data set. Figure 5.16 illustrates a worked example of a stability ratio calculation of a multi-module project.

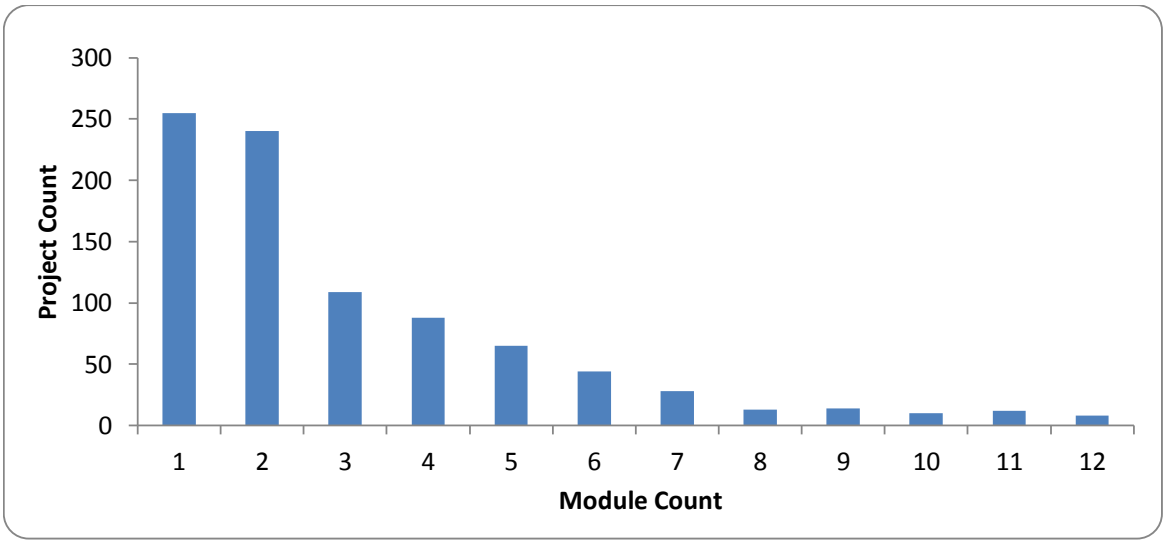


Fig. 5.15 The number of projects grouped by the number of distinct modules within their codebase.

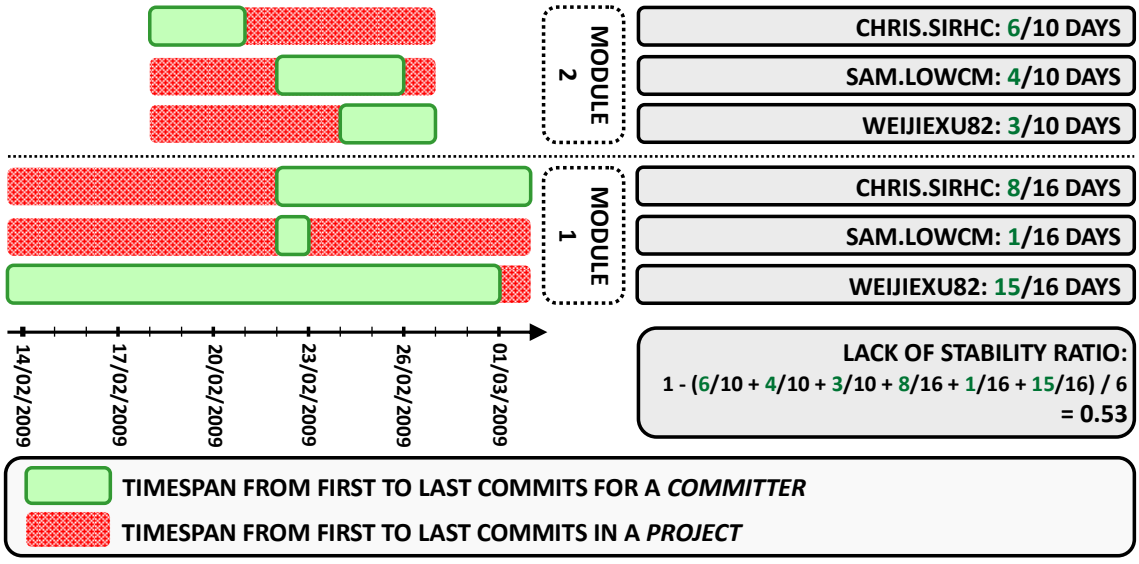


Fig. 5.16 A worked example showing the calculation of the Lack of Stability Ratio in a project called 'PipeDreamAgent' from the stability sample comprising two distinct modules.

Table 5.1 A matrix of Spearman correlation coefficients showing the relationship between various project-level variables

	REVISION COUNT	PROJECT DURATION	MODULE COUNT	LACK OF STABILITY RATIO
REVISION COUNT	1.00			
PROJECT DURATION	0.16	1.00		
MODULE COUNT	0.10	0.32	1.00	
LACK OF STABILITY RATIO	-0.13	0.01	0.04	1.00

KEY	+VE STRONG	+VE MODERATE	+VE WEAK	NONE	-VE WEAK	-VE MODERATE
	0.7 - 1.00	0.40 - 0.69	0.01 - 0.39	0.00	-0.39 – -0.01	-0.40 - -0.69

5.5.3 Validation of the Lack of Stability Ratio

In the previous chapter, one of the key considerations was to mitigate for the fact that increasing team sizes can accompany an increase in functional complexity which could have a confounding impact. LSR is, perhaps, a less intuitive and direct measure than team size and, therefore, it is even more critical to quantify the relationship between this measure and other key factors that may have a confounding impact on CK metrics. Table 5.1 shows that LSR has negligible correlations to revision count, project duration and module count.

By way of qualitative validation, it is notable in Figure 5.18 that the visual representation of the time-frames of activity of two outlier projects from the stability sample show features that would be expected given their respective LSR values. The project named 'Dmdirc' has a high stability ratio and it is clear that the committers to this project work together for an extended period of time with an almost total degree of overlap. Conversely, 'Cykelgarage' shows an early commit by one author followed by a lull in activity and a series of commits by five other authors with some degree of overlap. While it is arguable that this measure inordinately penalises projects with an early commit followed by a period with no committer activity, it is also worth pointing out that this is quite a rare pattern and furthermore it is helpful to capture the gap in time that elapses between periods of activity as those gaps may be associated with a loss of knowledge from the team. In this particular example the commentary in the commit logs reveal that the initial commit activity to Cykelgarage represented an substantial check-in of code representing the first set of functionalities for the software. The subsequent lull is therefore significant in that it creates distance between the initial committer and the subsequent committers to the project - an aspect that is and should be captured by LSR.

The probability distribution for LSR across the stability project sample is illustrated in Figure 5.17 and, while visually similar to a normal distribution, the Kolmogorov-Smirnov test returns a P-value of 0.00 and a D-statistic of 0.50 revealing that half the observations reside outside a normal distribution.

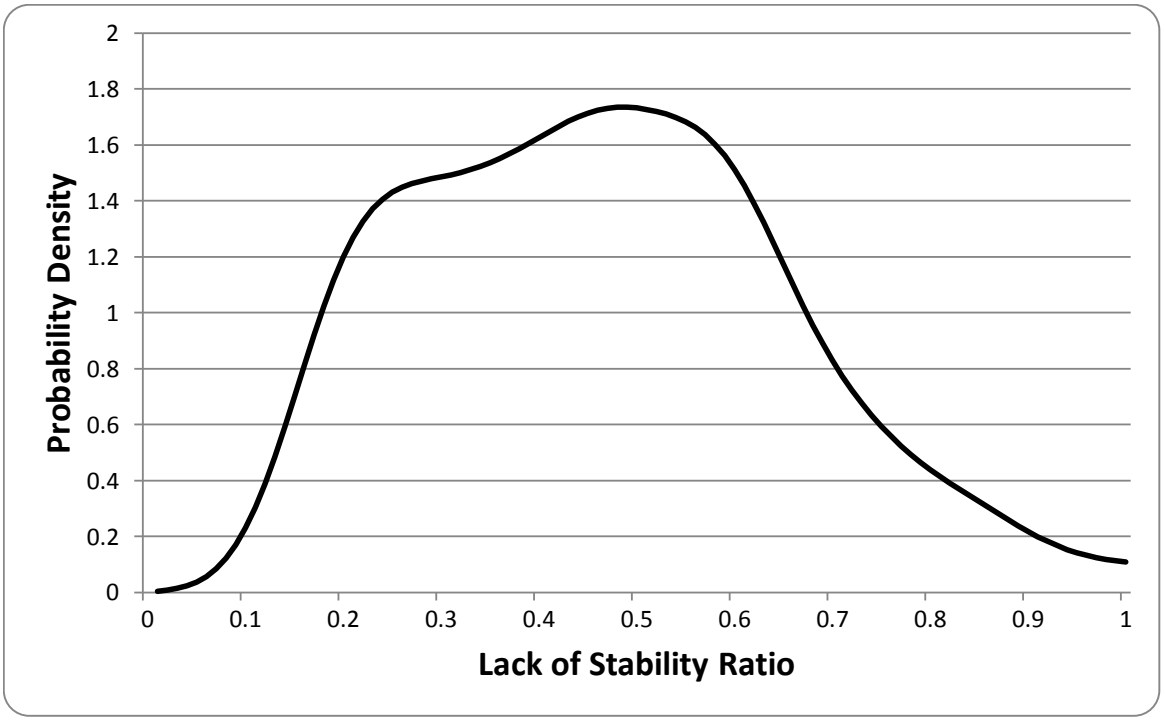


Fig. 5.17 The probability distribution for the Lack of Stability Ratio (LSR) across the stability sample.

5.5.4 Results

The initial set of results visualisations are represented as scatter diagrams showing the mean CK metric values at a project level plotted against LSR for those projects. Averaging CK metrics may be misleading as it can be heavily skewed by outlier values and thus misrepresentative of the data set. That said, it can be helpful as a first step in determining if there any clear trends manifest which can then be explored further. There are two observations that can immediately be made. First, the majority of the data points exist on the left-hand-side of the plots. This is a natural given that the majority of projects exhibit LSR values lower than 0.5 as confirmed by the probability distribution depicted in 5.17. The second observation is

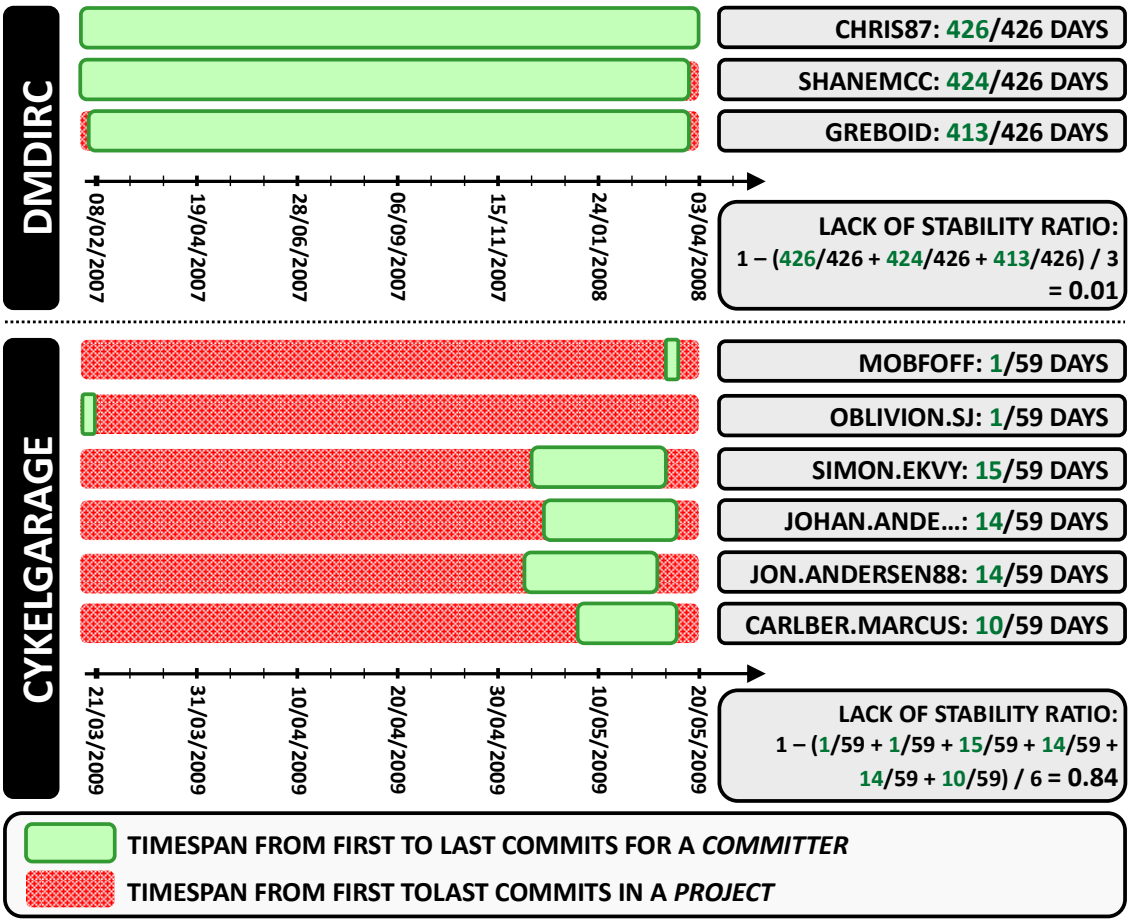


Fig. 5.18 Visualising the 'time-frames of activity' for two outlier projects: Cykelgarage and Dmdirc.

that there is no clear and powerful relationship between LSR and the project averaged metric values that manifest in these plots.

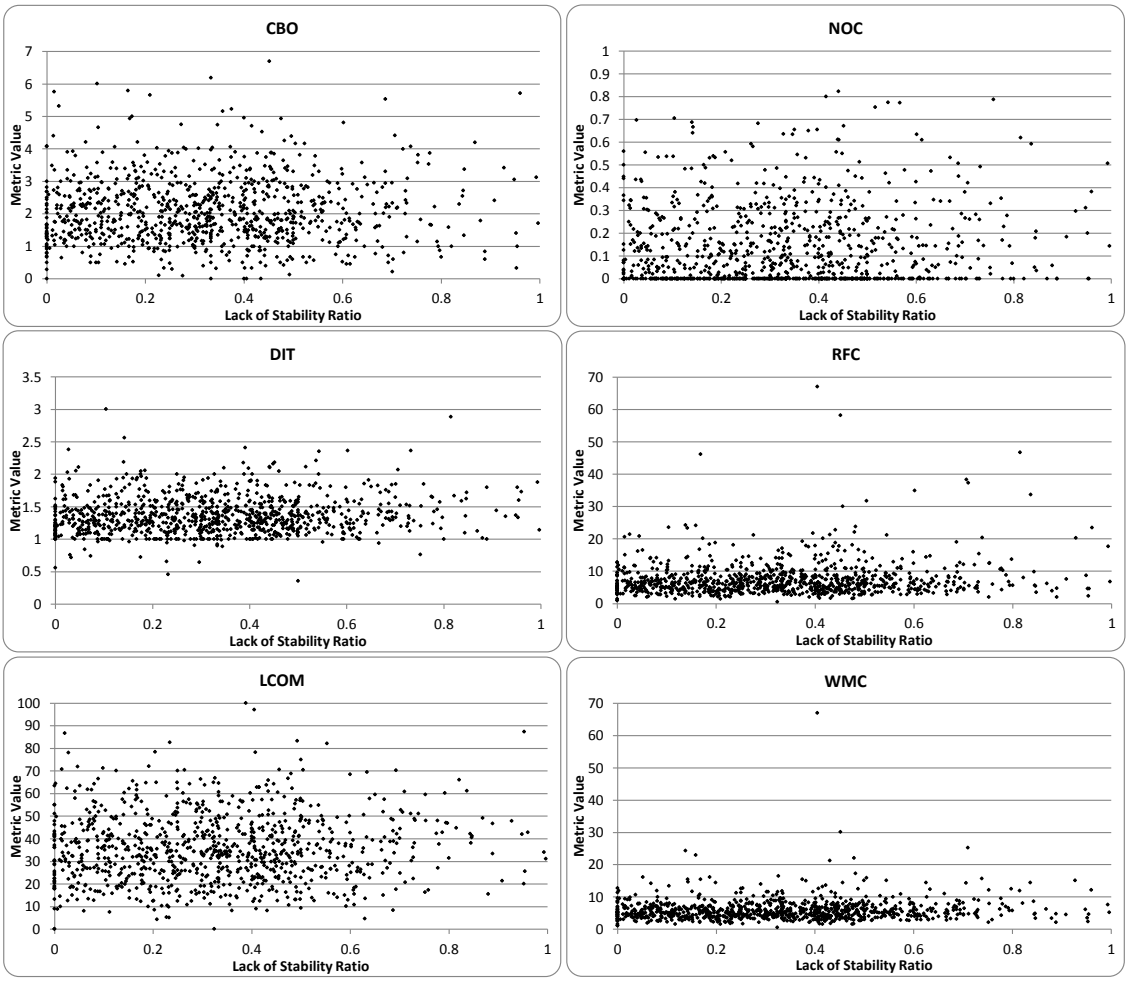


Fig. 5.19 Scatter diagrams plotting metric values against stability ratios.

The charts in Figure 5.20 show mean metric values for projects exhibiting a range of stability ratios. There is a noticeable upward trend across all metric types with the exception of CBO and NOC, implying at this early stage of analysis that teams with greater stability produce projects with higher cohesion and lower inheritance complexity.

Table 5.2 shows the results of a basic Ordinary Least Squares linear regression with LSR as the sole independent variable. In a departure from the analysis in the prior chapter, it is not necessary to include revision count into this model as the correlation analysis in Table 5.1 shows that LSR does not track any of the key project-level factors and therefore those factors

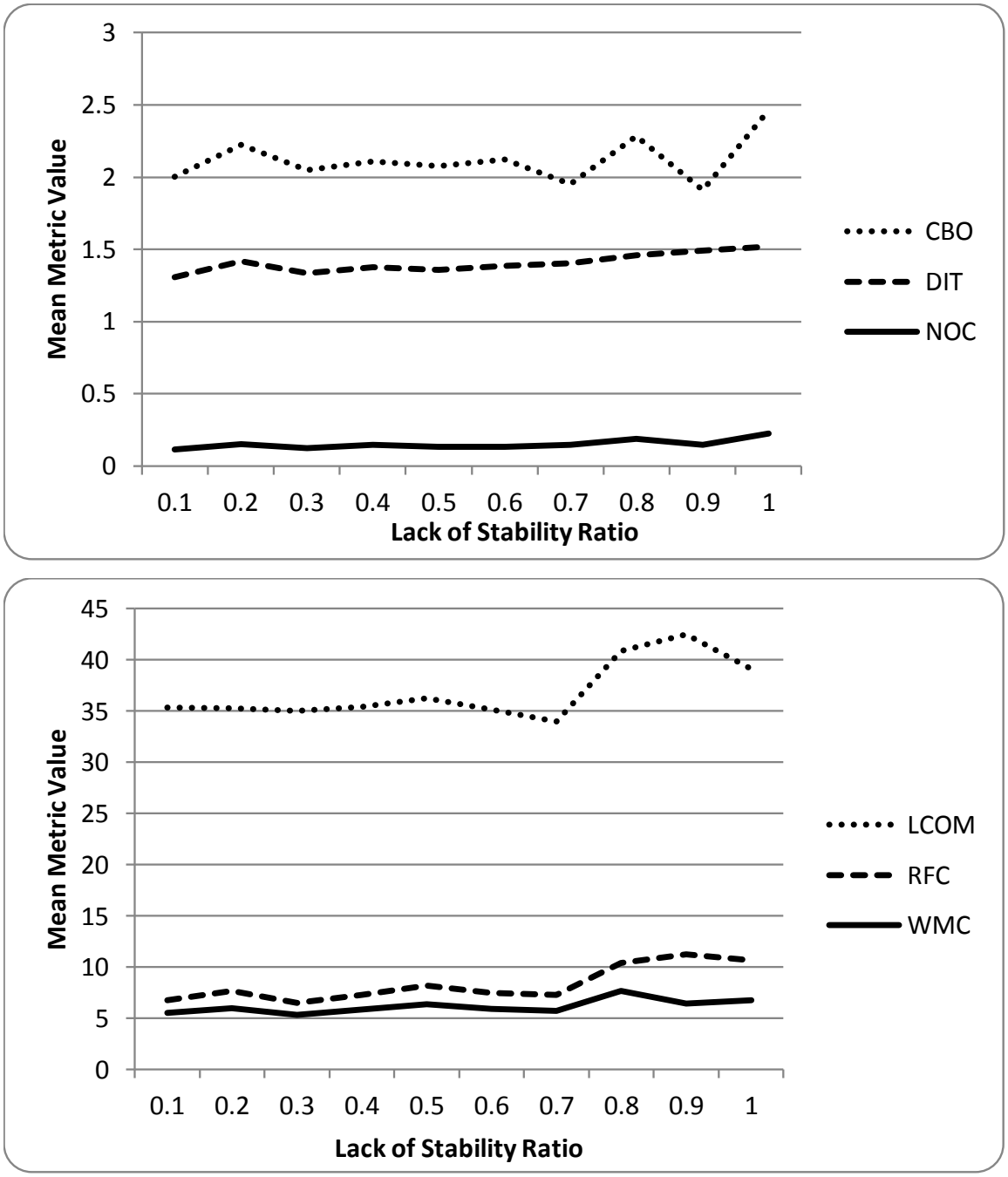


Fig. 5.20 Mean project-level metrics grouped by stability ratio (rounded up to one decimal place).

should not have a confounding impact. The model can be expressed as follows where LSR is multiplied by a coefficient (β), γ is the intercept, and ε is the standard error:

$$Metric\,Value = \beta_{LSR}\,LSR + \gamma + \varepsilon$$

A number of observations can be made based on the results in Table 5.2.

- **Rejection of null hypothesis H0,2:** The R-Squared values show that LSR can explain a high proportion of the variance observed in the DIT and LCOM metrics, a substantial proportion of the variance of CBO, and a minimal element of the variance of RFC and WMC. All regressions report a p-value < 0.005 - i.e. the Bonferroni corrected α - and can be considered highly significant. CBO captures coupling and modularity, DIT captures inheritance complexity and LCOM captures coupling. Consequently, this model provides support for the alternative hypothesis H1,2.1 that *'less stable development teams produce software exhibiting higher coupling, higher complexity, lower cohesion and lower modularity'*.
- **Stability has a powerful impact on CBO, DIT and LCOM:** The high coefficient estimates show that LSR has a fairly powerful impact on the CBO, DIT and LCOM metrics. For instance, the 8.67 coefficient reported in the CBO column indicates that a movement of 0.1 in LSR would have a 0.87 impact on the CBO: a metric with a mean of 3.55 within the stability sample. The low standard errors and high T-statistics show that this linear model provides a reasonable fit.

While this result evidences the impact of LSR on CK metrics, the 'idiosyncratic' project features referred to in Chapter 4 (previously established to play a significant role in the relationship between team size and CK metrics) is likely to play a role in determining how LSR impacts the structural attributes of individual projects. To take a hypothetical but intuitive example, if a software development team is geographically collocated and working on a highly complex problem, stability should have greater predictive power than for a development team that is geographically dispersed and working on a relatively simple problem solving to established and documented design patterns. While individually establishing the impact of these myriad of factors is well beyond the scope of this work, using the 'Linear Mixed Models' (LMM) statistical approach can, as demonstrated in Chapter 4, provide a linear regression which factors in this idiosyncratic component to establish coefficients with greater

Table 5.2 Results of the OLS linear regression with intra-project stability as a single independent variable.

	CBO	DIT	LCOM	NOC	RFC	WMC
R-squared	**0.23**	**0.61**	**0.44**	**0.01**	**0.12**	**0.06**
LSR Coefficient (β_{LSR})	**8.67**	**4.03**	**105.70**	**0.84**	**37.31**	**22.75**
RMSE (ε)	**5.97**	**1.17**	**43.17**	**2.86**	**37.53**	**55.64**
Intercept (γ)	**3.29**	**1.65**	**45.78**	**0.31**	**10.06**	**6.99**
LSR Standard Error	**0.07**	**0.02**	**0.54**	**0.03**	**0.47**	**0.40**
LSR T-Statistic	**119.53**	**277.37**	**196.42**	**24.47**	**79.86**	**56.95**

P-values across all displayed regressions = 0.00
Degrees of Freedom = 49,616

accuracy, reducing residual errors in the process. This can be expressed in a similar way to the equation in the previous section where γ_p is now the project-specific intercept:

$$MetricValue = \beta_{LSR} LSR + \gamma_p + \varepsilon$$

Table 5.3 shows the results of the LMM regression and the following observations can be made.

- **Sample variance is substantially higher than group variance:** As observed with the LMM models of the previous chapter, the groups exhibit lower variance than the overall sample confirming the significant influence that the project-specific idiosyncratic characteristics have on CK metric values.
- **Lower Residuals:** Again, consistent with the application of LMMs in the previous chapter, Table 5.3 shows substantially lower residuals across all metric regressions when compared to the OLS results in Table 5.2.
- **Lower coefficient estimates:** The lower coefficient estimates in the LMMs indicate that team stability has less impact on metric values than otherwise apparent in the OLS regression. This is only noteworthy in the context of highlighting that LMMs revise the coefficient estimates to reduce residuals given the flexibility of project-specific intercepts.

Table 5.3 Results of the 'random intercepts' linear regression with intra-project stability as a single categorical independent variable with observations grouped by project.

		CBO	DIT	LCOM	NOC	RFC	WMC
Statistics	Sample variance	31.33	0.69	1348	7.57	1391.63	1020.44
	Group variance	2.92	0.74	784.84	0.04	46.28	14.20
LSR Coefficient (β_{LSR})		5.90	3.33	100.49	0.65	23.60	17.85
RMSE (ε)		3.20	1.01	31.03	1.54	21.41	17.72

P-values = 0.00
Degrees of Freedom = 49,616

5.6 Inter-project stability analysis

While studying intra-project stability enables the quantitative analysis of the impact of stability on the structural metrics of software, it is also informative to study at another key dimension of team stability: namely the impact of *inter-project stability* on structural metrics; that is the stability that is gained through a development team retaining a consistent set of committers across multiple projects. This variant of stability intuitively can bear relevance to those practitioners looking to understand how continuing with an unchanged development team can bring benefits to internal and external attributes of the produced software. For this analysis the same null and alternate hypothesis hold as in the previous section: H0,2 and H1,2.1 respectively.

5.6.1 Analysis approach

As documented earlier in Section 5.4 of this chapter, there are a number of criteria that are applied in order to obtain a dataset that lends itself well to contrasting by inter-project team stability: namely that the project timelines should not overlap and that the chronologically later project should not contain any committers that did not exist in the earlier project. The nature of the subsequent analysis is twofold. First, the metrics of projects within 'stable project pairs' are compared to each other as depicted in Figure 5.21. The second analysis is a linear regression to model the relationship between inter-project team stability and CK metrics.

To perform a linear regression, it is necessary to derive a categorical measure of lack of stability which can serve as an independent variable acting on CK metrics as dependent variables within the linear model. To achieve this the model assigns a binary with a value of '1' assigned to the first (chronologically) of a stable project pair and '0' assigned to the second project in the pair. This is an approach that attributes a total lack of team stability to the earlier project and total stability to the later project of the stable project pair. This is a simplification as it is entirely conceivable that any project team may have collaborated previously on other projects outside the GoogleCode forge and thus not captured within our data set and not reflected in this measure of lack of stability. Furthermore, in this analysis no account is made of the intra-project stability that accrues through the course of the earlier or the later project within a pair. These simplifications do not constitute a significant threat to the validity of this analysis as there is certainly an accumulation of inter-project stability from the earlier project to the later project within the stable project pair, enabling meaningful observations to be made. However, this measure leaves some scope for refinement as will be discussed in the Future Work in the next chapter of this thesis.

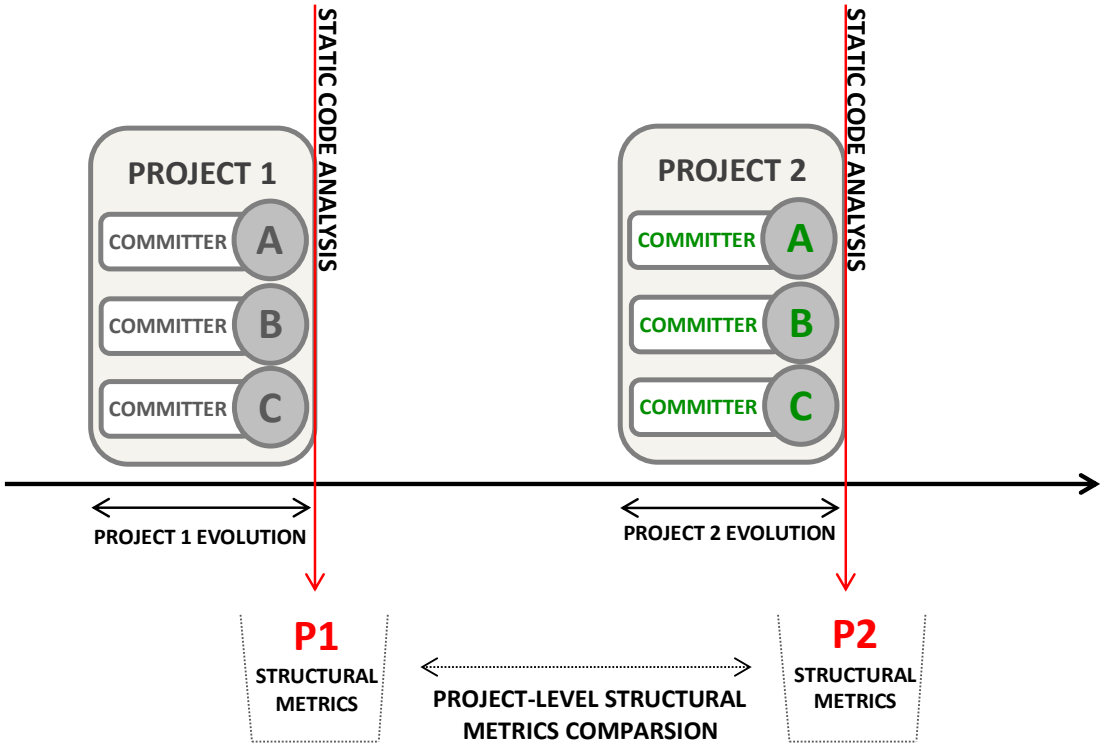


Fig. 5.21 An illustration of the inter-project stability analysis approach.

5.6.2 Results

Figure 5.22 shows the result of the first phase of analysis detailing the proportion of 'stable project pairs' where the structural metrics of the each member of the pair exhibit a significant difference from the other with a 95% confidence interval; i.e. a Mann-Whitney test returning p-values of < 0.05. Significant difference across a large proportion of projects are observed - from a maximum of 38% for RFC to a minimum of 22% for NOC. Consistent with the inter-project stability analysis, this is further rejection of null hypothesis H0,2 that *'Development team stability does not impact the coupling, complexity, cohesion or modularity of the produced software'*. The nature of this difference is studied, as in the prior analyses in this thesis, through linear modelling.

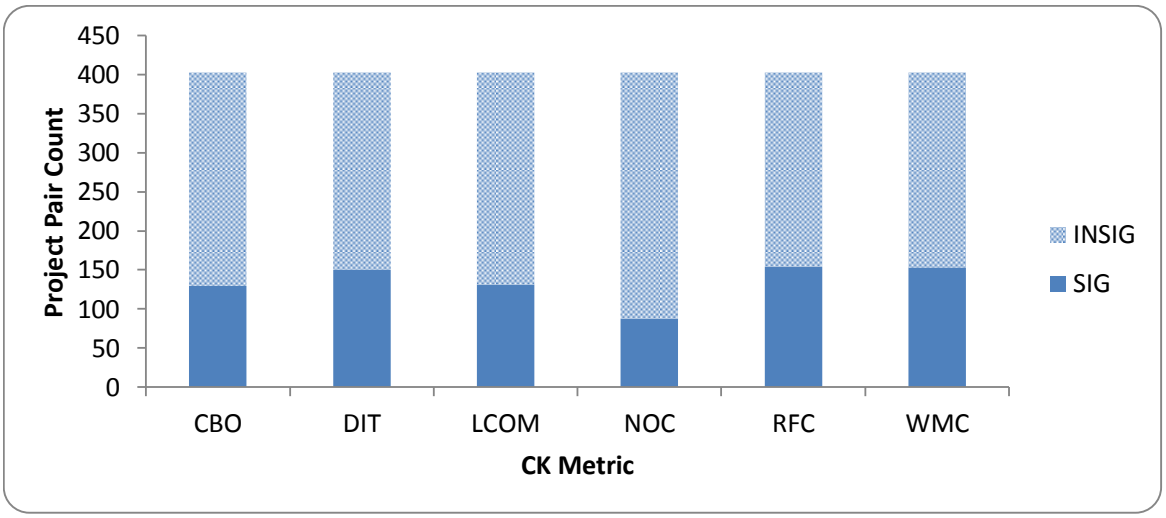


Fig. 5.22 A chart showing the number of project pairs that show a significant difference (i.e. p-values < 0.05) between the metrics of each project in the pair.

Table 5.4 details the results of an OLS linear regression with the categorical binary lack of stability measure as a single independent variable. Table 5.5 shows the results of the LMM Regression. The following observations can be made from these results.

- **Further rejection of null hypothesis H0,2:** The R-squared values show that a substantial proportion of the DIT and LCOM metrics variance is explained by this inter-project lack of stability. The positive coefficients along with the low standard errors relative to those coefficients shows that CK metrics trend positively with inter-project lack of stability; a result consistent with the observations in the earlier intra-project stability analysis. This result provides further support for alternative hypothesis H1,2.1 that

Table 5.4 Results of the 'ordinary least squares' linear regression with inter-project stability as a single categorical independent variable.

	CBO	DIT	LCOM	NOC	RFC	WMC
R-squared	**0.12**	**0.35**	**0.29**	**0.01**	**0.04**	**0.02**
Binary Lack of Stability Coefficient (β_{ILSR})	**3.33**	**1.62**	**46.19**	**0.31**	**12.04**	**8.06**
RMSE (ε)	**6.17**	**1.23**	**39.70**	**1.89**	**39.59**	**50.57**
Intercept (γ)	**3.63**	**1.70**	**44.22**	**0.33**	**16.33**	**10.33**
Binary Lack of Stability Standard Error	**0.03**	**0.01**	**0.26**	**0.01**	**0.22**	**0.19**
Binary Lack of Stability T-Statistic	**102.91**	**205.39**	**178.79**	**23.41**	**54.32**	**42.26**

P-values across all displayed regressions = 0.00
Degree of freedom = 79,057

Table 5.5 Results of the 'random intercepts' linear regression with inter-project stability as a single categorical independent variable with observations grouped by project.

		CBO	DIT	LCOM	NOC	RFC	WMC
Statistics	**Sample variance**	**31.33**	**0.69**	**1348**	**7.57**	**1391.63**	**1020.44**
	Group variance	**5.80**	**2.24**	**2109.93**	**0.07**	**76.36**	**28.75**
Binary Lack of Stability Coefficient (β_{ILSR})		**0.77**	**0.04**	**2.51**	**0.21**	**5.69**	**5.44**
RMSE (ε)		**5.56**	**0.82**	**36.93**	**2.50**	**40.48**	**35.53**

P-values for all shown regressions = 0.00
Degrees of Freedom = 79,057

less stable development teams produce software exhibiting higher coupling, higher complexity, lower cohesion and lower modularity.

- **LMM results consistent with previous results:** When compared to the basic OLS regression, as with the intra-project analysis, the coefficient estimates are lower (with broadly lower residuals) given the project-specific intercepts of the LMM regression.

Table 5.6 Loading coefficients of the Principal Component Analysis as applied to the team stability analysis sample.

	LOC	Revision Count	Lack of Stability Ratio	Committer Count	CBO	DIT	LCOM	NOC	RFC	WMC
PC-1	0.13	0.15	-0.01	0.08	0.40	0.35	0.25	0.36	0.54	0.44
PC-2	-0.16	-0.38	0.19	-0.40	-0.14	-0.29	0.43	-0.37	0.20	0.41

5.7 Results at a Project Level

5.7.1 Project Selection

Applying PCA to the team stability sample, a new set of loading coefficients emerge as documented in Table 5.6. It is notable that LSR hardly features in the first principal component. This can be explained by the fact that LSR explains a great deal of variance in the CK metrics - variance which is captured through the substantial loading coefficients on the CK metrics themselves. Figure 5.23 shows the sample scattered by the two principal components.

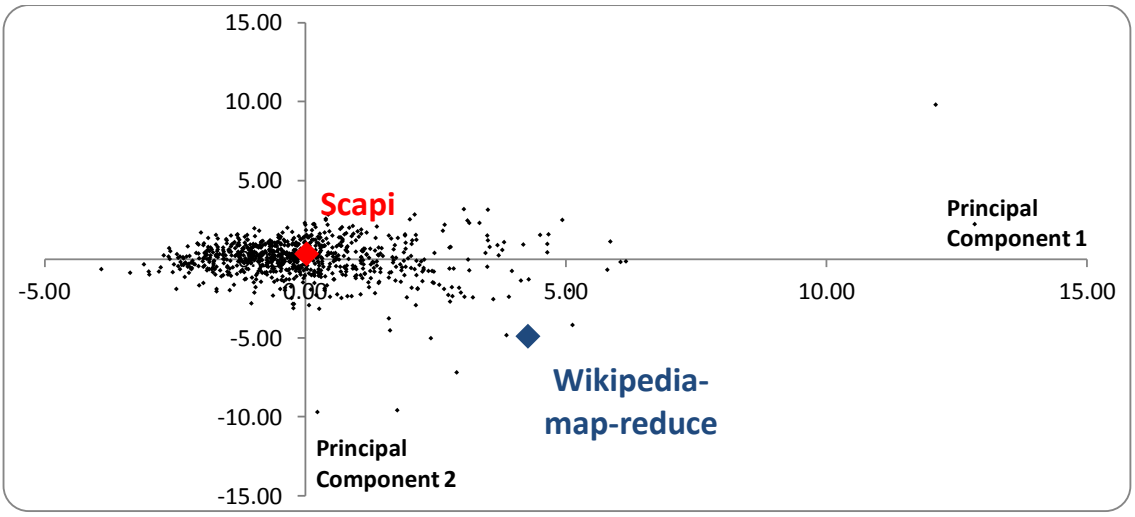


Fig. 5.23 A visualisation of the team size sample scattered across the two principal components. The selection of Aviator and Precise for further study.

Through the intra-team stability analysis, the broad trends observed were that DIT, LCOM, RFC, WMC trended positively with LSR. Two projects were selected with varying LSR values exhibiting relative metrics trends similar to those broader trends. Wikipedia-map-

reduce and Scapi have LSR values of 0.41 and 0.15 respectively indicating that the latter has a project team which is substantially more stable. These particular projects were chosen as have similar revision counts (129 and 157 respectively) and five unique committers each. The two projects appear in distant positions from one another in Figure 5.23 with Scapi residing close to the centre of the cluster while Wikipedia-map-reduce is very much an outlier owing to its high RFC and WMC values.

Wikipedia-map-reduce is an API that allows analysis of Wikipedia using the Hadoop Map-Reduce framework for parallel processing. Is is purely server-side software with no graphical interface. Scapi is a loan administration system which forms the basis of a company called Monteplus which operates as an online pawn shop, swapping various personal items or assets for cash.

Although both projects share the same team size, a deeper analysis of committer behaviour analysis depicted in Figure 5.24 shows that Wikipedia-map-reduce has a single prolific contributor who is responsible for the majority of the codebase while Scapi's five committers more equitably distributed responsibility of the codebase amongst themselves. In the next chapter more consideration will be given to the role of 'core' committers versus that of 'peripheral' committers and how this can present both a threat to validity and an opportunity for further work.

5.7.2 Project Comparison

Through qualitatively reviewing the code in both Scapi and Wikipedia-map-reduce, the latter emerges as the more structurally complex of the two projects. Although there is certainly less to critique in Wikipedia-map-reduce than in the previously reviewed Precise project - an observation which is consistent with the lower absolute LCOM and RFC numbers in Wikipedia-map-reduce relative to Precise - there are still some examples of poor encapsulation and inordinate structural complexity which will lead to poor cohesion, high coupling and low modularity. The Encoder class is an example of this as the code snippet in Figure 5.26 highlights. It is an excessively large class at 1616 lines, 34 methods and four distinct inner classes. This class is fairly anomalous within the broader codebase which is generally well written.

In Wikipeda-map-reduce there is substantially more use of inheritance which confirms the higher DIT compared to Scapi (1.36 and 1.16 respectively). This is largely driven by the fact

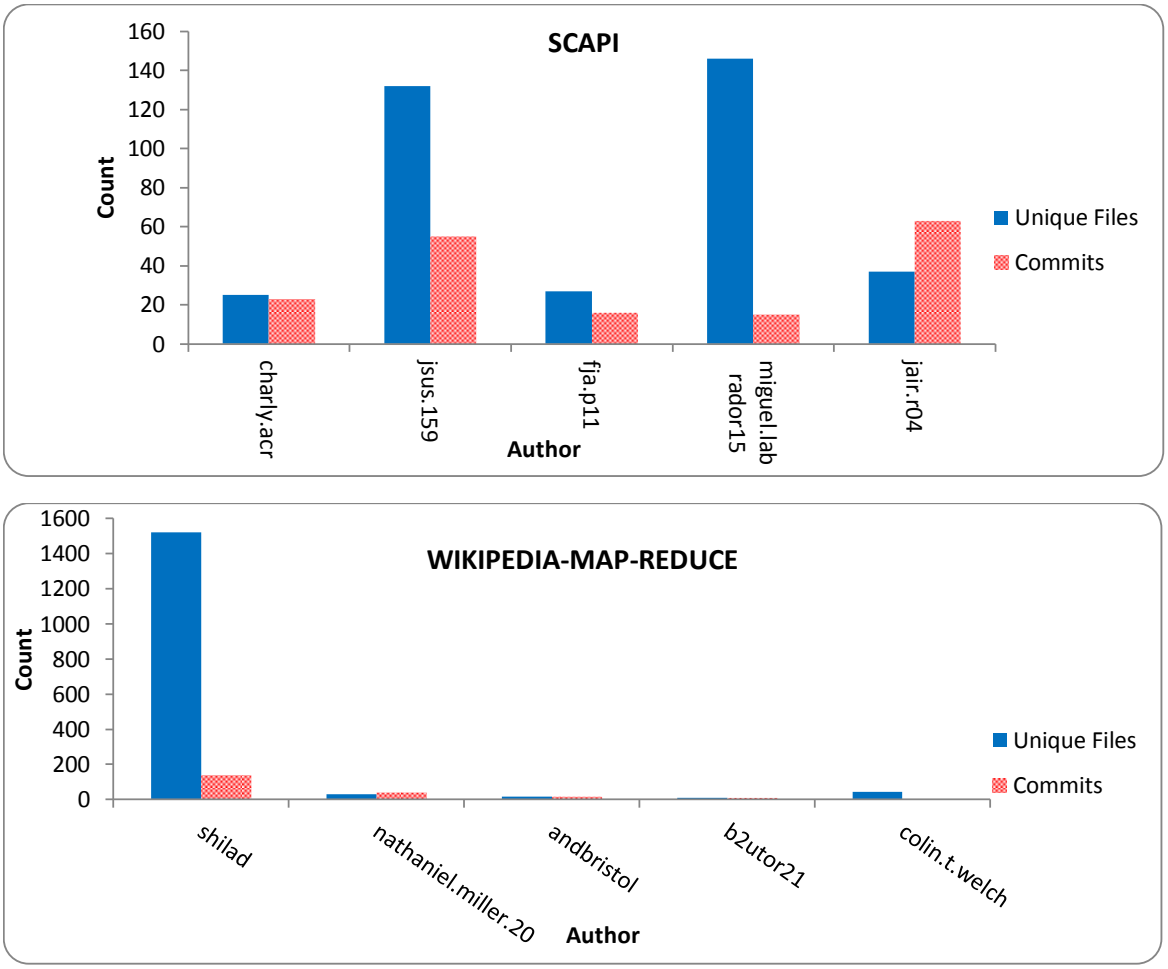


Fig. 5.24 Committer behaviour analysis for the Scapi and the Wikipedia-Map-Reduce projects.

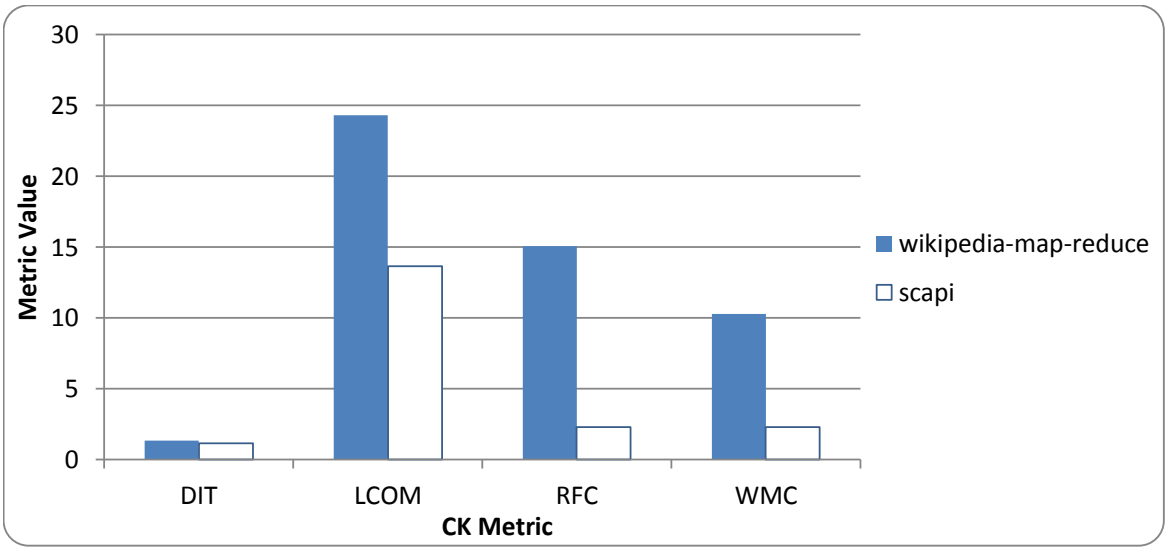


Fig. 5.25 Key structural attributes for Wikipedia-map-reduce compared against Scapi.

that Wikipedia-map-reduce has a substantial library of collections implementations which functionally lend themselves well to sharing behaviour via inheritance.

Table 5.7 documents the intercept and residual values for the Scapi and Wikipedia-map-reduce projects from the intra-project LMM regression. As was observed the similar analysis of team size in Chapter 4, the project that is more distant from the centre of the cluster exhibits higher residuals - in this case Wikipedia-map-reduce. This is attributable to the fact that the coefficient estimates of the regression line will be dominated by those non-outlier projects that make up the main cluster in the scatter plot and consequently show lower residuals. The intercept values are more difficult to interpret and are influenced by two key factors. The first is that Wikipedia-map-reduce has higher metric values which will influence a higher intercept value. However, given that the LSR is higher for this project, the regression line will have further to travel along the x-axis in order to intercept with the y-axis, which causes a decrease in the intercept values. As a result, the intercepts are a mixed bag with Wikipedia-map-reduce showing higher intercept values for CBO, DIT, NOC and RFC and lower intercept values for LCOM and WMC.

While the team stability analysis has accurately identified project teams with a higher lack of stability produces software with degraded structural attributes (from a maintainability standpoint), there are other influencing factors given that stability cannot explain all the variance in the structural metrics of the software. Of these project-specific 'idiosyncratic

Table 5.7 The intercepts and residuals for the Scapi and Wikipedia-ma-reduce projects.

		CBO	DIT	LCOM	NOC	RFC	WMC
Intercepts (γ_p)	**Scapi**	-0.41	-0.06	12.24	-0.02	-0.07	0.41
	Wikipedia-map-reduce	1.76	0.65	-2.77	0.16	1.40	-0.01
Residuals (ε_p)	**Scapi**	2.08	1.89	72.64	0.00	9.91	9.91
	Wikipedia-map-reduce	13.09	2.19	68.28	7.59	44.73	27.95

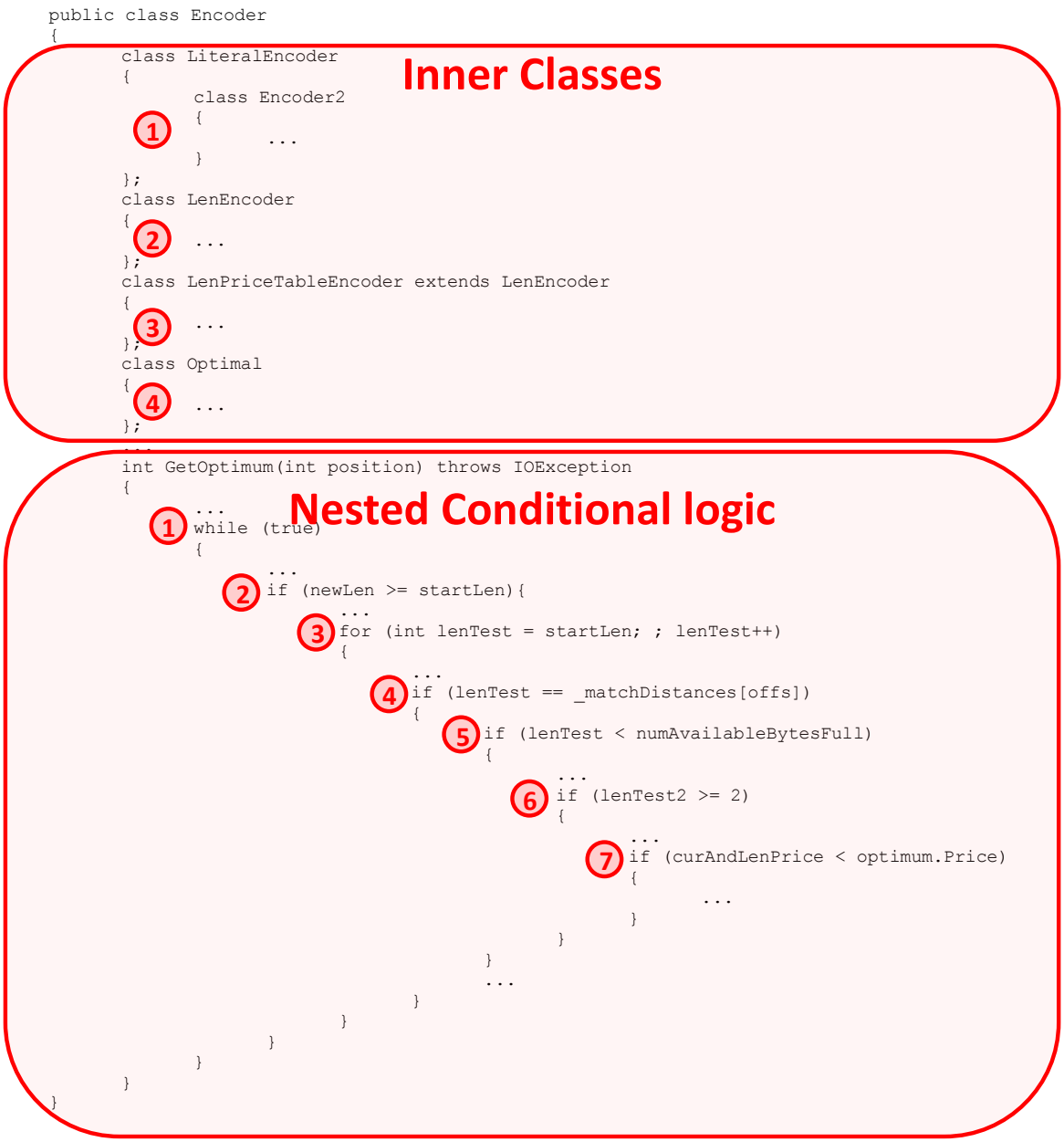
P-values across all regressions: 0.00

factors', such as relative experience levels or problem domain knowledge, further analysis would be beneficial but is beyond the scope of this work.

5.8 Summary

A number of observations can be made as a result of the analyses in this chapter. The intra-project stability analysis yielded clear trends with the introduction of the LSR measure being central to the analysis. Those teams exhibiting greater stability exhibited structural metrics that tended to be lower across measures of structural complexity and cohesion. The trends on inter-project stability were similar. The approach of mining for 'stable project pairs' proved to be a useful approach in contrasting metric populations, showing significant difference between the structural metrics of the project pairs across a large proportion of the stability analysis data set. The linear regressions proved that a substantial portion of the variance with the sample was attributable to the categorical measure of lack of stability. However, the R-squared values were lower in comparison to the intra-project stability linear model, possibly attributable to the lower precision associated with the inter-project binary categorical lack of stability measure.

Given the negative correlation between these structural metrics and fault-proneness, it can be confirmed that the observations within this chapter are consistent with the work of Huckman et al. (Huckman et al., 2009) and Gardner et al. (Gardner et al., 2012) who found that greater team stability was associated with lower fault-proneness. Referring, again, to table 2.4 (the survey of research correlating CK metrics to the sub-attributes of maintainability), inferences can be drawn from the observed structural trends against the impact on the




```
public class Encoder
{
        class LiteralEncoder
        {
                class Encoder2
                {
                        ...
                }
        };
        class LenEncoder
        {
                ...
        };
        class LenPriceTableEncoder extends LenEncoder
        {
                ...
        };
        class Optimal
        {
                ...
        };
        ...
        int GetOptimum(int position) throws IOException
        {
                ...
                while (true)
                {
                        ...
                        if (newLen >= startLen){
                                ...
                                for (int lenTest = startLen; ; lenTest++)
                                {
                                        ...
                                        if (lenTest == _matchDistances[offs])
                                        {
                                                if (lenTest < numAvailableBytesFull)
                                                {
                                                        ...
                                                        if (lenTest2 >= 2)
                                                        {
                                                                ...
                                                                if (curAndLenPrice < optimum.Price)
                                                                {
                                                                        ...
                                                                }
                                                        }
                                                }
                                                ...
                                        }
                                }
                        }
                }
        }
}
```

Fig. 5.26 A code snippet from Encoder class within the Wikipedia-map-reduce project. Multiple inner classes and nested iterative blocks are numerically labelled.

maintainability of the software. With the results in this chapter showing that lack of cohesion, coupling and inheritance complexity tend to increase against a decreasing team stability, and given the already established negative correlations between the impacted CK metrics and maintainability in the prior literature, it can be deduced that lower team stability leads to lower software maintainability. This is depicted in Figure 5.27 and allows for a rejection of null hypothesis H0,2 and a confirmation of the alternative hypotheses H1,2.1 and H1,2.2.

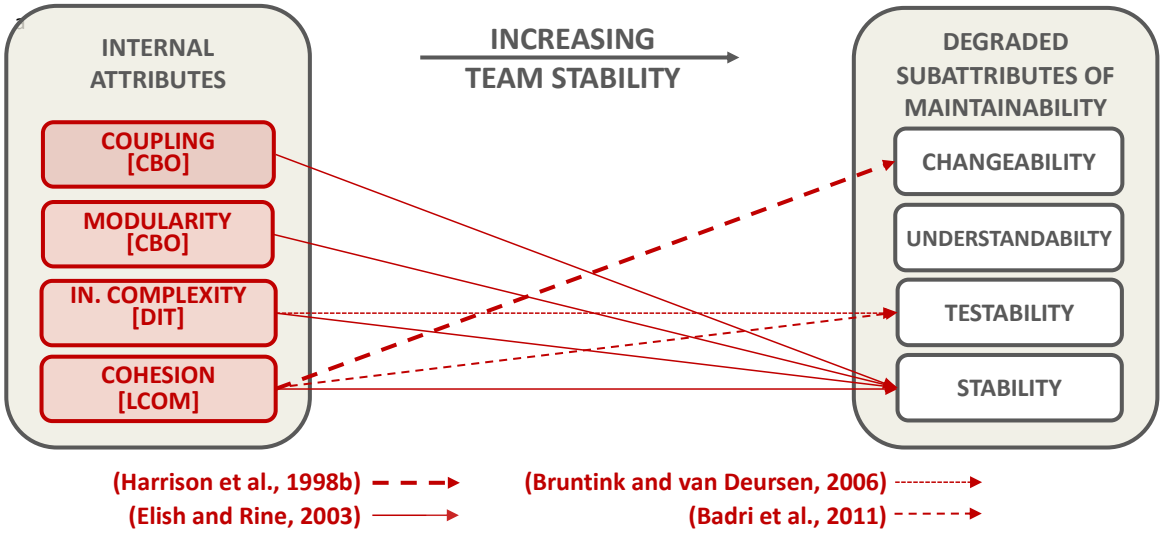


Fig. 5.27 Summary of the results of the Intra-project stability analysis. This analysis shows similar trends to the team size analysis with Changability, Testability and Stability all associated with a degradation inline with the observed trends within the structural metrics.

5.9 Chapter Review

This chapter studied the relationship between team stability and the CK metrics of the produced software. As in the previous chapter, at the outset the sample extraction and exploratory data analysis was presented. The two measures of Inter-project and intra-project team stability were defined and modelled individually. The linear models were developed and analysed in conjunction with specific projects from within the sample.

The next chapter is a discussion primarily covering the contribution to knowledge within this thesis, threats to validity, and potential avenues of future work.

Chapter 6

Discussion

6.1 Introduction

This chapter is organised into five sections as illustrated in Figure 6.1. Section 6.2 reviews the thesis objectives, hypotheses and research questions relative to the contribution of this research. Section 6.3 reviews the contributions to knowledge contained within this research. Section 6.4 details the limitations of this work in the context of the threats that they pose to the validity of this research. Section 6.5 is a brief summary of the distilled conclusions of this thesis. Section 6.6 considers some of the personal reflections of the author upon this work and the state of the art. Finally, Section 6.7 documents future avenues of research within the field of mining software repositories.

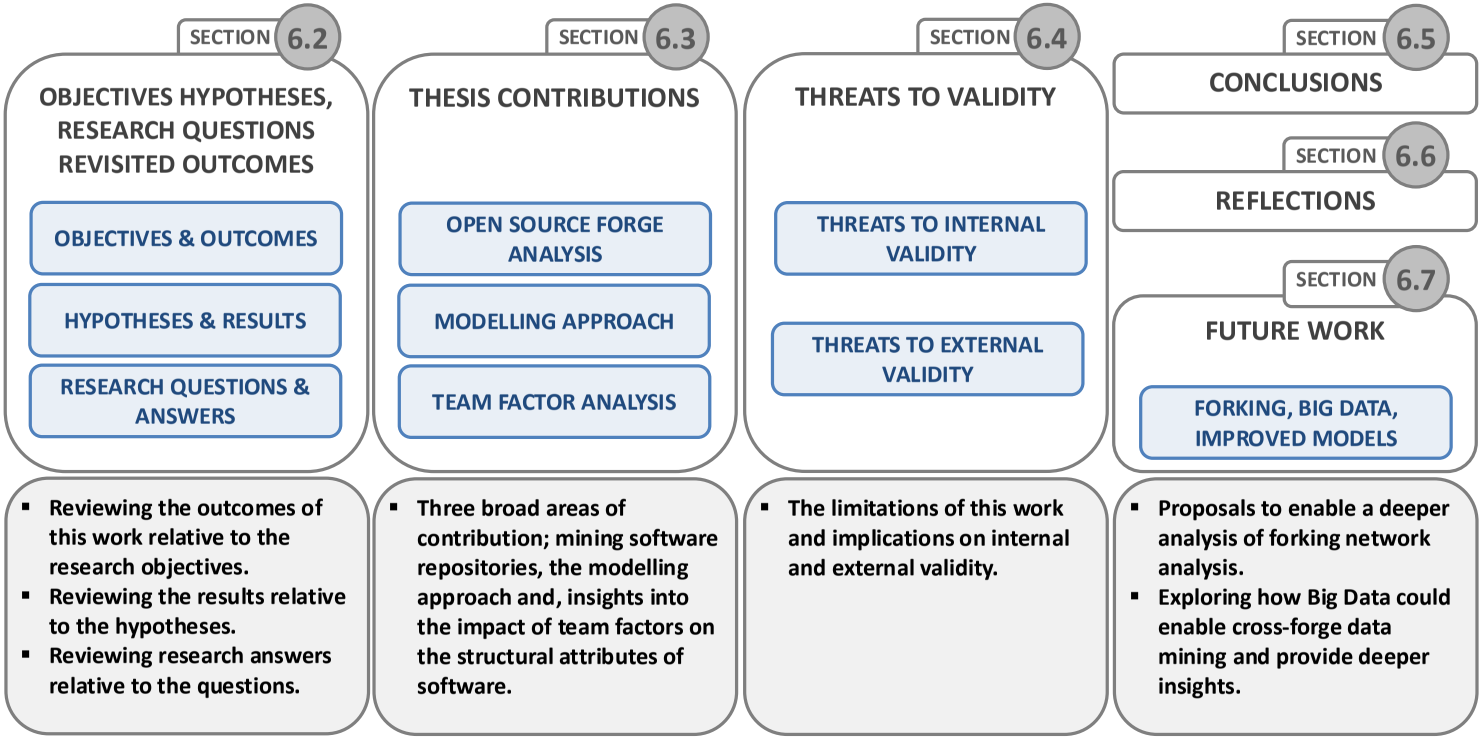


Fig. 6.1 Chapter 6 outline providing an overview of the contents of each section.

6.2 Objectives, Hypothesis, Research Questions Revisited

In this section the objectives, hypotheses and research questions are reviewed, assessed and discussed relative to the outcomes in the thesis.

Table 6.1 shows a restatement of the objectives of this research assessed against the outcomes discussed in this thesis. Broadly the research objectives have been met and the results clearly show a relationship between the team factors and the internal and external attributes of the software.

Table 6.2 summarises how the hypotheses formulated at the outset of this research mapped onto the results. The alternative hypotheses around the development team size analysis (H1,1.1, H1,1.2) stated that larger development teams would produce software with essentially degraded structural attributes resulting in degraded maintainability. The rationale of the hypothesis was based on the prior work of Nagappan et al., Caglayan et al., Mockus and Bell et al., all of whom observed that there was a negative correlation between development team size and fault-proneness (Nagappan et al., 2008; Mockus, 2010; Bell et al., 2013; Caglayan et al., 2015). Given that fault-proneness and structural metrics are correlated, and in the absence of any code-level insights, it was considered a plausible hypothesis that all structural attributes of software and team size would show a similar correlation which would be indicative of lower maintainability.

The results showed that those attributes that had been established to correlate to fault-proneness - coupling, cohesion, modularity - did, indeed, show degradation as team sizes increased. This was consistent with prior research and a confirmation of the alternative hypothesis. The drivers behind trends of degraded coupling and cohesion could be hypothesised to be attributable to the difficulties of effective communication in a larger team leading to developers with conflicting design patterns or alternate approaches causing degraded structural attributes. Through qualitative analysis and engaging development team members, the research community could shed more light as to the drivers behind the trends revealed in this study.

The alternative hypotheses (H1,2.1, H1,2.2) that less stable teams exhibit degraded structural metrics and lower maintainability was based on research by Huckman et al. that found that greater team stability resulted in lower fault-proneness and higher budget adherence (Huckman et al., 2009). The rationale then followed, as with the previous alternative hypothesis (H1,1.1), that this was caused by a general degradation in the internal structural

Table 6.1 Summary of objectives and outcomes.

Objective	Outcome
O1,1: *Observe how structural metrics progress as software projects evolve.*	Established probability distributions of CK metrics and correlation matrices relating metrics to team factors. No clear metric trends against code volume. Clear trends against code revisions.
O1,2 *Control for confounding factors.*	Ruled out code volume as a confounding factor to the analysis in this research. Controlling for revisions by adopting an approach of bucketing metrics by revision count and team size such that populations with the same revision count are compared to one another.
O1,3: *Formulate a definition of the software development team size and analyse the impact of this factor on the structural metrics.*	Defining development team size as the cumulative number of committers to commit to a project. Mann-Whitney tests on the bucketed metric populations, and linear regression techniques observing that increasing team sizes yield decreasing measures of inheritance complexity, coupling, cohesion and modularity
O1,4: *Deduce the likely result of the impact of team size on the maintainability of software.*	Increasing team sizes result in structural metrics that have been associated in prior research with degraded levels of the maintainability sub-attributes of testability, changeability and testability.
O2,1: *Identify and mitigate the pitfalls associated with mining software repositories for the purposes of team stability analysis*	A challenge to social network analysis in consistently tracking committers as they traverse through the forge as multiple user identifiers are occasionally used by the same committer. Strategy to rationalise to a single identifier. Threat to validity emanating from forked projects distorting VCS log history. Developed a heuristic to searches for common commits across projects, identifying them as related as parent-fork and excluded from study.
O2,2: *Formulate a definition of the software development team stability and analyse the impact of this factor on the structural metrics.*	Proposal of a calculation to quantify intra-project team stability - a measure capturing the degree of stability accrued by a team through the course of a project. Observing that increasing team stability yield decreasing measures of structural complexity, coupling, cohesion and modularity.
O2,3: *Deduce the likely result of the impact of team stability on the maintainability of software.*	Analysis indicates that lower team stability causes structural attributes to trend in a direction that is associated with degradation in the sub-attributes of maintainability.

Table 6.2 Summary of null hypotheses and results.

Null Hypothesis	Result
H0,1: Development team size does not impact the coupling, complexity, cohesion or modularity of the produced software.	Projects developed by larger team sizes exhibited an increase in coupling (reflected by larger CBO values), an increase in inheritance complexity (reflected by higher DIT values) and a decrease in cohesion (reflected by larger LCOM values). This is a rejection of the null hypothesis H0,1.
H0,2: Development team stability does not impact the coupling, complexity, cohesion or modularity of the produced software.	There is a noticeable upward trend across DIT, LCOM, RFC, WMC metrics confirming that teams with lower stability produce projects with lower cohesion and higher structural complexity. This is a rejection of the null hypothesis.

Table 6.3 Summary of research questions and answers.

Research Question	Answer
RQ1: What is the impact of team size on the structural properties of software and its resultant maintainability?	Greater team sizes result in degraded measures of cohesion, coupling and structural complexity and enhanced measures modularity. These observations are consistent with degradation in maintainability.
RQ2: What is the impact of team stability on the structural properties of software and its resultant maintainability?	Greater team stability results in enhanced measures of cohesion and structural complexity. As a result, it can be inferred the maintainability will be enhanced.

attributes of the software. Higher inter-team and intra-team stability was found to be linked to higher measures of cohesion and modularity, and lower measures of coupling and inheritence complexity. Again, through engaging development team members through surveys in conjunction with a detailed analysis of individual commits, further insights could be gained into the factors driving these observations.

Table 6.3 summaries the headline results against each of the research questions. Broadly speaking, smaller development team sizes and greater team stability result in software with enhanced structural attributes. However, larger team sizes were found to be linked to lower functional complexity which is also associated with enhanced maintainability.

6.3 Thesis Contributions

The contributions in this work span three broad areas: furthering the art in the field of mining software repositories, the unique aspects of the linear modelling approach in this thesis and, finally, insights into the impact of team factors on the structural attributes of software.

6.3.1 Open-Source Forge Analysis

As discussed in the 'Related Work' chapter, the practical challenges of mining open-source forges is well documented in existing research. Thematically this work covers tooling, pitfalls and insights from the perspective of researchers. Separately to this there is work that performs static analysis on selected open-source projects and, also, extracts insights which can help process improvement in some fashion. This research focuses on a single open-source forge and spans both of these areas of research with several specific contributions.

- **GoogleCode forge analysis:** This research studies an entire forge revealing several insights which go to a greater level of granularity than existing open-source forge research which tends to derive insights at the forge and project level (Howison et al., 2009; Iqbal et al., 2012; Squire, 2017). Committer behaviour across the forge was studied as it informed the approach to the team size and stability analysis. It was possible, through this, to establish the population of unique committers (identity reconciliation challenges notwithstanding as documented in Section 5.2.1) and track their activity throughout the history of the forge. This enabled observations on the number of projects that committers contribute to as well as the nature of their contributions (number of commits, number of files per commit and timeframes of activity). With this information it was possible to identify where committers collaborated on multiple projects, crucial to the team stability sample identification.

  This was a one of many potential use-cases for this type of data set and social network analysis in open-source is an active field of research with plenty of other use-cases (Hassan, 2008; Hemmati et al., 2013). For a representative sample of the forge it was then possible to mine structural metrics and join those observations to the commit meta-data (through the definition of a relational database schema). Through studying observations from 173,190 class files in the team size sample it was confirmed that CK metrics from projects of this forge follow heavily skewed non-normal distributions in-line with the work of Succi et al. and Basili et al. who had conducted their research

on a smaller disparate data sets (Basili et al., 1996; Succi et al., 2005). The reported moderate collinearity between each of CBO (capturing cohesion and modularity) RFC and WMC (capturing structural complexity) was also confirmed.

- **Identifying forked projects:** Forking presents a number of perils when not properly considered. These perils can affect researchers extracting insights by mining VCS histories, developers identifying projects to contribute to and end-users looking to utilise a project artefact. This research is concerned with the first use-case and has, for the first time, highlighted forking as a threat to the validity of network analysis in open-source forges. Existing research focusses on project forking for the purposes of studying the its motivations and outcomes (Robles et al., 2006; Nyman and Mikkonen, 2011). This research presents an approach to the automated identification of forks avoiding the costly code clone analysis proposed by Ray and Kim (Ray et al., 2012) and relying solely on VCS logs which are much less costly to mine.

  The approach presented was practically applied to mining the entirety of the GoogleCode forge and the threat to validity that forking poses was quantified with 6.25% of projects in the forge found to be forked and therefore showing commit history which is misleading when taken out of context. The approach presented is not limited to GoogleCode but can be generalised to all forges which use SVN or GIT as their underlying VCS.

6.3.2 Modelling approach

There are several noteworthy aspects to the nature of the linear models used to capture the trends observed in the GoogleCode forge that constitute a contribution to knowledge.

- **Modelling CK metrics as a dependent variable:** Existing research casts CK metrics as the *independent* variable and models their relationship with the externally observable attributes of software as documented extensively in the related work sections 2.3.3-2.3.4. Within this research, for the first time, these CK metrics are treated as the *dependent* variables and the impact of team factors on these variables are modelled. This type of analysis allows existing models to be brought to bear in order to deduce the likely impact of these team factors on the externally observable attributes of software. For instance, this research concludes that LCOM has a positive linear relationship with team size. Bruntink et al. and Badri et al. both found LCOM to be an inverse predictor

of testablity (Bruntink and van Deursen, 2006; Badri et al., 2011). It is therefore it is reasonable to suggest that team size is likely to have an inverse relationship with the testability of the produced software.

Through this modelling approach, it was discovered that revisions are a confounding factor when studying the impact of team factors on CK metrics. This complements both the work of Emam et al. and Zhou et al. who found that controlling for size was essential when modelling CK metrics as the *independent* variable (El Emam et al., 2001; Zhou and Leung, 2006).

- **Definition of team stability:** Huckman established a general approach to measuring team stability using the cumulative time that team members worked together (Huckman et al., 2009). In this thesis the Huckman approach was transposed onto a quantitative measure of stability that can be derived through the analysis of VCS logs alone, lending itself to open-source forge studies. This work presented a distinction between *intra*-project and *inter*-project stability: respectively the stability accrued through the course of an individual project and that gained through the collaboration of the project team on multiple projects. Intra-project stability was captured through the formulation of the Lack of Stability Ratio (LSR) and the practical application of this measure to a representative sample from the GoogleCode forge was documented within this thesis.

6.3.3 Team Factor Analysis

At the core of this thesis lies the established relationship between the team factors of size and stability and the structural metrics of software.

- **Team size trends:** Prior research established team size as a key determinant of project success with models finding a relationship between team size and both lower productivity and increased fault-proneness. This thesis contributes to the art by also establishing team size as a significant predictor to CK metric values, shedding light on the potential underlying effects that drive the externally observable attributes. Both the correlation analysis and the linear models showed a positive relationship between team size and all metrics with the exception of NOC. The effect of team size was particularly marked in DIT and LCOM with team size accounting for almost half the variance in these metrics. This work applies linear mixed models to a multi-project metrics study for the first time finding a strong idiosyncratic 'project-specific' aspect that

outweighs the team size effect. Given the models summarised in Table 2.4, this implies an inverse relationship between team size and the sub-attributes of maintainability which is generally consistent with the prior literature in the field.

- **Team stability trends:** Huckman et al. studied a dataset comprising over 1000 projects and found that increased team stability yielded lower fault-proneness. Through the course of this thesis it was established empirically that there is a positive correlation between a *lack* of team stability and CK metrics which, in turn would be associated with a deterioration in the sub-attributes of maintainability and fault-proneness. Through the discovery of these trends this work sheds a light on the underlying effects that could potentially have driven Huckman's observations (or indeed those of Gardner et al. who found that increased team stability was linked to an increase in client satisfaction (Gardner et al., 2012)).

6.4 Threats to Validity

This section covers the internal and external threats to validity affecting this research.

6.4.1 Threats to External Validity

- **Development models** Open-source projects tend to have a particular dynamic which sees a limited number of core contributors taking a central role while the majority of contributors take a peripheral role and do not engage in projects for extended periods of time (Howison et al., 2006). This can contrast with commercial software which often is developed by a relatively engaged development team. In either development approaches there could core contributors who act as gatekeepers into the version control system and mandate a review and approval process prior to any commits becoming part of the main source code repository. It can be hypothesised that this could ultimately impact the structural attributes of the software. For instance, a central core of experienced committers with a good knowledge of the system could provide guidance on component reuse where such opportunities may otherwise have not been exploited. Within the GoogleCode repository there is no project meta-data that can be exploited to inform on these factors.

- **Physical locations** Brooks posits that larger teams face greater difficulties in communication compared to smaller teams and hence productivity is impacted. However, one significant factor that influences the efficiency of communication is the physical location of the members of a team. Teasley argues that collocating a team can double productivity (Teasley et al., 2000). Factoring in the physical setup of the development teams contributing to the studied projects is beyond the scope of this research.

- **Development languages** It is important to re-state that this research only takes into account observations that can be made of Java code, to the exclusion of all other languages. It is possible that there may be factors that cause particular trends that are observed in Java software to be absent from software developed in other object-oriented languages. It could be argued that, as the results observed in this study are generally consistent with the vast body of research (research which does span multiple OO languages such as C++, Java and Smalltalk), it is unlikely to be a significant threat to validity. However, nonetheless caution should be exercised when applying the lessons learnt in this study to projects written in other languages.

- **Generalisation to other forges** While this work uses a large data set (certainly in comparison to similar studies) there was no attempt to establish how the forge chosen for this study, GoogleCode, could differ in nature to other forges. While there is no indication that GoogleCode is, indeed, biased towards any particular influencing factor, it cannot be ruled out as a threat to validity. It is also worth noting that, while this work succeeded in observing broad trends across the forge, it is noted that not all projects within the samples followed these trends. There are a number of factors that can impact the structural attributes a particular project, not captured in this study, and which may affect the generalisation of this work.

6.4.2 Threats to Internal Validity

- **Linking structural attributes to external attributes** Part of this research is to infer the impact of the observations made of the structure of software on its externally observable attributes. This inference draws on the work of prior researchers who have established correlations between metric values and aspects of maintainability. For that reason, any inferences drawn in this work inherit the threats to validity in that prior research. Where we link CK metric observations to external attributes of understandability and changeability, it is necessary to state the limitations of the research conducted by Harrison et al, whose work first established correlations between

metric values and these external attributes: namely that their use of small student projects as a data set constitutes a threat to validity when applying these results to larger commercial or open-source projects (Harrison et al., 1998b). Likewise Bruntink and van Deursen, whose work established correlations between CK metrics and testability, note that their criteria for evaluating testability may not be applicable to all projects (Bruntink and van Deursen, 2006).

- **Functional nature of projects** This threat to validity concerns the mix of server-side and client-side programming that exists, particularly, in those projects with a User Interface (UI). Best practices dictate that the UI should be a thin layer with the business logic existing in a server-side service layer. In a typical Java project, the server-side programming would be in Java while the client side could be in any one of a number of scripting languages such as JavaScript. As this UI layer is not studied in this research, there is a 'blind spot' whose size could vary according to degree of adherence to best practices. For instance, if larger teams tend to adhere to the principle of 'separation of concerns' to a greater degree than smaller teams, it could be that greater complexity observed in the server side is incorrectly attributed to the larger teams while, in fact, that complexity also exists in the smaller team's work - only that it exists in the client-side scripting code. This is depicted, in basic terms in Figure 6.2.
- **Team sizing model** The project comparison analysis in Chapters 4 and 5 showed that, while we attribute a single figure to the size of a development team based on the cumulative number of project committers, the reality of the matter is more nuanced. Individual committers or groups of committers are often responsible for a disproportionate contribution to the project and there is a loss of information when this behaviour is reduced to a single number. In Section 4.1.1 it is argued that the contribution of so-called 'peripheral' committers could not be ignored as they are undoubtedly an influence on the codebase of a project. However, it can be equally argued that not quantifying or factoring in the nature of the committer contributions does not allow us to draw a potentially important distinction between projects with differing proportions of core and peripheral developers.

6.5 Conclusions

As mentioned in section 1.4 of this thesis, the overarching research problem that this work aims solve for is the difficulty in appropriately sizing and resourcing software development

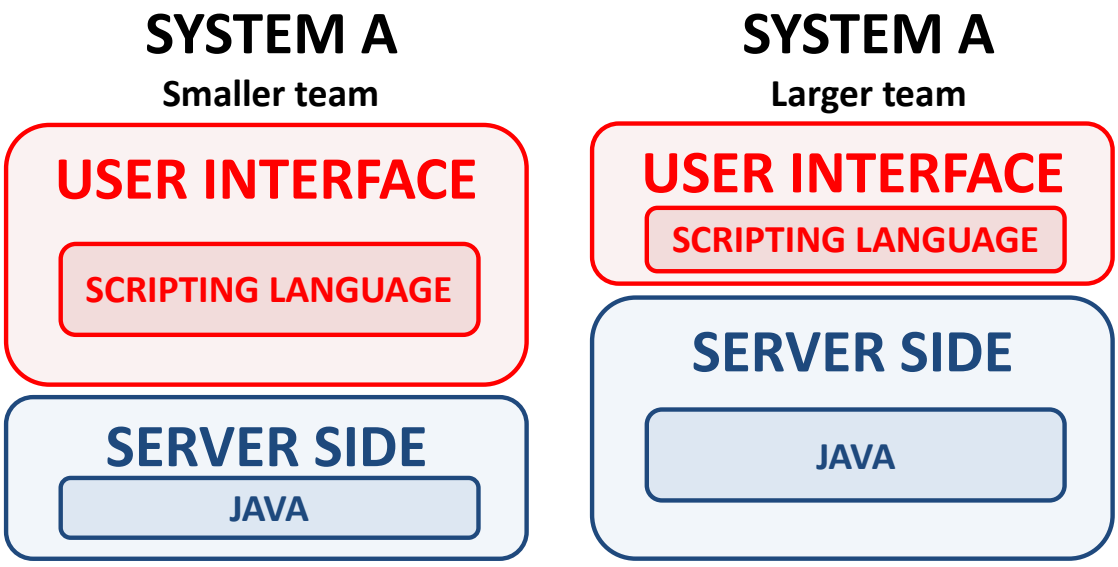


Fig. 6.2 An illustration of how a larger team could produce software with a greater adherence to the separation of concerns with more of the complexity residing on the server side.

teams to achieve optimal performance and stakeholder satisfaction. It was discussed how organisations often had multiple options available to them in this regard; for instance forming new large teams or seconding existing small stable teams.

This work helps inform practitioner decision-making by providing greater insight into the impact that team composition can have on the sub-attributes of maintainability. Rather than simply observing a relationship between team factors and externally observable attributes, this work provides an insight into how large or unstable teams lead to the degradation of the internally observable attributes which consequently drive the degradation of externally observable attributes including maintainability. As stated by Fenton and Bieman, practitioners are accustomed to measuring and monitoring internal attributes throughout the development process (Fenton and Bieman, 2014). Internally observable attributes of software can be measured and monitored in real-time through tools such as Sonar enabling practitioners to monitor and mitigate risks that may arise from unstable or large teams (SonarQube, 2018).

While the focus on team size and stability provided insights that it is hoped will be of value to both the research community and practitioners, there were other significant contributions within this work. A methodology was put forward to enable the study of externally observable attributes of software through the internal structural attributes of software, leveraging existing research to tie both strands together. In doing so, practitioners can focus on specific, measurable structural attributes as a mitigation strategy against some of the observed negative trends that can result from team factors. Furthermore, this work brings relevance to existing research. By way of example, Chidamber et al. (Chidamber et al., 1998) find that higher measures of coupling and lower levels cohesion are indicators of lower productivity, greater rework and design effort. These structural attributes are side-effects of larger team sizes, so it is reasonable to deduce that the aforementioned degradation in productivity, rework and design effort are likely to also result from larger teams.

While studying the impact of team factors by studying the GoogleCode forge, many practical difficulties were encountered in mining a large and diverse forge. In particular, performing accurate and reliable network analysis for the purposes of the team stability analysis was particularly challenging and it is hoped that the work in this thesis to mitigate these threats to validity will prove helpful to the broader research community.

6.6 Reflections

The author has a number of personal reflections on this work from the vantage point of both a researcher and an experienced practitioner.

- **Challenges mining open-source forges:** A significant proportion of the effort expended in this work has been in obtaining, cleaning and modelling the data set. This is not unusual; a recent survey of data scientists reported that 70% of their time is typically spent collecting, labelling, cleaning, organising and modelling data while only 10% is spent mining data for patterns (CrowdFlower, 2017). The author was struck by the logistical complexity of mining a large forge despite the clean interface that Version Control Systems expose to make both code and meta-data available. For instance, retreival of revision logs across 236,787 projects necessitated several iterations of the VCS mining scripts in order to parallelise the data retrieval process. It was also necessary to host the mining software on a remote virtual server in order to achieve the requisite download speeds to complete the retrieval process in a sensible time-frame. Similarly, executing network analysis on such a large data set presented its own challenges in writing the software in a suitably memory efficient manner. Once the data set was sampled and distilled to an easily consumable form, conducting statistical tests was easier than the author had anticipated. This was due, in large part, to the ease of use of data analysis tools such as the Anaconda data science workbench and the many open-source python libraries with their active developer communities (Anaconda, 2018).

- **Future roles for machine learning and big data:** The vast quantities of data and meta-data that are available within software repositories are arguably under-utilised and should be leveraged to improve success rates of software projects in industry. As discussed in this thesis, this data could be used to identify, in real-time, potential threats to the externally observable attributes of software. While software metrics have seen some industry adoption, particularly through the use of Sonar in enforcing so-called 'quality-gates' that commits must pass become part of the software (Ampatzoglou et al., 2018; SonarQube, 2018), the use-cases of management adoption of structural metrics remain few and far between. There exists a gulf between the empirical approach to the study of software by the research community and the approach of practitioners which is often less structured and less empirical. One of the aims of this research is to help, in whatever small way, bridge this gap. One key reflection that the author has from this

thesis is that driving a more empirically-oriented approach to software development is key to its continuing maturity as an engineering discipline.

- **Reproducibility of results:** While surveying the prior literature, it became evident that reproducibility of results in the field of mining software repositories is a particular challenge. Some academic research draws upon closed-source software, not available to the research community, to research industrial case studies. Even where open-source software is the subject of study, often the literature inadequately documents the methodology; a real hindrance to anyone attempting to validate the work or, indeed, replicate the study to serve as a baseline for their own study. Given the burgeoning interest in data science, the research community would do well to make curated data sets available to the broader community through GitHub or specialised data set sharing and collaboration platforms such as Kaggle to crowd source efforts to uncover insights and help drive greater interest in industry adoption of metrics (GITHub, 2018; Kaggle, 2018).

6.7 Future Work

To support the network analysis that drove the team stability analysis, there was effort to establish parent-fork project relationships throughout the GoogleCode forge. This effort was limited to those relationships that had the potential to skew the inter-project team stability analysis and therefore focussed exclusively on those forks that had been established through a direct clone of a repository from within the GoogleCode forge itself. As a separate strand of research, there is value in a broader and more detailed study of forking relationships. To support this broader study of forking relationships and to also drive a more comprehensive study of open-source developer contributions, it is worthwhile to consider what shape a cross-forge data mining study may take. In this section a Hadoop-based architecture is proposed as a potential solution to analysing the vast data sets that a cross-forge mining effort would yield.

6.7.1 Network Analysis

When studying open-source repositories, there are a number of perspectives from which data about project forking relationships drive value. Capiluppi et al. proposed an approach to

quantify committer contributions to open-source projects and assign a measure similar to the H-index used in the academic world (Capiluppi et al., 2012). Any assessment of committer contributions will be skewed if equal weight is assigned to commits that import work from other projects versus a committer's original work. This is particularly pertinent given that there has been a recent upsurge in the use of open-source repositories such as GitHub for identifying and recruiting software development talent. To a recruiter it is essential that individuals are not being attributed credit for work that is simply imported from other projects as bad hiring decisions can be very difficult to reverse.

Successful projects can garner a large number of forks and occasionally the fork can overtake the parent in popularity such as in the case of Firefox forked from Mozilla and Ubuntu forked from Debian. Clearly there will be some visibility on forks which garner a large amount of activity but many forks fail to make a significant contribution at all. It will remain an open question as to how many forks fail due to a lack of innovation and how many fail simply through lack of visibility in the wider community but we believe that providing visibility on the alternate development streams would help inform developer and user decision-making. For example, a particular set of customisations present on a fork may make that project more attractive to a certain user community. From a contributor's perspective, a highly productive, yet smaller, developer community may be more attractive to join. According to a recent survey by Jiang et al. 42% of developers believe that there is value in an automated recommendation tool to assist in choosing repositories to fork (Jiang et al., 2017).

Based on the lessons learnt from the narrow study of project relationships in this thesis a broader solution is presented that aims to achieve several goals:

- **Plug-in heuristics** The network analysis in the previous chapter used a heuristic based on 'common commits' to identify project forks. That particular heuristic relied on the observation of multiple commits with identical meta-data across projects as evidence of the forking relationship. This heuristic is particularly suited to identifying projects which have the potential to skew VCS based network analysis within a single forge but this is by no means the only useful heuristic in the broader context of mining. By way of example, adding additional mining commit comments or code clone detection will yield a more comprehensive analysis.
- **Directional relationships** The network analysis in this thesis did not look at the directionality of the relationship between related projects. There was no attempt to discern the parent from the fork as this data was not relevant to the mining of 'stable project pairs'. However, from the perspective of a project stakeholder, this information

is relevant. A basic heuristic which could be applied to this problem could identify the project with the greatest number of commits as the parent is likely to be the more active project. However there are, of course, exceptions to this norm.

- **Visualisations** While the raw analysis mapping out parent-fork project relationships is a useful input to those looking to conduct detailed network analysis or measure developer contributions, it is not an appropriate output for project stakeholders looking to select a project for adoption as a user or a contributor. For this reason, a visualisation can provide value for complex project network graphs.

To solve for these challenges a simple bespoke framework was prototyped which was based on several of the design patterns and components in the toolchain that was presented earlier in this thesis. Figure 6.3 presents a UML depiction of the class structure of the implementation. At the top-level a Controller is responsible for reading commit data from the various implementations of CommitLoader - retrieved in the form of the object graph described in the Commit interface and its dependencies - and passes them into the implementations of the Heuristic which returns an instance of Result containing all the projects found to be forked from others. The heuristic interface is flexible enough to allow for the capture of meta-data such as a measure of the code volume of original content within a fork. For visualization, a node network graph renderer can generate visualisations for the node relationships as illustrated in Figure 6.4. For this prototype the Arbor.js framework was used as it is open-source and fairly basic, although there are several advanced network graphing tools.

There are a number of avenues that the research community could pursue by way of furthering this work. The first is to consider the design of this framework and consider basing a tool around this, creating additional heuristics to hone the detection of forks in an open-source forge. In particular, there could be value in leveraging some of the research in the field of code-clone detection for this purpose as there is significant overlap between the challenges of code-cloning and the mapping project relationships. Furthermore, it is recommended that open-source forges - particularly any newcomers entering the market - further their web-based tools to provide visualisations on fork relationships between projects. This could be made less of a burdensome task by mandating developers to provide data around the key relationships upon project initiation.

It is worth noting that since the research in this thesis was conducted, Ren et al. have produced a beta implementation of a GitHub fork visualisation tool called Forks Insight which carries out some of what is described in this section, albeit against a single forge (Ren

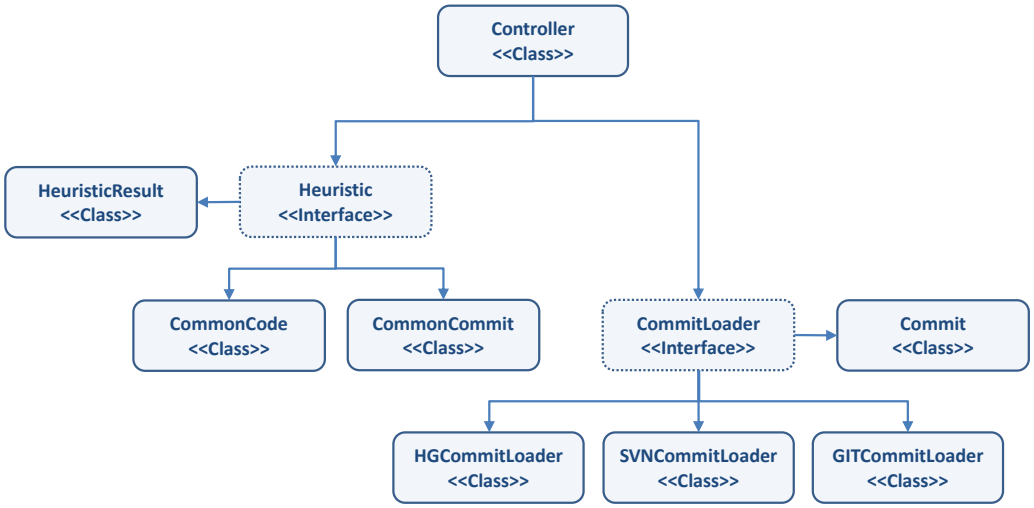


Fig. 6.3 UML Diagram illustrating structure of the prototype of heuristics framework

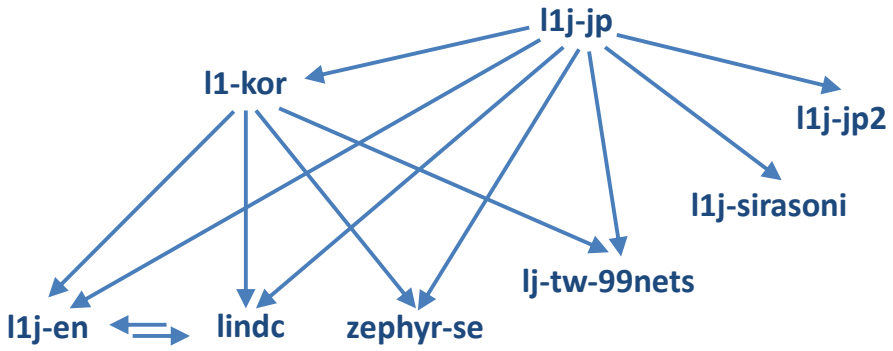


Fig. 6.4 An example network graph using the example of the Lineage project which is an online multiplayer role playing game which is particularly popular in Korea and Japan. The projects illustrated are mostly language variants of Lineage server emulators. There are a number of additional forks on the periphery of the diagram which represent development streams which did not garner much activity.

et al., 2018). It will certainly be exciting to see if this project gains adoption in the wider community.

6.7.2 Cross-Forge Data Mining

While this and other research shows that there can be significant value derived through studying an individual forge, it is worth also drawing attention to some of the limitations of narrowly focusing on a single forge. Relationships between those projects within the forge to projects outside of the forge cannot be captured which could lead to incomplete network analysis or incorrect assessments of committer contributions. Similarly, the code cloning research discussed in Chapter 5 relies on traversing multiple forges to capture data from the major hubs of open-source activity.

Mining multiple forges also opens opportunities to observe trends of community engagement, committer traversal and product quality, potentially highlighting where particular forges can improve or benefit from offering richer toolsets to their customers. This may have implications for an end-users choice of forge or indeed to the forge's business model. For example, if established that a particular forge attracts a more experienced and committed development community than their peers, they might prefer the GitHub business model of offering paid hosting plans to individuals and businesses rather than the SourceForge ad-supported business model.

A number of challenges arise when attempting to mine across forges. While the toolchain presented in this thesis provides cross-VCS support, cross-forge support would be required in those components that drive the VCS mining. Specifically, the FlossMole artefacts would need to be added or updated for the additional forges to be included in the analysis. Crowston and Squire have recently documented the challenge facing FlossMole to 'continually provide broader access and more sophisticated and relevant data and analyses' (Crowston and Squire, 2017). Furthermore, any data that would need to be mined from the forge webpages themselves would necessitate the encoding of particular parsing logic. In the case of this research, the VCS repository links were parsed from the project homepages in GoogleCode. If this were to become a pattern, one could envisage a rules-driven mining tool that enables the addition of forges for mining without the need for software build effort.

A greater challenge arises from the vast volumes of data that mining multiple forges would produce. By way of example, if we were to add the contents of GitHub, SourceForge and

GoogleCode this would amount to approximately 20 million projects. If each project were to take up 500KB of data storage - which would equate to the size of a fairly modest VCS log - the total storage required would be over a terabyte. This is well into the realms of what is commonly termed 'Big Data'. With these types of data sets, relational databases can prove inefficient and costly. Kononenko et al. created a framework based on an Apache Cassandra-based big data solution to aid rapid open-source code searching (Kononenko et al., 2014). An industry standard approach to enable the storage and deep analysis of large data sets of this nature is a Hadoop-based architecture.

Hadoop is an open-source framework designed to distribute computing and data storage using cheap off-the-shelf commodity hardware eliminating the need for costly vendor-specific machines (Apache, 2018a). Underlying Hadoop is the core concept of 'data locality' which encourages essentially combines both the data and processing layer avoiding costly shifting of data to bring it to the computing logic. The Hadoop ecosystem is built on the Hadoop Distributed File System (HDFS) and provides frameworks such as Spark which facilitate rapid and rich distributed data processing (Apache, 2018b). Figure 6.5 depicts a basic interaction between Spark and a Hadoop cluster illustrating the role of the resource manager in delegating computing to the individual Data Nodes where Spark workers execute processing directly on the data.

The research community should consider pooling effort and resourcing to implement such architecture to facilitate cross-forge mining.

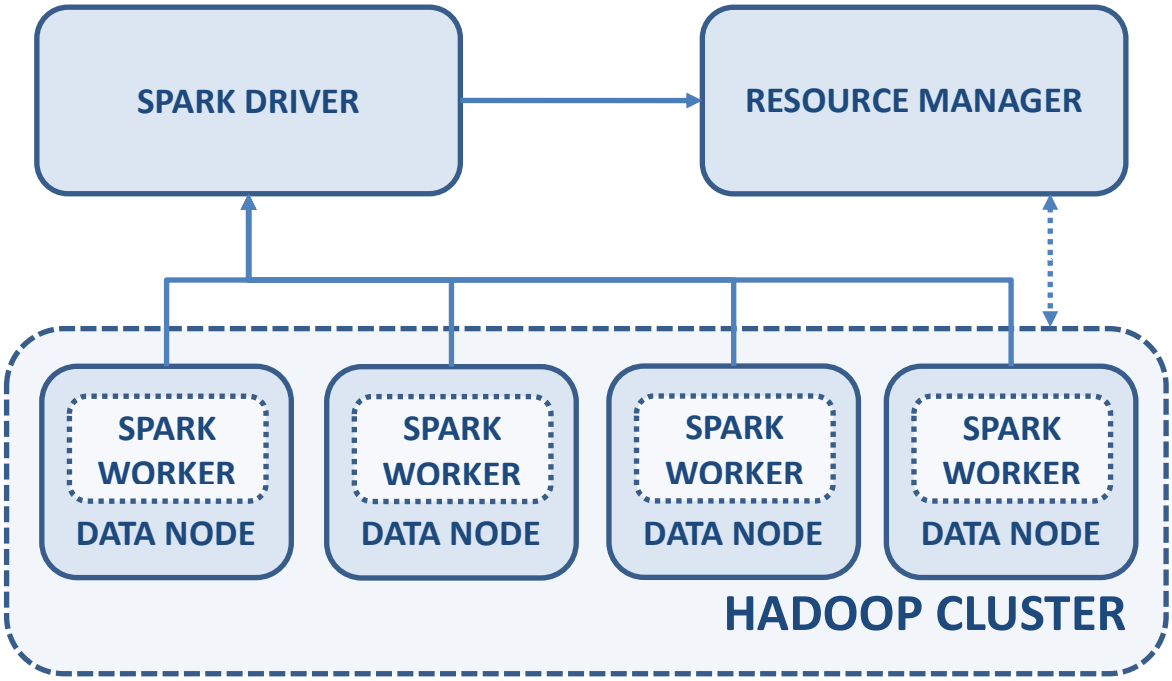


Fig. 6.5 A basic illustration of a Hadoop architecture.

References

Abounader, J. R. and Lamb, D. A. (1997). A data model for object-oriented design metrics. *Retrieved May*, 26:2005.

Abrahamsson, P. (2013). Measuring the success of software process improvement: the dimensions. *arXiv preprint arXiv:1309.4645*.

Abreu, F. B. and Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, volume 186, pages 1–8.

Agrawal, A., Rahman, A., Krishna, R., Sobran, A., and Menzies, T. (2018). We don't need another hero?: the impact of heroes on software development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 245–253. ACM.

Akgün, A. E. and Lynn, G. S. (2002). Antecedents and consequences of team stability on new product development performance. *Journal of Engineering and Technology Management*, 19(3):263–286.

Akiyama, F. (1971). An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359.

Alfayez, R., Behnamghader, P., Srisopha, K., and Boehm, B. (2018). An exploratory study on the influence of developers in technical debt. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 1–10. ACM.

Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., and Avgeriou, P. (2018). A framework for managing interest in technical debt: An industrial validation.

Anaconda (2018). *Anaconda*. https://www.anaconda.com accessed 2018-09-16.

Andrejczuk, E., Rodríguez-Aguilar, J. A., Roig, C., and Sierra, C. (2017). Synergistic team composition. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 1463–1465. International Foundation for Autonomous Agents and Multiagent Systems.

Apache (2018a). *Apache Hadoop*. http://hadoop.apache.org accessed 2018-09-16.

Apache (2018b). *Apache Spark*. http://spark.apache.org accessed 2018-09-16.

Assembla (2018). *Assembla*. http://www.assembla.com accessed 2018-09-16.

Badri, L., Badri, M., and Toure, F. (2011). An empirical analysis of lack of cohesion metrics for predicting testability of classes. *International Journal of Software Engineering and Its Applications*, 5(2):69–85.

Bagnato, A., Barmpis, K., Bessis, N., Cabrera-Diego, L. A., Di Rocco, J., Di Ruscio, D., Gergely, T., Hansen, S., Kolovos, D., Krief, P., et al. (2017). Developer-centric knowledge mining from large open-source software repositories (crossminer). In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 375–384. Springer.

Baker, J., Cunei, A., Flack, C., Pizlo, F., Prochazka, M., Vitek, J., Armbruster, A., Pla, E., and Holmes, D. (2006). A real-time java virtual machine for avionics-an experience report. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 384–396. IEEE.

Balla, K., Bemelmans, T., Kusters, R., and Trienekens, J. (2001). Quality through managed improvement and measurement (qmim): Towards a phased development and implementation of a quality management system for a software company. *Software Quality Journal*, 9(3):177–193.

Bao, L., Xing, Z., Xia, X., Lo, D., and Li, S. (2017). Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 170–181. IEEE.

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

Basili, V. R. and Perricone, B. T. (1984). Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52.

Behrens, C. A. (1983). Measuring the productivity of computer systems development activities with function points. *IEEE Transactions on Software Engineering*, (6):648–652.

Bell, R. M., Ostrand, T. J., and Weyuker, E. J. (2013). The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering*, 18(3):478–505.

Bernardi, M. L., Canfora, G., Di Lucca, G. A., Di Penta, M., and Distante, D. (2018). The relation between developers communication and fix-inducing changes: An empirical study. *Journal of Systems and Software*, 140:111–125.

Bevan, J., Whitehead Jr, E. J., Kim, S., and Godfrey, M. (2005). Facilitating software evolution research with kenyon. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 177–186. ACM.

Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM.

Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. (2009). The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE.

Blackburn, J., Lapré, M. A., and Van Wassenhove, L. N. (2006). Brooks' law revisited: improving software productivity by managing complexity.

Boehm, B. W., Brown, J. R., and Kaspar, H. (1978). Characteristics of software quality.

Bolker, B. M., Brooks, M. E., Clark, C. J., Geange, S. W., Poulsen, J. R., Stevens, M. H. H., and White, J.-S. S. (2009). Generalized linear mixed models: a practical guide for ecology and evolution. *Trends in ecology & evolution*, 24(3):127–135.

Bonferroni, C. E. (1936). *Teoria statistica delle classi e calcolo delle probabilita*. Libreria internazionale Seeber.

Boydston, R. E. (1984). Programming cost estimate: Is it reasonable? In *Proceedings of the 7th international conference on Software engineering*, pages 153–159. IEEE Press.

Britton, T., Jeng, L., Carver, G., Cheak, P., and Katzenellenbogen, T. (2013). Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*.

Brixtel, R., Fontaine, M., Lesner, B., Bazin, C., and Robbes, R. (2010). Language-independent clone detection applied to plagiarism detection. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 77–86. IEEE.

Brooks, F. (1986). The mythical man-month. *Tutorial*, 11:35–42.

Bruntink, M. and van Deursen, A. (2006). An empirical study into class testability. *Journal of systems and software*, 79(9):1219–1232.

Caglayan, B., Turhan, B., Bener, A., Habayeb, M., Miransky, A., and Cialini, E. (2015). Merits of organizational metrics in defect prediction: an industrial replication. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 89–98. IEEE.

Capiluppi, A. and Beecher, K. (2009). Structural complexity and decay in floss systems: An inter-repository study. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 169–178. IEEE.

Capiluppi, A., Boldyreff, C., Beecher, K., and Adams, P. J. (2009). Quality factors and coding standards–a comparison between open source forges. *Electronic Notes in Theoretical Computer Science*, 233:89–103.

Capiluppi, A., Serebrenik, A., and Youssef, A. (2012). Developing an h-index for oss developers. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 251–254. IEEE.

Capra, E. and Wasserman, A. I. (2008). A framework for evaluating managerial styles in open source projects. In *IFIP International Conference on Open Source Systems*, pages 1–14. Springer.

Carthey, J., de Leval, M. R., and Reason, J. T. (2001). The human factor in cardiac surgery: errors and near misses in a high technology medical domain. *The Annals of thoracic surgery*, 72(1):300–305.

Cartwright, M. and Shepperd, M. (2000). An empirical investigation of an object-oriented software system. *IEEE Transactions on software engineering*, 26(8):786–796.

Chen, C., Li, K., Ouyang, A., Zeng, Z., and Li, K. (2018). Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1275–1288.

Chidamber, S. R., Darcy, D. P., and Kemerer, C. F. (1998). Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on software Engineering*, 24(8):629–639.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object oriented design.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493.

Chong, C. Y. and Lee, S. P. (2015). Analyzing maintainability and reliability of object-oriented software using weighted complex network. *Journal of Systems and Software*, 110:28–53.

Chopra, A., Verma, P., and Puri, S. (2018). Empirical study on impact of developer collaboration on source code.

Choukse, D., Kanellopoulos, D. N., and Singh, U. K. (2012). Developing secure web applications. *International Journal of Internet Technology and Secured Transactions*, 4(2-3):221–236.

Chow, T. and Cao, D.-B. (2008). A survey study of critical success factors in agile software projects. *Journal of systems and software*, 81(6):961–971.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

Comino, S., Manenti, F. M., and Parisi, M. L. (2005). From planning to mature: on the determinants of open source take off. Technical report, University of Trento.

Conaldi, G. and Lomi, A. (2013). The dual network structure of organizational problem solving: A case study on open source software development. *Social Networks*, 35(2):237–250.

Correia, J. P., Kanellopoulos, Y., and Visser, J. (2009). A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 61–70. IEEE.

Côté, V., Bourque, P., Oligny, S., and Rivard, N. (1988). Software metrics: an overview of recent results. *Journal of Systems and Software*, 8(2):121–131.

CrowdFlower (2017). *CrowdFlower, 2017 Data Scientist Report*.

Crowston, K. and Squire, M. (2017). Lessons learned from a decade of floss data collection. In *Big Data Factories*, pages 79–100. Springer.

Čubranić, D. and Murphy, G. C. (2003). Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international Conference on Software Engineering*, pages 408–418. IEEE Computer Society.

CVS (2018). *CVS*. http://www.nongnu.org/cvs accessed 2018-09-16.

Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM.

Daniel, S. and Stewart, K. (2016). Open source project success: Resource access, flow, and integration. *The Journal of Strategic Information Systems*, 25(3):159–176.

De Alwis, B. and Sillito, J. (2009). Why are software projects moving from centralized to decentralized version control systems? In *Proceedings of the 2009 ICSE Workshop on cooperative and human aspects on software engineering*, pages 36–39. IEEE Computer Society.

Deemer, P., Benefield, G., Larman, C., and Vodde, B. (2010). The scrum primer. *Scrum Primer is an in-depth introduction to the theory and practice of Scrum, albeit primarily from a software development perspective, available at: http://assets. scrumtraininginstitute. com/downloads/1/scrumprimer121. pdf*, 1285931497:15.

Deshpande, A. and Riehle, D. (2008). The total growth of open source. In *IFIP International Conference on Open Source Systems*, pages 197–209. Springer.

DiBona, C. (2015). *Bidding farewell to Google Code*. Google. https://opensource.googleblog.com/2015/03/farewell-to-google-code.html accessed 2018-09-16.

Eichhorn, H., Cano, J. L., McLean, F., and Anderl, R. (2018). A comparative study of programming languages for next-generation astrodynamics systems. *CEAS Space Journal*, 10(1):115–123.

Eilhard, J. and Ménière, Y. (2009). A look inside the forge: Developer productivity and spillovers in open source projects.

El Emam, K., Benlarbi, S., Goel, N., and Rai, S. (1999). *A validation of object-oriented metrics*. National Research Council Canada, Institute for Information Technology.

El Emam, K., Melo, W., and Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75.

Elish, M. O. and Rine, D. (2003). Investigation of metrics for object-oriented design logical stability. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 193–200. IEEE.

English, M. and Mc Creanor, P. (2009). Exploring the differing usages of programming language features in systems developed in c++ and java.

Erickson, C. (2012). *Small Teams Are Dramatically More Efficient than Large Teams*. https://spin.atomicobject.com/2012/01/11/small-teams-are-dramatically-more-efficient-than-large-teams/ accessed 2018-09-16.

Ernst, N. A. (2018). Bayesian hierarchical modelling for tailoring metric thresholds. *arXiv preprint arXiv:1804.02443*.

Fenton, N. and Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC Press.

Foucault, M., Teyton, C., Lo, D., Blanc, X., and Falleri, J.-R. (2015). On the usefulness of ownership metrics in open-source software projects. *Information and Software Technology*, 64:102–112.

Gaffney Jr, J. E. (1981). Metrics in software quality assurance. In *Proceedings of the ACM'81 conference*, pages 126–130. ACM.

Gangwar, P., Kumar, S., and Rastogi, N. (2014). Web solution using more secure apache http server with the concept of full virtualization. *International Journal of Computer Applications*, 98(22).

Gardner, H. K., Gino, F., and Staats, B. R. (2012). Dynamically integrating knowledge in teams: Transforming resources into performance. *Academy of Management Journal*, 55(4):998–1022.

German, D. M. (2004). Mining cvs repositories, the softchange experience. *Evolution*, 245(5,402):92–688.

German, D. M., Cubranić, D., and Storey, M.-A. D. (2005). A framework for describing and understanding mining tools in software development. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM.

Gharehyazie, M., Ray, B., Keshani, M., Zavosht, M. S., Heydarnoori, A., and Filkov, V. (2018). Cross-project code clones in github. *Empirical Software Engineering*, pages 1–36.

Ghofrani, J., Mohseni, M., and Bozorgmehr, A. (2017). A conceptual framework for clone detection using machine learning. In *Knowledge-Based Engineering and Innovation (KBEI), 2017 IEEE 4th International Conference on*, pages 0810–0817. IEEE.

Gil, Y. and Lalouche, G. (2017). On the correlation between size and metric validity. *Empirical Software Engineering*, 22(5):2585–2611.

GIT (2018). *GIT*. https://git-scm.com accessed 2018-09-16.

GITHub (2018). *GITHub*. http://www.github.com accessed 2018-09-16.

GITLab (2018). *GITLab*. http://www.gitlab.com accessed 2018-09-16.

Glass, R. L. (2006). The standish report: does it really describe a software crisis? *Communications of the ACM*, 49(8):15–16.

GNU (2007). *General Public License, Free Software Foundation*, 13:2010.

Goeminne, M. and Mens, T. (2013). A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986.

Gomathi, S. and Linda Edith, P. (2013). An overview of object-oriented metrics. a complete survey. *International Journal of Computer Science Engineering Technology*, 4(9).

GoogleCode (2018). *GoogleCode*. http://www.googlecode.com accessed 2018-09-16.

Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., and Ghezzi, C. (2010). An empirical investigation into a large-scale java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 11. ACM.

Gyimothy, T., Ferenc, R., and Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910.

Hackman, J. R. (2002). *Leading teams: Setting the stage for great performances*. Harvard Business Press.

Halstead, M. H. (1977). Elements of software science. operating and programming systems series, vol. 2.

Harrison, R., Counsell, S. J., and Nithi, R. V. (1998a). An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496.

Harrison, R., Counsell, S. J., and Nithi, R. V. (1998b). An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering*, 3(3):255–273.

Harter, D. E., Kemerer, C. F., and Slaughter, S. A. (2012). Does software process improvement reduce the severity of defects? a longitudinal field study. *IEEE Transactions on Software Engineering*, 38(4):810–827.

Hassan, A. E. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE.

Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE.

Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., and Godfrey, M. W. (2013). The msr cookbook: Mining a decade of research. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 343–352. IEEE.

Heričko, M., Živkovič, A., and Rozman, I. (2008). An approach to optimizing software development team size. *Information Processing Letters*, 108(3):101–106.

Hoch, J. E., Pearce, C. L., and Welzel, L. (2010). Is the most effective team leadership shared? *Journal of Personnel Psychology*.

Howison, J., Conklin, M., and Crowston, K. (2009). Flossmole: A collaborative repository for floss research data and analyses. In *Software Applications: Concepts, Methodologies, Tools, and Applications*, pages 85–94. IGI Global.

Howison, J. and Crowston, K. (2004). The perils and pitfalls of mining sourceforge. In *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, pages 7–11. IET.

Howison, J., Inoue, K., and Crowston, K. (2006). Social dynamics of free and open source team communications. In *IFIP International Conference on Open Source Systems*, pages 319–330. Springer.

Hsia, P., Hsu, C.-T., and Kung, D. C. (1999). Brooks' law revisited: a system dynamics approach. In *Computer Software and Applications Conference, 1999. COMPSAC'99. Proceedings. The Twenty-Third Annual International*, pages 370–375. IEEE.

Huckman, R. and Staats, B. (2013). The hidden benefits of keeping teams intact. *Harvard business review*, 91(12):27.

Huckman, R. S., Staats, B. R., and Upton, D. M. (2009). Team familiarity, role experience, and performance: Evidence from indian software services. *Management science*, 55(1):85–100.

Humphrey, W. S. (2005). Why big software projects fail: The 12 key questions.

Hussain, S., Keung, J., Khan, A. A., and Bennin, K. E. (2016). Detection of fault-prone classes using logistic regression based object-oriented metrics thresholds. In *Software Quality, Reliability and Security Companion (QRS-C), 2016 IEEE International Conference on*, pages 93–100. IEEE.

Iqbal, A., Cyganiak, R., and Hausenblas, M. (2012). Integrating floss repositories on the web. Technical report, Technical Report, Digital Enterprise Research Institute (DERI), NUIG. Available from: http://www. deri. ie/sites/default/files/publications/iqbal_a_et_al_dec_2012. pdf.(Cited on pages 12, 22, and 105.).

Iqbal, M. A. (2015). *Large scale data integration of OSS repositories for automated soft and technical factors assessment*. PhD thesis.

ISO 15504-5:2012 (2012). Information technology - process assessment. Standard, International Organization for Standardization, Geneva, CH.

ISO 9126-1:2001 (2001). Product quality - part 1: Quality model. Standard, International Organization for Standardization, Geneva, CH.

Jaccard, P. (1901). Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bull. Soc. Vaud. Sci. Nat.*, 37:241–272.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

Jermakovics, A., Sillitti, A., and Succi, G. (2011). Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 24–31. ACM.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P. S., and Zhang, L. (2017). Why and how developers fork what from whom in github. *Empirical Software Engineering*, 22(1):547–578.

Johari, K. and Kaur, A. (2012). Validation of object oriented metrics using open source software system: an empirical study. *ACM SIGSOFT Software Engineering Notes*, 37(1):1–4.

Joshi, K., Hernandez, J., Martinez, J., AbdelFattah, K., and Gardner, A. K. (2018). Should they stay or should they go now? exploring the impact of team familiarity on interprofessional team training outcomes. *The American Journal of Surgery*, 215(2):243–249.

Juergens, E., Deissenboeck, F., and Hummel, B. (2009). Clonedetective-a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering*, pages 603–606. IEEE Computer Society.

Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131.

Kaggle (2018). *Kaggle*. http://www.kaggle.com accessed 2018-07-10.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM.

Kappelman, L. A., McKeeman, R., and Zhang, L. (2006). Early warning signs of it project failure: The dominant dozen. *Information systems management*, 23(4):31–36.

Kiefer, C., Bernstein, A., and Tappolet, J. (2007). Mining software repositories with isparol and a software evolution ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 10. IEEE Computer Society.

Kitchenham, B. (2010). Whats up with software metrics? a preliminary mapping study. *Journal of systems and software*, 83(1):37–51.

Kolmogorov, A. (1933). Sulla determinazione empirica di una leggi di distribuzione. *Giorn. 1st it lit o Ital. Attuari, 4 (1933), 83*, 91.

Kononenko, O., Baysal, O., Holmes, R., and Godfrey, M. W. (2014). Mining modern repositories with elasticsearch. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 328–331. ACM.

Kpodjedo, S., Ricca, F., Galinier, P., Antoniol, G., and Guéhéneuc, Y.-G. (2013). Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of Software: Evolution and Process*, 25(2):139–163.

Krishnan, M. S. (1998). The role of team factors in software cost and quality: An empirical analysis. *Information Technology & People*, 11(1):20–35.

Kuhrmann, M., Konopka, C., Nellemann, P., Diebold, P., and Münch, J. (2015). Software process improvement: where is the evidence?: initial findings from a systematic mapping study. In *Proceedings of the 2015 International Conference on Software and System Process*, pages 107–116. ACM.

Lalsing, V., Kishnah, S., and Pudaruth, S. (2012). People factors in agile software development and project management. *International Journal of Software Engineering & Applications*, 3(1):117.

Lee, A., Carver, J. C., and Bosu, A. (2017). Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: a survey. In *Proceedings of the 39th International Conference on Software Engineering*, pages 187–197. IEEE Press.

Lee, M.-W., Roh, J.-W., Hwang, S.-w., and Kim, S. (2010). Instant code clone search. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 167–176. ACM.

Lewis, W. E. (2016). *Software testing and continuous quality improvement*. CRC press.

Li, W. and Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122.

Linders, B. (2011). *Establishing and maintaining stable teams*. https://www.benlinders.com/2011/establishing-and-maintaining-stable-teams/ accessed 2018-09-16.

Livshits, B. and Zimmermann, T. (2005). Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 296–305. ACM.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

Lozano, A. and Wermelinger, M. (2008). Assessing the effect of clones on changeability. *International Conference on Software Maintenance*.

Lozano, A., Wermelinger, M., and Nuseibeh, B. (2007). Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 18. IEEE Computer Society.

Ludwig, J., Xu, S., and Webber, F. (2017). Compiling static software metrics for reliability and maintainability from github repositories. In *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*, pages 5–9. IEEE.

Macheronne, L. (2013). *Quantifying the Impact Of Agile Practices*. http://maccherone.com/publications/KEYNOTE-MetricsAndInsights-Maccherone-0604-1245PM.pdf accessed 2018-09-16.

Madeyski, L. and Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal*, 23(3):393–422.

Maillart, T. and Sornette, D. (2016). Aristotle vs. ringelmann: A response to scholtes et al. on superlinear production in open source software. *arXiv preprint arXiv:1608.03608*.

Malhotra, R. and Jain, A. (2012). Fault prediction using statistical and machine learning methods for improving software quality. *Journal of Information Processing Systems*, 8(2):241–262.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

Martinez-Romo, J., Robles, G., Gonzalez-Barahona, J. M., and Ortuño-Perez, M. (2008). Using social network analysis techniques to study collaboration between a floss community and a company. In *IFIP International Conference on Open Source Systems*, pages 171–186. Springer.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

McCabe, T. J. and Butler, C. W. (1989). Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425.

Mcconnell, S. (2018). *Less is More, Jumpstarting Productivity with Small Teams*. http://www.stevemcconnell.com/articles/art06.htm accessed 2018-09-16.

McLeod, L. and MacDonell, S. G. (2011). Factors that affect software systems development project outcomes: A survey of research. *ACM Computing Surveys (CSUR)*, 43(4):24.

Meccia, M. (2015). *Critical Success Factors for Application Development*. https://www.linkedin.com/pulse/critical-success-factors-application-development-part-meccia-1 accessed 2018-09-16.

Menard, S. (2002). *Applied logistic regression analysis*, volume 106. Sage.

Meneely, A. and Williams, L. (2009). Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462. ACM.

MetricsProject (2018). *Metrics Eclipse Plug-in*. http://metrics.sourceforge.net accessed 2018-09-16.

Miller, J. (2006). *Want productivity? Try some team continuity (and a side of empowerment too)*. http://codebetter.com/jeremymiller/2006/05/06/want-productivity-try-some-team-continuity-and-a-side-of-empowerment-too/ accessed 2018-09-16.

Mirheidari, S. A., Arshad, S., Khoshkdahan, S., and Jalili, R. (2012). Two novel server-side attacks against log file in shared web hosting servers. In *Internet Technology And Secured Transactions, 2012 International Conference for*, pages 318–323. IEEE.

Mockus, A. (2010). Organizational volatility and its effects on software defects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–126. ACM.

Mockus, A., Fielding, R. T., and Herbsleb, J. (2000). A case study of open source software development: the apache server. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 263–272. IEEE.

MSDN (2015). *Code Metrics Value*. Microsoft Developer Network. https://msdn.microsoft.com/en-us/library/bb385914.aspx accessed 2018-09-16.

Nagappan, N., Murphy, B., and Basili, V. (2008). The influence of organizational structure on software quality. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 521–530. IEEE.

Nasir, M. H. N. and Sahibuddin, S. (2011). Critical success factors for software projects: A comparative study. *Scientific research and essays*, 6(10):2174–2186.

nDepend (2018). *nDepend*. http://www.ndepend.com accessed 2018-09-16.

Nesi, P. and Querci, T. (1998). Effort estimation and prediction of object-oriented systems. *Journal of Systems and Software*, 42(1):89–102.

Nguyen, V., Deeds-Rubin, S., Tan, T., and Boehm, B. (2007). A sloc counting standard. In *COCOMO II Forum*, volume 2007, pages 1–16.

Nyman, L. and Mikkonen, T. (2011). To fork or not to fork: Fork motivations in sourceforge projects. In *IFIP International Conference on Open Source Systems*, pages 259–268. Springer.

O'Grady, S. (2011). *What Black Duck Can Tell Us About GitHub, Language Fragmentation and More*. http://redmonk.com/sograd y/2011/06/02/blackduck-webinar/ accessed 2018-09-16.

Okutan, A. and Yıldız, O. T. (2014). Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181.

Oliveira, P., Valente, M. T., Bergel, A., and Serebrenik, A. (2015). Validating metric thresholds with developers: An early result. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 546–550. IEEE.

Onoue, S., Hata, H., Kula, R. G., and Matsumoto, K. (2018). Human capital in software engineering: A systematic mapping of reconceptualized human aspect studies. *arXiv preprint arXiv:1805.03844*.

Ordonez, M. J. and Haddad, H. M. (2008). The state of metrics in software industry. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 453–458. IEEE.

Ortu, M., Hall, T., Marchesi, M., Tonelli, R., Bowes, D., and Destefanis, G. (2018). Mining communication patterns in software development: A github analysis.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Paulk, M. C., Curtis, B., Chrissis, M. B., and Weber, C. V. (1993). Capability maturity model, version 1.1. *IEEE software*, 10(4):18–27.

Pearson, K. (1901). Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572.

Pendharkar, P. C. and Rodger, J. A. (2009). The relationship between software development team size and software development cost. *Communications of the ACM*, 52(1):141–144.

Pirzadeh, L. (2010). Human factors in software development: a systematic literature review.

Plowman, N. (2015). *Seven Factors of Effective Team Performance*. http://www.brighthubpm.com/monitoring-projects/62415-seven-factors-of-effective-team-performance/ accessed 2018-09-16.

Portillo-Rodríguez, J., Vizcaíno, A., Piattini, M., and Beecham, S. (2012). Tools used in global software engineering: A systematic mapping review. *Information and Software Technology*, 54(7):663–685.

PowerSoftware (2018). *Power Software*. http://www.powersoftware.com accessed 2018-09-16.

Prather, R. E. (1984). An axiomatic theory of software complexity measure. *The Computer Journal*, 27(4):340–347.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.

Radatz, J., Geraci, A., and Katki, F. (1990). Ieee standard glossary of software engineering terminology. *IEEE Std*, 610121990(121990):3.

Ragkhitwetsagul, C., Krinke, J., and Marnette, B. (2018). A picture is worth a thousand words: Code clone detection based on image similarity. In *Software Clones (IWSC), 2018 IEEE 12th International Workshop on*, pages 44–50. IEEE.

Rainer, A. and Hall, T. (2002). Key success factors for implementing software process improvement: a maturity-based analysis. *Journal of Systems and Software*, 62(2):71–84.

Ray, B., Wiley, C., and Kim, M. (2012). Repertoire: A cross-system porting analysis tool for forked software projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 8. ACM.

Ray, P. P. (2017). A survey on visual programming languages in internet of things. *Scientific Programming*, 2017.

RedMonk (2011). *What Black Duck Can Tell Us About GitHub, Language Fragmentation and More*. http://redmonk.com/sogrady/2011/06/02/blackduck-webinar/ accessed 2018-09-16.

Rempel, P. and Mäder, P. (2017). Preventing defects: the impact of requirements traceability completeness on software quality. *IEEE Transactions on Software Engineering*, 43(8):777–797.

Ren, L., Zhou, S., and Kästner, C. (2018). Forks insight: providing an overview of github forks. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 179–180. ACM.

RhodeCode (2016). *Version Control Systems Popularity in 2016*. https://rhodecode.com/insights/version-control-systems-2016.

Robles, G. (2004). Remote analysis and measurement of libre software systems by means of the cvsanaly tool. IET.

Robles, G. and Gonzalez-Barahona, J. M. (2005). Developer identification methods for integrated data from various sources. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5.

Robles, G. and González-Barahona, J. M. (2012). A comprehensive study of software forks: Dates, reasons and outcomes. In *IFIP International Conference on Open Source Systems*, pages 1–14. Springer.

Robles, G., Gonzalez-Barahona, J. M., Michlmayr, M., and Amor, J. J. (2006). Mining large software compilations over time: another perspective of software evolution. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9. ACM.

Rodger, J. A., Pankaj, P., Nahouraii, A., et al. (2011). Knowledge management of software productivity and development time. *Journal of Software Engineering and Applications*, 4(11):609.

Rodriguez, D., Sicilia, M., Garcia, E., and Harrison, R. (2012). Empirical findings on team size and productivity in software development. *Journal of Systems and Software*, 85(3):562–570.

Rosenberg, L. H. (1998). Applying and interpreting object oriented metrics.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495.

Saberwal, H. K., Singh, S., and Kaur, S. (2013). Empirical analysis of open source system for predicting smelly classes. In *International Journal of Engineering Research and Technology*, volume 2. ESRSA Publications.

Santos, L. D. C., Saraiva, R. M., Perkusich, M., de Almeida, H. O., and Perkusich, A. (2017). An empirical study on the influence of context in computing thresholds for chidamber and kemerer metrics. In *SEKE*.

Sauer, C. and Cuthbertson, C. (2003). The state of it project management in the uk 2002-2003.

Scheer, A.-W. and Habermann, F. (2000). Enterprise resource planning: making erp a success. *Communications of the ACM*, 43(4):57–61.

Schmidt, R., Lyytinen, K., and Mark Keil, P. C. (2001). Identifying software project risks: An international delphi study. *Journal of management information systems*, 17(4):5–36.

Scholtes, I., Mavrodiev, P., and Schweitzer, F. (2016). From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects. *Empirical Software Engineering*, 21(2):642–683.

Schwarz, N., Lungu, M., and Robbes, R. (2012). On how often code is cloned across repositories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1289–1292. IEEE Press.

Schweik, C. M., English, R. C., Kitsing, M., and Haire, S. (2008). Brooks' versus linus' law: an empirical test of open source projects. In *Proceedings of the 2008 international conference on Digital government research*, pages 423–424. Digital Government Society of North America.

SciTools (2018). *SciTools*. http://www.scitools.com accessed 2018-09-16.

SEC (2013). Administrative proceeding file no. 3-15570. Securities and Exchange Commission.

Shankland, S. (2006). *Google launches open-source repository*. CNET. https://www.cnet.com/uk/news/google-launches-open-source-repository/ accessed 2018-09-16.

Sharma, A. K., Kalia, A., and Singh, H. (2012). Metrics identification for measuring object oriented software quality. *International Journal of Soft Computing and Engineering*, 2(5):255–258.

Sharp, D. C., Pla, E., Luecke, K. R., and Hassan, R. (2003). Evaluating real-time java for mission-critical large-scale embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 30–36. IEEE.

Shatnawi, R. (2010). A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on software engineering*, 36(2):216–225.

Smirnov, N. (1948). Table for estimating the goodness of fit of empirical distributions. *The annals of mathematical statistics*, 19(2):279–281.

Smith, R. K., Hale, J. E., and Parrish, A. S. (2001). An empirical study using task assignment patterns to improve the accuracy of software effort estimation. *IEEE Transactions on software engineering*, 27(3):264–271.

SonarQube (2018). *SonarQube*. https://www.sonarqube.org accessed 2018-09-16.

Song, Q., Guo, Y., and Shepperd, M. (2018). A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*.

Sørensen, T. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Biol. Skr.*, 5:1–34.

Spearman, C. (1904). The proof and measurement of association between two things. *American journal of Psychology*, 15(1):72–101.

Spinellis, D. (2005). Tool writing: a forgotten art? *IEEE Software*, 22(4):9–11.

Spinellis, D. and Gousios, G. (2018). How to analyze git repositories with command line tools: we're not in kansas anymore. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 540–541. ACM.

Squire, M. (2009). Integrating projects from multiple open source code forges. In *Database Technologies: Concepts, Methodologies, Tools, and Applications*, pages 2301–2312. IGI Global.

Squire, M. (2017). The lives and deaths of open source code forges. In *Proceedings of the 13th International Symposium on Open Collaboration*, page 15. ACM.

StackOverflow (2018). *StackOverflow*. http://www.stackoverflow.com accessed 2018-09-16.

Subramanyam, R. and Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310.

Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., and Russo, B. (2005). An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104.

Sulayman, M., Urquhart, C., Mendes, E., and Seidel, S. (2012). Software process improvement success factors for small and medium web companies: A qualitative study. *Information and Software Technology*, 54(5):479–500.

Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 99–108. ACM.

Tamburri, D. A., Palomba, F., Serebrenik, A., and Zaidman, A. (2018). Discovering community patterns in open-source: A systematic approach and its evaluation. *Empirical Software Engineering*.

Tang, M.-H., Kao, M.-H., and Chen, M.-H. (1999). An empirical study on object-oriented metrics. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 242–249. IEEE.

Teasley, S., Covi, L., Krishnan, M. S., and Olson, J. S. (2000). How does radical collocation help a team succeed? In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 339–346. ACM.

Terceiro, A., Rios, L. R., and Chavez, C. (2010). An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In *Software Engineering (SBES), 2010 Brazilian Symposium on*, pages 21–29. IEEE.

Tian, Y., Tan, H., and Lin, G. (2018). Statistical properties analysis of file modification in open-source software repositories. In *Proceedings of the International Conference on Geoinformatics and Data Analysis*, pages 62–66. ACM.

Timm, N. H. (2002). Applied multivariate analysis. *Springer*.

TIOBE (2017). *TIOBE Index for February 2017*. http://www.tiobe.com/tiobe-index/ accessed 2018-09-16.

Tiwari, N. M., Upadhyaya, G., Nguyen, H. A., and Rajan, H. (2017). Candoia: A platform for building and sharing mining software repositories tools as apps. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 53–63. IEEE.

Tosun, A., Ahmed, M., Turhan, B., and Juristo, N. (2018). On the effectiveness of unit tests in test-driven development. In *Proceedings of the 2018 International Conference on Software and System Process*, pages 113–122. ACM.

Trendowicz, A. and Münch, J. (2009). Factors influencing software development productivitystate-of-the-art and industrial experiences. *Advances in computers*, 77:185–241.

Tuckman, B. W. (1965). Developmental sequence in small groups. *Psychological bulletin*, 63(6):384.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2017). When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088.

Tukey, J. W. (1977). Exploratory data analysis.

Vijayasarathy, L. R. and Butler, C. W. (2016). Choice of software development methodologies: Do organizational, project, and team characteristics matter? *IEEE software*, 33(5):86–94.

Von Krogh, G., Haefliger, S., Spaeth, S., and Wallin, M. W. (2012). Carrots and rainbows: Motivation and social practice in open source software development. *MIS quarterly*, 36(2):649–676.

Wasserman, A. I., Guo, X., McMillian, B., Qian, K., Wei, M.-Y., and Xu, Q. (2017). Osspal: finding and evaluating open source software. In *IFIP International Conference on Open Source Systems*, pages 193–203. Springer.

Weyuker, E. J., Ostrand, T. J., and Bell, R. M. (2008). Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559.

Wiegmann, D. A., Eggman, A. A., ElBardissi, A. W., Parker, S. H., and Sundt, T. M. (2010). Improving cardiac surgical care: a work systems approach. *Applied ergonomics*, 41(5):701–712.

Xenos, M., Stavrinoudis, D., Zikouli, K., and Christodoulakis, D. (2000). Object-oriented metrics-a survey. In *Proceedings of the FESMA*, pages 1–10.

Xiong, Y., Meng, Z., Shen, B., and Yin, W. (2017). Mining developer behavior across github and stackoverflow. In *SEKE*, pages 578–583.

Xu, J., Ho, D., and Capretz, L. F. (2008). An empirical validation of object-oriented design metrics for fault prediction.

Yamashita, A., Abtahizadeh, S. A., Khomh, F., and Guéhéneuc, Y.-G. (2017). Software evolution and quality data from controlled, multiple, industrial case studies. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 507–510. IEEE.

Yeh, M.-L., Chu, H.-P., and Lue, P. (2005). Influences of team longevity and stability on r&d performance. *IJEBM*, 3(3):209–213.

Youssef, A. (2012). Impact of collaboration on structural software quality. In *Proceedings of the 10th International Symposium on Open Collaboration*. ACM.

Youssef, A. and Capiluppi, A. (2015). The impact of developer team sizes on the structural attributes of software. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pages 38–45. ACM.

Zhou, X. and Stephens, M. (2012). Genome-wide efficient mixed-model analysis for association studies. *Nature genetics*, 44(7):821–824.

Zhou, Y. and Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on software engineering*, 32(10):771–789.