Investigating the architectural drivers of defects in open-source software systems: an empirical study of defects and reopened defects in GNOME

By
Ali Almossawi

B.Sc., Computer Systems Engineering
University of East Anglia, 2005
M.S., Software Engineering
Carnegie Mellon University, 2007

Submitted to the System Design and Management program in partial fulfillment of the requirements for the degree of Master of Science in Engineering and Management at the Massachusetts Institute of Technology June 2012

© 2012 Ali Almossawi. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author’s signature .................................................. Ali Almossawi
System Design and Management Program
Engineering Systems Division

Certified by .................................................. Alan D. MacCormack
Adjunct Professor, Harvard Business School, Harvard University
Formerly, Visiting Associate Professor, Sloan School of Management
Thesis Supervisor

Accepted by .................................................. Patrick Hale
Senior Lecturer, Engineering Systems Division
Director, System Design and Management Program
To my parents
To my wife and best friend, Danah
Page intentionally left blank
Investigating the architectural drivers of defects in open-source software systems: an empirical study of defects and reopened defects in GNOME

By
Ali Almossawi

Submitted to the System Design and Management Program on May 11, 2012 in partial fulfillment of the requirements for the degree of Master of Science in Engineering and Management

Abstract
In major software systems that are developed by competent software engineers, the existence of defects in production is unlikely to be an acceptable situation. And yet, we find that in several such systems, defects remain a reality. Furthermore, the number of changes that are fixed only to then be reopened is noticeable. The implications of having defects in a system can be frustrating for all stakeholders, and when they require constant rework, they can lead to the problematic code-test-code-test mode of development. For management, such conditions can result in slipped schedules and an increase in development costs and for upper management and users, they can result in losing confidence in the product.

This study looks at the drivers of defects in the mature open-source project GNOME and explores the relationship between the various drivers of these defects and software quality. Using defect-activity and source-code data for 32 systems over a period of eight years, the work presents a multiple regression model capable of explaining 16.2% of defects and a logistic regression model capable of explaining between 13.6% and 18.1% of reopened defects. The study also shows that although defects in general and reopened defects appear to move together, defects in general correlate with a measure of complexity that captures how components connect to each other whereas reopened defects correlate with a measure that captures the inner complexities of components, thereby suggesting that different types of defects are correlated with different forms of complexity.

Thesis advisor: Alan D. MacCormack
Title: Adjunct Professor, Harvard Business School, Harvard University
Formerly, Visiting Associate Professor, Sloan School of Management
Acknowledgements

I remain grateful for the guidance and accessibility of my thesis advisor, Alan MacCormack. I would on occasion find myself in the dark and barren land of intellectual vacuity and am grateful for the calmness with which Alan would guide me out of it every time.

I would also like to thank Andre Klapper from the GNOME Bugsquad for his insights and enthusiasm and for answering the questions that I had about the Evolution project so quickly as well as Dan Sturtevant from MathWorks for writing the original versions of several of the scripts used for data processing.

I am also indebted to the sources of love, joy and support in my life: my father, whom I try very hard to be a mirror image of, my mother from whom I learnt hard work, my brother who is an inseparable part of me, my wife, without whom I couldn’t imagine life, and my beautiful daughter, Fatima, who would quietly sit on my lap while I typed this document.
# Table of Contents

**Chapter 1: Introduction** .................................................................................................................. 6  
1.1 Why look at defects? ......................................................................................................................... 6  
1.2 Why look at reopened defects? ....................................................................................................... 7  
1.3 Research questions ......................................................................................................................... 8  
1.4 Organization of the remainder of the document ........................................................................... 10  

**Chapter 2: Software quality and complexity** ......................................................................................... 11  
2.1 Defects in software ........................................................................................................................ 11  
2.2 Tracking defects ............................................................................................................................ 13  
2.3 Measuring software quality .......................................................................................................... 14  
   2.3.1 Defect density ......................................................................................................................... 15  
   2.3.2 Defective fixes ....................................................................................................................... 16  
   2.3.3 Time-to-fix ............................................................................................................................ 17  
2.4 Measuring system complexity ....................................................................................................... 17  
   2.4.1 Lines of code ........................................................................................................................ 18  
   2.4.2 Cyclomatic complexity ....................................................................................................... 19  
   2.4.3 Design structure matrices and fan-in/fan-out metrics ....................................................... 21  
   2.4.4 Visibility matrices and propagation cost ............................................................................ 24  
2.5 Limitations of software metrics .................................................................................................. 29  

**Chapter 3: Method** .......................................................................................................................... 30  
3.1 The dataset .................................................................................................................................. 30  
3.2 Choosing the systems .................................................................................................................... 31  
3.3 Variables of interest ....................................................................................................................... 33  
3.4 Descriptive statistics .................................................................................................................... 34  
3.5 Collecting and processing the data ............................................................................................... 36  
3.6 Source-code data ........................................................................................................................ 37  
3.7 Defect activity data ....................................................................................................................... 38  
3.8 Removing singletons ..................................................................................................................... 43  

**Chapter 4: Results** .......................................................................................................................... 44  
4.1 Preliminary analysis ....................................................................................................................... 44  
4.2 Statistical models .......................................................................................................................... 46  
4.3 Taking a look at reopened defects ............................................................................................... 51  

**Chapter 5: Discussion and practical implications** .............................................................................. 59  

**References** ....................................................................................................................................... 64  

**Appendices** ..................................................................................................................................... 69  
Appendix A: Data-processing steps .................................................................................................... 70  
Appendix B: First-order matrix to visibility matrix ............................................................................ 73  
Appendix C: Full correlation table ..................................................................................................... 74  
Appendix D: Subset of dataset showing dependencies ...................................................................... 75  

vii
Chapter 1

Introduction

1.1 Why look at defects?

In mature software systems that are developed by competent software engineers, the existence of defects in production is unlikely to be an acceptable situation. And yet, we find that in several such systems, defects remain a reality. Furthermore, the number of changes that are fixed only to then be reopened is significant. For example, in GTK+, a popular open-source toolkit, about 1.35% of fixed bugs were reopened one or more times at some point, and in Evolution, a popular open-source email client, it is around 3.88%. Anecdotal evidence reveals a similar situation in large-scale enterprise systems as well. The implications of having changes that need to be reworked multiple times can be frustrating for all stakeholders - managers, users and developers - and can lead to the problematic code-test-code-test mode of development. For management, it can result in high development costs and slipped schedules as well as in upper management and users losing confidence in the product.

A wide body of work going back as far as several decades reveals that there are numerous factors that can influence the existence of defects in software systems. Brooks in his seminal work observed that lack of calendar time and faulty specifications can lead to defects (Brooks 1975), McCabe in his seminal paper stated that his particular measure of complexity can help improve software quality (McCabe 1976), Halstead suggested that defects were a function of a software system’s volume (Halstead 1977), Hutchens and Basili (Hutchens, D.H. and Basili 1985) and Selby and Basili (Selby and Basili 1991)
showed that components with higher coupling were more likely to have defects, Sosa et al. argued for component loops as good predictors of defects (Sosa, Mihm and Browning 2009), MacCormack et al. argued for a measure of direct and indirect dependencies, called propagation cost (MacCormack, Rusnak and Baldwin 2006, MacCormack, Rusnak and Baldwin 2008), Cataldo et al. showed that intra-work, communication-based dependencies between developers was perhaps a better predictor of defects than syntactic dependencies (Cataldo, et al. 2009) and Sliwersky et al. showed that the day of week that changes were worked on determined whether or not new defects were introduced (Sliwerski, Zimmerman and Zeller 2009). Other studies have found that employee productivity, the defect report’s clarity and the number of people involved in working on a defect all influence the existence of defects in a system and the length of time needed to fix them (McConnell 2004, Bettenburg, et al. 2008, Anbalagan and Vouk 2009, Akaikine 2010).

Furthermore, defects put a serious strain on organizations in the form of cost and reputation. Maintaining software after it is released is itself a costly endeavor (MacCormack and Sturtevant 2011) and can in fact constitute over 90% of total costs (Seacord, Plakosh and Lewis 2003). Defects add to that cost by demanding more time and effort. In more serious circumstances, defects can result in catastrophic losses as was the case with the explosion of the $500 million Ariane 5 rocket 50 seconds after takeoff (Jézéquel and Meyer 1997) and the fatal consequences of the Therac-25 radiation therapy machine (Leveson 1993).

1.2 Why look at reopened defects?

Despite this extensive literature examining defects and their predictors, very few studies have considered the distinction between defects in general and reopened defects, which one may contend are more damaging to a software system’s performance and reputation. Of the 46,800 confirmed defects in our dataset, 44,278 of them were fixed at some point (94.61%) and of those 951 were reopened after being fixed (2.15%). Quick scans
through other mature software projects that use the Bugzilla defect-tracking system reveals similar numbers, which although are not staggeringly high, are still significant given that these are mature systems.

Shihab et al. were perhaps one of the first to look at reopened defects within the context of the Eclipse project and observed that a defect report’s comment text, its description text, the time it took to fix the defect and the impacted component were the most important factors in deciding whether a defect got reopened (Shihab, et al. 2010). Zimmermann et al. did a similar study on the Windows operating system and found the main factors to be process-related (Zimmermann, et al. 2012). This study aims to add to the limited body of knowledge on reopened defects via an empirical investigation of the GNOME open-source project.

1.3 Research questions

The study began as exploratory work and then evolved into a structured attempt to find answers to some open questions. The first of these questions looks at the inputs to a system within the context described so far and asks:

RQ1: What are the architectural drivers of defects and reopened defects in GNOME?

Previous work has found an array of architectural and non-architectural factors to be predictors of defects. Herein, the primary class of factors that are explored is architecture, which builds on prior studies that argue for using coupling, an inter-component concept, as a measure of complexity, as well as studies that use intra-component measures such as cyclomatic complexity. Given that quantitative data can possibly abstract away the subtleties of reality, this first research question shall also consider qualitative factors.
The second research question intends to see if predictive models may be developed to explain both types of defects. It therefore goes a step further and asks:

**RQ2: Can we predict defects and reopened defects in GNOME?**

The third research question asks if reopened defects are similar to defects in general:

**RQ3: Are defects in general and reopened defects in GNOME fundamentally different, meaning do they have different predictors?**

Reopened defects are in essence a subset of defects, and so it would be useful to see whether they share the same predictors and occur in the same modules.

It is important to keep in mind that in order to keep the study focused, architectural factors, primarily those used in previous studies in similar contexts, will be the main focus, and hence process and project-related factors will not be the main focus of the analysis and discussion.
1.4 Organization of the remainder of the document

The document proceeds by providing a brief overview of the literature on software quality and software complexity in chapter 2. Therein, software metrics are discussed and a subset of pertinent metrics is elaborated on. Chapter 3 goes over the dataset used for this study, providing details of the data extraction, transformation and loading processes. Chapter 4 shares the statistical models that were developed and goes through the resulting regression equation used to explain defects, starting with the results of analyses done on defects in general and then presenting the results of analyses done on reopened defects. Finally, chapter 5 discusses the practical implications of the results and makes note of areas worth investigating in the future.
Chapter 2

Software quality and complexity

2.1 Defects in software

Defects emerge in engineering systems as a result of mismatches between requirements and implementation. A requirements document serves as a contract for the system and the yardstick by which its functionality, that is to say, its functional requirements and quality attributes, is gauged. Any behavior, or use-case, that breaks said contract constitutes a defect. Defects are unavoidable in man-made artifacts precisely because they are artifacts. Therefore, the question isn’t how to avoid them, but rather how to mitigate and manage them.

A fundamental concept in managing defects is that of phase containment (McConnell 1998), which refers to the practice of fixing defects in the same phase that they are detected in. Fixing an anticipated defect in the requirements phase is much cheaper, possibly even free, if done during the same phase, as there are fewer cascading changes that would need to be done as a result of that fix. Fixing that same defect in production would be much more costly, as it involves tinkering with greater complexity, with a live system, with active users and customers and with a longer organizational and technical chain of change. Employee morale may take a hit as may user confidence. The cost of fixing a defect is a function of all of these factors and more.

A recently coined term that captures an interesting phenomenon in defect buildup is technical debt. The idea is that as the software development process progresses, decisions are made and every time a short-cut of some kind is taken, one accumulates
debt that then has to be paid off somewhere down the line, in the form of time and effort. A colloquial term used in software engineering to describe such changes is kluges.

Kluges are quick, ‘Band-Aid’ fixes that are applied without considering their implications on the rest of the system or on future changes; as these kluges increase in number, they eventually lead to enough frustration that they result in a refactoring of the codebase. Refactoring is another way of saying that the system is reevaluated from an architectural standpoint and significant portions of it are rewritten or moved around.

An important component of software engineering quality assurance (QA) practices is regression testing. Regression testing is based on the idea that every time one makes a change to a system in the form of a fix or otherwise, one needs to ensure that the change does not break existing functionality. To achieve regression testing, organizations typically build up a suite of test cases that achieve as high a coverage of the system’s use-cases as possible\(^1\). Running the suite of test cases at any given point of time can be a good indication of the system’s health. Regression testing may be done manually or it may be done using automated tools. Or in fact, it may not be done at all.

Quality assurance, though, is a function of more than one activity, and as previous work has shown, combining these activities is the key to getting the most out of them (McConnell 2004). In his work on software project management, McConnell reports that an investigation into the defect-detection rate of a set of defect-detection techniques revealed that no technique on its own can achieve more than a modal rate of 75%. Regression testing, incidentally, has a modal rate of just 25%. Other studies have shown

---

\(^1\) For the unaware reader, a test case is a sequence of steps designed to check the pre-conditions, post-conditions and invariants of a set of functions. If they all hold, then the functionality is deemed fine; otherwise, it is deemed faulty. An example invariant, stated as prose, may assert the following: “The withdrawn amount can never be more than the available balance”. If a call is ever made to the `withdrawCash` function and the invariant is broken, the function or component’s code would need to be investigated.
that informal techniques, such as code reviews, are the most effective way of finding defects (Myers, The Art of Software Testing 1979).

2.2 Tracking defects

Defect tracking is a critical part of a project’s QA process as it not only allows for a way to manage defects, but it also allows for defect activity data to be collected, allowing them to then be used as a means for improving the product and process. With defect tracking software, one can create a report for each defect and record an array of attributes about it, such as its status, reporter, assignee, description, priority, severity and history. Various defect tracking systems exist, with Bugzilla being one of the more widely used ones for open-source projects.

Among the data typically stored for a defect in a defect tracking system is where in its workflow it is, as well as data about all of its previous states. During its lifetime, a defect may go through the states shown in diagram 1. A more elaborate diagram and one that more accurately reflects reality in Bugzilla may be found in the Bugzilla Guide (Bugzilla 2009).

![Diagram 1: An abstract representation of a defect's lifecycle](image)

Other possible resolutions: 
{invalid, wontfix, worksforme, duplicate, obsolete)
In GNOME’s Bugzilla, a defect starts as a defect report created by some user and gets assigned the *unconfirmed* status. The process of triaging then determines whether or not the defect is an actual defect and if so moves its state to *new*. Invalid or duplicate defects are removed during triaging and marked accordingly by taking them to the *resolved* state and setting their resolution field to one of *invalid*, *wontfix*, *duplicate*, *obsolete*. A resolved defect can hence have one of several resolutions. Defects that are reported in older versions of a system that are no longer supported are taken to the *resolved obsolete* state and those that that can’t be reproduced by the developer are taken to the *resolved worksforme* state. Discussions may ensue in the comments section of the defect report if disagreements arise between contributors. A triaging manual serves as a set of guidelines for users (GNOME). Once a developer takes possession of the defect, it moves to *assigned*.

Once an assigned defect is fixed, a patch may be attached to the defect report and once it is committed and confirmed to be working, the defect is taken to the *resolved fixed* state. In some cases, a defect report may be reopened either because the defect is noticed again or because of contentions between developers, in which case the defect report is revisited and its status is moved from *resolved fixed* to *reopened*. If a new defect report is created, good triaging ought to catch it as a duplicate, close it as invalid and reopen the original defect report. Triaging sets apart a useful defect-tracking system from a disorganized, haphazard one and is a function of the existence of as well as the effectiveness of the project’s bugmasters. Former GNOME bugmaster, Luis Villa, defines the bugmaster as someone who “voraciously reads as many bugs as possible, and attempts to help the organization deal with those bugs by constantly improving the quality of the bug information hackers/developers, leaders/managers, and QA/volunteers have” (Villa).

### 2.3 Measuring software quality

A software system is made up of a number of facets, each of which can be traced to a set of specifications, which collectively make up the system’s requirements specification.
According to Garlan et al. and others, they are: functional requirements, quality attributes and constraints (Garlan and Shaw 1996). The second is of particular importance since it captures the so-called “-ilities” that prove to have a far greater importance in the success or failure of a system than its functional requirements. McConnell includes a subset of these “-ilities” in his definition of what constitutes software quality (McConnell 2004).

Quality attributes are usually cross-cutting and require quantification of some sort in order to be measurable. This quantification may be in the form of precise numbers, such as mean-time-to-failure for the quality attribute of ‘availability’, for example, or it may be in terms of orders of magnitude, such as in the case of performance, for example. For the sake of analysis, it is essential for measures of software quality to be precise.

Kan categorizes software quality metrics into three main categories: product metrics, process metrics and project metrics (Kan 1995), with each of the three defined as follows:

- **Product metrics**: metrics that describe characteristics of the end-product such as size and functional and quality attributes.
- **Process metrics**: metrics that can be used to improve the software development process, such as time-to-fix and defect removal effectiveness.
- **Project metrics**: metrics that describe characteristics of the people that develop the product and the environment that they are in, such as the number of engineers, schedule and cost.

These appear to serve as a good taxonomy for quality assurance metrics as they cover the three main dimensions within which software development activities occur. The following sections present one product metric, defect density, and two process metrics, defective fixes and time-to-fix.

### 2.3.1 Defect density

The defect-density metric is a measure of the ratio of defects to some measure of size for a particular system. It has as its numerator the absolute number of defects in the system.
during the time period being measured, which may be the lifetime of a particular version or the lifetime of the system since inception, and as its denominator lines of code. Lines-of-code, of course, is not the only measure of system size, but it is the measure most commonly used.

\[
\text{Defect density} = \frac{\text{total confirmed defects}}{\text{total lines of code}} \times 100
\]

The definition of what constitutes lines-of-code is important to keep in mind since it can be ambiguous, i.e. does it only measure executable lines, or executable lines plus definitions, or executable lines plus definitions plus comments? There is no universal definition of what lines-of-code ought to measure nor is there a consensus between authors and practitioners (Boehm 1981, Conte, Dunsmore and Shen 1986), which is why consistency within the same study is important and being aware of it when comparing results from different studies is critical. Furthermore, it is advisable to use the same tool for counting lines as tools may vary in the way they define the measure. Kan points out that IBM, for example, have developed their own tools for counting this and other measures that they use across all divisions (Kan 1995).

The defect-density metric may at times result in a fraction that is too small to be convenient during analysis, particularly when working with large codebases, so using thousands-of-lines-of-code (KLOC) for the denominator is typically used in such cases. Given that it controls for system size, defect density is a useful metric to use to compare different releases of the same product or different products.

2.3.2 Defective fixes

The defective-fixes metric is a measure of how well a fix sticks once applied, that is to say, it captures the defects that have been deemed resolved and fixed only to then be reopened at some point in the future (Kan 1995). Kan suggests using a straight count of
defective fixes rather than a percentage, citing arguments against the latter, which make the point that percentages may at times give a falsely optimistic picture. One may wish to control for system size, in which the measure becomes:

\[
\text{Defective fixes} = \frac{\text{total reopened defects}}{\text{lines of code}}
\]

Given the existence of good triaging standards and a good quality assurance team, the measure allows one to see how many defects are reopened and then through analysis reason about why it may be that their fixes fail to stick. The point about triaging is critical since without that, reopened defects may not get marked as duplicates and may then get counted as new ones. The end goal, of course, is to target having zero defective fixes.

2.3.3 Time-to-fix

The time it takes to fix a defect can be critical for defects that impact an important set of use-cases, those that prove to be show-stoppers for an upcoming release or for those that are not severe in terms of business functions, but are so annoying that they impact the user experience. The measure for time-to-fix is usually the mean time it takes to take a defect report from new to resolved (or open to closed) for all defects in the release or product. The median may be used if the set includes extreme values (Kan 1995). It may also be useful to split defects by severity when calculating time-to-fix and observing potential variations in the measure for different types of defect and the respective impact of each on users satisfaction.

2.4 Measuring system complexity

Defining complexity in any domain can be challenging given its ambiguity; in software it is perhaps more challenging given the artificial, non-tangible nature of software artifacts. Furthermore, and as Fred Brooks points out, “descriptions of a software entity that abstract away its complexity often abstract away its essence.” (Brooks 1987). Verily, the
discipline can be so haphazard that the net that one would have to cast to capture all the characteristics of complexity proves to be much larger and may cross into non-technical domains such as management.

It is useful to include measures of complexity in any analysis of architecture and quality as they have been shown to have predictive power. The proceeding sections present a subset of these metrics. The technical measures discussed herein look at complexity using two lenses with varying focal lengths. If a system is envisioned as a graph of nodes and edges, the first lens zooms into each of the nodes, extracts its internal structure and consolidates it with that of every other node; the sum then constitutes the system. Conversely, the other lens is shorter, and hence focuses more on the edges that connect the nodes together, reasoning about their cardinality, patterns and lengths.

2.4.1 Lines of code

Lines-of-code is the simplest measure of a system’s complexity. Intuitively perhaps, one may be able to see that the more lines of code there are in a system, i.e. the larger it is in size, the more likely it is for it to have defects. Though other measures of size, such as file counts, may also be used, lines-of-code is typically the more widely used measure of size. Despite the emergence of more clever metrics, some practitioners argue that lines-of-code remains one of the best measures of complexity (Hatton 2008, Shore 2008).

Studies have shown that the measure tends to have an inverse relationship with defect density, that is to say, a larger system size has a smaller defect density, though other studies cited by Kan, for example, show it to be curvilinear relationship: defect density decreases linearly as system size increases only to then curve up at the tail (Kan 1995, Withrow 1990). The dataset used for this study shows a similar relationship (see chart 1), with the apparent optimal system size lying at around 380,000 lines of code.
2.4.2 Cyclomatic complexity

Cyclomatic complexity, developed by Thomas McCabe in 1976, measures the number of linearly independent paths within a software system and can be applied either to the entire system or to a particular class or function (McCabe 1976). By viewing a block of code as a control graph, the nodes constitute indivisible lines of code that execute in sequence and the directed edges connect two nodes if one can occur after the other. So, for example, branching constructs like if-else statements would result in a node being connected to two output nodes, one for each branch. The formula for the measure is defined as:

\[ v(G) = e - n + 2p \]

Where \( v(G) \) is the cyclomatic number of a graph \( G \), \( e \) is the number of edges, \( n \) is the number of nodes and \( p \) is the number of connected components, or exit nodes, in the graph. Visually, a block of code with a single if-else statement after the start of execution in it would look like the graph in diagram 2. Herein, \( e = 6 \), \( n = 6 \) and \( p = 1 \), so the cyclomatic complexity is then \( 6 - 6 + 2 \times 1 = 2 \).
The additive nature of the metric means that the complexity of several graphs is equal to the sum of each graph.

In terms of statistical power, cyclomatic complexity has been shown in countless studies to be a positive predictor of defects and has therefore been widely used in industry for that purpose. Some recent studies, however, have concluded that the measure has only as much statistical power as lines-of-code (Hatton 2008).

Furthermore, when controlling for system size, Kan and others report inconclusive results with some studies showing that it remains a significant positive predictor, others showing that it becomes a significant negative predictor and others still showing that it loses all significance (Kan 1995). In this study, cyclomatic complexity controlled for system size appears to have no significance, neither in terms of correlation nor as a predictor, with respect to defect density, though it does appear to show significant positive correlation with reopened defects.

Diagram 2: A control graph with a cyclomatic complexity of two
2.4.3 Design structure matrices and fan-in/fan-out metrics

The Design Structure Matrix (DSM) is an engineering tool first proposed by Don Steward that can be used to model complex systems within the compact form of a square matrix (Steward 1981). Research methods that apply DSMs to software have achieved interesting results, particularly by using it to visualize and hence analyze quality attributes such as modularity (Sosa, Mihm and Browning 2009). Some firms have started incorporating DSMs into their development environments to encourage their use for dependency analysis in practice\(^2\) (JetBrains).

In a DSM, one lists the components of the system on both axes in some order and then within the matrix, wherever there happens to be a dependency between two components, one would mark that point with a dot. The end result is a matrix that communicates the state of a system’s interconnected components, or, put differently, its dependencies. The elements across the axes need not be components; they may be tasks, for example, in which case a DSM may be used as a tool for project management (Eppinger, et al. 1994). Various studies have shown the applicability of DSMs to domains including aerospace, automotive, manufacturing and telecommunications (MacCormack, Rusnak and Baldwin 2006).

Diagram 3 below shows a simple DSM where dependencies are marked with a ‘•’. The call graph to its right is the hypothetical system that it models.

\(^2\) For the interested reader, an interactive, experimental, visualization of GTK+ v2.28’s codebase may be found online (Almossawi 2011).
A DSM is read starting with an element in the vertical axis, so for example, one may say “Component C calls component B” or “Component A calls components B, C, D and E”. Notice that some components only have outward dependencies, such as component A, while others only have inward dependencies, such as component E, while others still have both inward and outward dependencies, such as components G and D. Note that dependencies captured in a DSM need not be binary, though in this study we define them to be binary to simplify the analysis.

A DSM of a particular system can then be used to reason about its modularity, which in software as well as other engineering disciplines is usually a desirable quality attribute\(^3\). In software, a DSM may have source-code files as elements and various forms of interactions between source-code files, such as function calls, casts, global variables, etc. as dependencies.

\(^3\) Quality attributes are about trade-offs; more modularity may result in better maintainability, modifiability and scalability, though it may also result in lower performance. It is the software architect’s job to determine the right balance of quality attributes given the system’s requirements and constraints.
Other studies point out that it is not only the number of dependencies between elements that proves an important indicator of modularity, but also the pattern of distribution of said dependencies (MacCormack, Rusnak and Baldwin 2006). The order in which elements are placed on the axes can impact the patterns and groupings that emerge. There is a body of work that discusses various partitioning methods for DSMs (Steward 1981), though in this study elements in DSMs are ordered by module, alphabetically, which results in the source-code files of the same module lying contiguously along the DSM’s axes. The term module here refers to directory structure.

Beside the visual aspect of a DSM, analysis may be done on it using tools such as MATLAB to elicit information such as dependency density, which is calculated by dividing the number of non-zero elements by the total number of elements. In this study, this metric is called first-order density, and is defined as:

\[
\text{First-order density} = \frac{\sum \text{non-zero matrix elements}}{n \cdot n}
\]

Where \( n \) is the number of rows or columns in the matrix. A system with fewer dependencies between its components has a DSM that is sparser and hence has a lower dependency density count.

Another way of calculating first-order density is to consider its numerator a sum of all rows in the matrix or a sum of all columns. The rows and columns, as it turns out, capture distinct characteristic of the system, with the rows giving an indication of the number of files that have incoming calls from other others, referred to in literature as fan-in, and the columns giving an indication of the number of files making outward calls to other files, referred to as fan-out (Warfield 1973, Yourdon and Constantine 1979). Hence, we get two metrics called fan-in and fan-out that are defined as:
• Fan-in: the number of elements that call a particular element
• Fan-out: the number of elements that this particular element calls

The two metrics allow one to see not only the dependencies between elements, but also their direction. Given a DSM, fan-in may be calculated by summing each row and dividing by the total number of elements. That would give the fan-in for each element, which is to say, a count of the number of elements that call each element. To obtain fan-out, one would sum across the columns and divide by the total number of elements. So for example, given the DSM shown in diagram 3, element A would have a fan-in of 4/9, or 44%, meaning that roughly half the elements in the system depend on it. Element a’s fan-out is 0/9, or 0%, meaning that there aren’t any elements that it depends on.

2.4.4 Visibility matrices and propagation cost

The simple DSM, as described thus far, only captures direct dependencies between elements. What if one wanted to get a more vivid view of a system’s interdependencies, i.e. capture not only direct dependencies but also indirect dependencies? Existing literature refers to this type of DSM as a visibility matrix, which is an idea based on the concept of reachability matrices (Warfield 1973), and the argument is made by some that they provide true visibility into the entire chain of dependencies that are interwoven between components in a system. The process of transforming a first-order dependency matrix that captures only direct dependencies to a visibility matrix that captures indirect dependencies as well can be achieved through the mathematical process of matrix multiplication.

To obtain a visibility matrix, a matrix is raised to successive powers of n, where n represents path lengths, which put differently, means that if the matrix is raised to the power of two, it would then show the indirect dependencies between elements that have a

---

4 Note that in visibility matrices, discussed in the proceeding section, we always start with a matrix that assumes that each component depends on itself, hence we never end up with a fan-in or fan-out of 0.
path length of two, i.e. calls from A to C, if A calls B and B calls C. Thereafter, by summing these matrices together one gets the visibility matrix, V, which shows the dependencies that exist for all possible path lengths up to n. Diagram 4 illustrates the process of raising a matrix to the power of four and then summing all matrices to get the visibility matrix for the hypothetical system shown in graph-form; an example using GTK+ is shown in appendix B. The value of \( n \) that a system’s matrix is raised to in order to obtain the visibility matrix is the number of elements minus 1; hence it is 5 for the example shown below. For this ripple effect to be useful for analysis, it is captured within a metric called propagation cost (MacCormack, Rusnak and Baldwin 2006).

Diagram 4: The process of matrix multiplication by which a visibility matrix is obtained (based on the example given in MacCormack et al. 2006)
Given a visibility matrix, one may similarly calculate visibility fan-in and visibility fan-out, which would give indications of the total number of direct and indirect elements that an element calls or that call an element. With visibility matrices, note that we start with a matrix that assumes that each element depends on itself, hence the existence of a ‘1’ throughout the diagonal. The average visibility fan-in or visibility fan-out constitute the measure known as propagation cost. Since there is symmetry between the two measures, i.e. for every fan-in there is a corresponding fan-out, only one of those is needed to calculate propagation cost. So taking the visibility matrix shown in diagram 4 as an example, propagation cost calculated using, say fan-in, becomes:

\[
\text{Propagation cost} = \frac{\sum \text{fan-in}}{n \cdot n} = \frac{1 + 2 + 2 + 3 + 3 + 4}{6 \cdot 6} = 42\%
\]

Propagation cost is hence a weighted average of all dependencies in a codebase, both direct and indirect, and thus serves as a measure of what may be deemed the true amount of coupling in a system and the more prominent driver of system structure (MacCormack, Rusnak and Baldwin 2008): a higher propagation cost indicates greater coupling and a possibly more monolithic system.

Visibility can give a different view of a system compared to direct dependencies. Take Anjuta, for example, which is a development environment for C/C++ and other languages. Diagram 5 shows two DSMs: the first-order matrix and the visibility matrix. The visibility matrix is denser, as one would expect, though notice that the patterns that emerge are somewhat different. For example, although the square block in the center and the ones in either corner remain visible in both matrices (these are the system’s cores; more on them later), the vertical and horizontal “buses”, which are hardly visible in the first-order matrix, are much more obvious in the visibility matrix. These buses allow one to realize that when indirect dependencies are considered, there are files in this system that only call out or only call in.
Notice now how the visibility matrix becomes much sparser in version 2.12 (diagram 6), suggesting a refactoring effort that resulted in a more modular structure and a decreased number of indirect dependencies.
Propagation cost may also be expressed by breaking it apart into four distinct categories, as shown in diagram 7, the intuition being that different directions and magnitudes of dependencies have varying impacts on software quality, as has been shown in previous work (MacCormack, Rusnak and Baldwin 2006). For example, a smaller core has been shown to result in fewer defects and hence a more stable system. The four categories are obtained by considering the different combinations of fan-in and fan-out values:

- **Core components**: files that depend on a lot of files and have a lot of files depend on them\(^5\). These have high fan-in and high fan-out.
- **Peripheral components**: files that don’t depend on a lot of files and don’t have a lot of files depend on them. These have low fan-in and low fan-out.
- **Shared components**: files that don’t depend on a lot of files, but have a lot of files depend on them. These have high fan-in and low fan-out.
- **Control components**: files that depend on a lot of files, but don’t have a lot of files depend on them. These have low fan-in and high fan-out.

---

\(^5\) In the spirit of propagation cost being a measure of “visibility”, it may be useful to restate the description as “files that see a lot of files and are seen by a lot of files”, and similarly for the rest.
In this and some previous studies, the cutoff point between levels is defined as the median visibility fan-in and median visibility fan-out. Values that lie on the median are pushed away from the core.

2.5 Limitations of software metrics

Software metrics are a great tool for one to have in one’s armory because, by definition, they have this wonderful trait of encapsulating all the facets and details, and indeed, complexities, of a system in some value or set of values. What could be better for a manager than to look at a series of values representing a series of projects and use the former to reason about the latter. By that same token, how does one ensure that those metrics do indeed encapsulate all the idiosyncrasies of the projects? And how about qualitative attributes that may be difficult to codify?

Kearny et al. make the point that software metrics generally suffer from the problem of not being idiosyncratic, i.e. they do not consider the specifics of the program being analyzed, the programmer and the domain in general (Kearney, et al. 1986). They discuss findings of several researchers, some of which appear to be contradictory. For example, one work found that global variables led to fewer defects whereas others showed that modularity led to fewer defects. They also show how easy it is to misinterpret the statistics and end up with inaccurate conclusions. On the flip-side, software metrics are also not particularly domain-agnostic, hence they may be criticized on both levels.

It may be helpful to view software metrics as just another abstraction method. A former lecturer once mentioned that abstraction is hiding detail and that good abstraction is hiding the right detail (Shaw). The same standard may be applied to software metrics by asking the question: are the system characteristics that the metric abstracts away the right characteristics, or not? And more importantly, is the metric useful within the context that it is used, or not, regardless of its universality?
Chapter 3

Method

3.1 The dataset

Given our interest in gauging the effectiveness of our architectural measures in predicting defects at the system level, we chose to collect data from the GNOME open-source community (GNOME). GNOME is a desktop environment that is run predominantly on top of Linux and is made up of over a hundred individual projects that reside within a federated system. The projects vary in size from tens of thousands of lines of code to over half a million lines and cover a period of approximately eight years of activity. Projects are run semi-autonomously by teams of contributors with varying levels of commitment, management and documentation styles and technical convictions. Though projects are written in a number of programming languages, most are in C/C++.

Most of the projects that comprise GNOME make use of the Bugzilla defect-tracking system (GNOME Bugzilla), which makes defect activity data readily available. The application allows a multitude of data combinations to be generated and also makes retrieving and transforming data programmatically a straightforward task.

One of the challenges of working with open-source data is the potential for inconsistencies. In GNOME, for example, the severity and priority fields, to know just two, are used very differently by different projects, with some ignoring them all together (Klapper 2012). The challenge is magnified when one works with a cross-sectional dataset, as is the case in this work. To demonstrate the potential haphazardness of some of the underlying data, here is an extract from the comments section of a bug report:
Of course, it is for that reason that mature open-source projects like GNOME have triaging teams and workflows in place. In GNOME, the Bugsquad is an overarching team of volunteers that helps maintain quality in most GNOME projects.

3.2 Choosing the systems

The GNOME project-listing page lists 168 projects that have been developed for GNOME. Henceforth, these shall be called systems⁶; a release of a particular system shall be called a system-version. As part of previous work done with GNOME (Mueller-Birn, et al. 2010), the source-code of 90 GNOME systems and their respective releases, chosen at random and which have data for the time period being considered, were retrieved. We made use of that set of systems as our starting point. Conducting a first-pass through the systems resulted in around two thirds of them being eliminated and 32 being deemed suitable for further analysis. Our criteria for selection were:

- A system should be made up of at least 100 source-code files, on average
- A system must have at least 10 releases
- A system must be written in C/C++

The first criterion ensures that first-order density and propagation cost are not impacted by small system sizes. Previous work has shown that these variables may become unreliable and less meaningful when the system size is too small, which makes intuitive sense. Also, intuitively, it is not very meaningful to talk about dependencies when the system size is small enough for one person to be knowledgeable about all of its modules.

⁶ Previous work looked into propagation cost as a measure of dependencies between modules of the same system; here, a similar exercise is done, but at the system level.
Given the cross-sectional nature of our dataset, the second criterion ensures that we don’t bias our analysis with systems that have too few observations. Hence, we set a minimum threshold of 10 observations per system.

Though our static analysis tool can work with a large set of programming languages, we choose to focus only on systems written in C/C++ to remove any disparities that may arise in our analysis as a result of comparing systems written in different programming languages. Filtering out potential extreme points for our regression model’s dependent variable, defects per kloc, further reduces the number of systems in our sample from 499 observations to 494. We end up with a sample of 32 systems and 494 observations.

Our dataset includes systems from version 2.0 of GNOME to version 2.32, though not all systems had a release for each of the official GNOME releases during that period. We identify systems’ releases using the GNOME version numbers even if the system has its own versioning protocol. Versions 1.0, 1.2, and 1.4 are not included in our analysis primarily because there are no defect data for those versions in the defect tracker. Furthermore, it made sense to start at version 2.0 for a number of reasons: much of the project was rewritten from scratch in version 2.0 (Loli 2008), the project suffered from relative instability prior to the version 2.0 rewrite, as is evident from our analysis of defect activity during that period, and because of the project’s switch to 6-month release cycles from version 2.0 onwards (GNOME Live 2008). For the sake of comparability, it makes sense to consider system-versions with the same lifespan.

Although a lot of defects were fixed prior to 2.0, looking at the growth patterns of the system-versions from their first release post-2.0 to their last one pre-2.32 reveals that 76% of them have in fact grown. Chart 2 shows the distribution for all system-versions (Min = -39%, Max = 1,525%, Median = 42%). Hence, it is evident that the systems were being actively being worked on and reworked and non-trivial defects were being reported and fixed and therefore our dataset does capture a lot of new development.
3.3 Variables of interest

Given that the research questions are concerned with what factors may be deemed good predictors of defects, we choose to focus on a set of architectural variables, namely, size, in terms of lines-of-code and files, complexity, in terms of measures of coupling and cyclomatic complexity and the amount of change in a system’s size between releases. The measures of coupling make use of variables that have been shown in prior work to be promising indicators of defects, namely, direct dependencies, propagation cost and the size of a system’s core, periphery, shared and control components (MacCormack, Rusnak and Baldwin 2006, MacCormack, Baldwin and Rusnak 2010). Bug-report attributes such as priority and severity are readily available, but we choose not to include them because of the inconsistency with which they are used throughout GNOME.
3.4 Descriptive statistics

Table 1 summarizes the descriptive statistics for our sample. Note that the system-
versions capture defect activity over a period of six months.

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Mode</th>
<th>SD</th>
<th>Min</th>
<th>Max</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>defects per kloc</td>
<td>.8393</td>
<td>.6332</td>
<td>0</td>
<td>.8606</td>
<td>0</td>
<td>5.275</td>
<td></td>
</tr>
<tr>
<td>defects reopened per kloc</td>
<td>.0155</td>
<td>0</td>
<td>0</td>
<td>.0283</td>
<td>0</td>
<td>.2285</td>
<td></td>
</tr>
<tr>
<td>first-order density</td>
<td>.0306</td>
<td>.0260</td>
<td>0</td>
<td>.0224</td>
<td>.0040</td>
<td>.1426</td>
<td></td>
</tr>
<tr>
<td>propagation cost</td>
<td>.2723</td>
<td>.2700</td>
<td>.2890</td>
<td>.1592</td>
<td>.0280</td>
<td>.7610</td>
<td></td>
</tr>
<tr>
<td>percent change in kloc</td>
<td>.0690</td>
<td>.0151</td>
<td>.2493</td>
<td>-.5872</td>
<td>3.288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kloc</td>
<td>128.53</td>
<td>76.90</td>
<td>2.368</td>
<td>134.98</td>
<td>11.740</td>
<td>731.71</td>
<td>63,494.54</td>
</tr>
<tr>
<td>kloc code(^7)</td>
<td>8.33</td>
<td>47.68</td>
<td>11.80</td>
<td>88.44</td>
<td>7.33</td>
<td>478.04</td>
<td>39,681.50</td>
</tr>
<tr>
<td>files</td>
<td>307.46</td>
<td>203.00</td>
<td>122</td>
<td>285.94</td>
<td>22</td>
<td>1527</td>
<td>151,883</td>
</tr>
<tr>
<td>core size</td>
<td>.0824</td>
<td>.0538</td>
<td>0</td>
<td>.0822</td>
<td>0</td>
<td>.3297</td>
<td></td>
</tr>
<tr>
<td>periphery size</td>
<td>.3074</td>
<td>.2518</td>
<td>.4344</td>
<td>.1805</td>
<td>.0755</td>
<td>.9268</td>
<td></td>
</tr>
<tr>
<td>shared size</td>
<td>.3082</td>
<td>.3134</td>
<td>.4262</td>
<td>.1160</td>
<td>.0052</td>
<td>.5000</td>
<td></td>
</tr>
<tr>
<td>control size</td>
<td>.3020</td>
<td>.3122</td>
<td>.1393(^a)</td>
<td>.1129</td>
<td>0</td>
<td>.5000</td>
<td></td>
</tr>
<tr>
<td>mccabe sum</td>
<td>11867</td>
<td>6671</td>
<td>2167</td>
<td>13280</td>
<td>1014</td>
<td>63846</td>
<td></td>
</tr>
<tr>
<td>mccabe per kloc</td>
<td>87.32</td>
<td>2167</td>
<td>73.73</td>
<td>2.52</td>
<td>32.38</td>
<td>139.72</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Descriptive statistics for the dataset (n=494)

Systems’ mean size is around 131,000 lines of code, which makes them a good size for analysis. Defects densities range from .01 defects per KLOC to around 5.275. Looking at defect densities less than .01 within the context of a system’s other releases indicates that they are most likely due to defects not being recorded in Bugzilla rather than them not

---

\(^7\) The variable kloc code captures just the executable lines of code and excludes all other lines of code, such as comments.
existing at all in a release. In any case, the sample provides a good amount of data to work with\(^8\).

Propagation costs range from around 3% to around 76%. Given the nature of the dataset, as described below, it could be the propagation cost metric is being influenced to a great extent by the large number of shared calls that the systems in our sample seem to be characterized by. The most noticeable observation from table 1 is that most systems have a very small core. See chart 3 for a comparison of core and control components: 30% of system-versions, have a tiny core; compare this to the near-uniform distribution of control component sizes. Kindly refer to appendix D for a more elaborate table showing the core, periphery, shared and control component sizes for a subset of the system-versions.

The percent change in kloc variable is the percent increase or decrease in the size of a system’s codebase from one release to the next. The same variable is used to capture increases as well as decreases and so negative values are possible. The mean percentage change in kloc is around 7% and the median is just 1.5%, so as a percentage of their total sizes, systems typically change from one release to the next.

\(^8\) Looking at the raw defect counts, that is, without controlling for system size, the mean defects per system-version is around 99 and the median is 58.
3.5 Collecting and processing the data

Processing the data involves the use of a set of tools and scripts. The general workflow is shown in diagram 8 and elaborated on in later sections. Our tool set is comprised of a static analysis tool called Understand (Scitools), a statistical analysis tool – SPSS (IBM), a numerical computing environment – MATLAB (MathWorks) and a number of local and web scripts. The input to the static analysis tool is a particular system-version’s codebase. The end-result is a set of metrics that are consolidated into a single dataset and analyzed by our statistical analysis tool.
### 3.6 Source-code data

Chart 4 shows the file activity for the ten largest projects, based on mean file size. Possible refactoring efforts are noticeable. Please refer to appendix A for details of the data processing steps. Evolution appears to be by far the biggest system, and, sees a steady decline in size until version 2.10 when it plateaus. It was during this period that Evolution saw its back-end split off into the Evolution Data Server (Klapper 2012).
Glom also sees a sharp drop of around 37% in size between versions 2.12 and 214. Other systems appear to be steady throughout. Others still appear to noticeably grow, such as Banshee, F-Spot, Anjuta and GTK+.

3.7 Defect activity data

GNOME’s bug-tracking software makes it easy to programmatically retrieve defect activity data. The URLs we use to retrieve defects from Bugzilla specify the following conditions:
1. Status ≠ “unconfirmed”
2. Resolution = “fixed” or resolution = “---”
3. Severity ≠ “enhancement”
4. Product = [the name of the GNOME project]
5. GNOME version = [the version number of the GNOME project]

One of the wrinkles with GNOME’s defect activity data is that defects aren’t recorded in the same way in all systems, nor are they always mapped to the versions in which they were discovered. We take the former into consideration in our regression analysis. For the latter, we accommodate that group of defects by defining unassigned defects as follows:

An unassigned defect \( d \) is caused by version \( v \) of a system if it was created within time period \( t \) to \( t' \), where \( t \) is the release date of \( v \) and \( t' \) is the release date of the next version. Per the conditions used in the search query shown above, the unassigned defect \( d \) must not have a status of unconfirmed or a resolution value other than fixed.

Though defects created by users within a particular time period need not necessarily map to the previous release, the assumption is that the second criterion would filter out most of those cases, i.e. a defect discovered by a user in an older version will likely have been resolved in the latest version. As a robustness check, we used this method to calculate assigned defects for a set of system-versions and got results that were very close to the actual number of defects reported by Bugzilla for those system-versions. The average overlap is 80% and the mode is 95%.

The sketch in diagram 9 below shows an abstract representation of how defects are mapped to releases:

---

9 The resolution field can be either ‘fixed’ or ‘---’, which accommodates both sets of closed and fixed defects as well as confirmed open ones.
Unassigned defects are not statistically different than assigned ones, i.e. they don’t show any difference in the type of defects that they capture. Chart 5 shows defect activity for five of the largest projects, per their mean file size, as before, and chart 6 shows the reopened defect activity for the same projects. Most systems exhibit a decline in confirmed defects, as in fact do the ones shown in said charts. Evolution sees the sharpest drop in defect activity. A few peaks are apparent around version 2.16 for example. For some projects, defects appear to steadily increase and then suddenly go down in between two successive releases, as can be seen for Anjuta and Banshee, for example. Reopened defects appear to be moving with defects in general, though there are some cases where defects drop, but reopened defects go up, such as the case for GTK+ during the first few releases shown in the chart, which may possibly be a result of the complete rewrite of the toolkit in version 2.0 and the impact of that major change on developers’ familiarity with the new code.
Chart 5: Defect activity for five of the largest systems (absolute)

Chart 6: Reopened defect activity for five of the largest systems (absolute)
For reopened defects, the same method is used with the added condition that the defect’s status be changed to *reopened* at some point in the future. Due to the fact that defects may not necessarily have been reopened from a *resolved fixed* state, the previous status is checked and only those that indicate that the defect was indeed fixed are included in the count. In all cases, the comments section was perused as an added robustness check; this was in response to a discussion with Andre Klapper, of the GNOME Bugsquad, who shared the following about Evolution’s history (Klapper 2012):

“...we had problems to handle the massive amount of reports in late 2006, as many reports were about crashers and missed enough information (debug symbols) in the stack-traces, so the GNOME Bugsquad decided to not set such non-useful reports to ‘needinfo’ status, but set it directly to ‘resolved incomplete’...So for reopened reports that previously were resolved incomplete before, it just means that the reporter did come back to provide more information...”
3.8 Removing singletons

We define singletons as source-code files that do not have inward or outward dependencies. Analysis of our set of system-versions reveals that the number of singletons is on average very small. Nevertheless, we take them out for our statistical models. For only around 7% of system-versions do singletons make up more than 10% of the codebase. The mean is 3.9% and the median is 2.1%. Those with the highest percentage of singletons are, predictably, libraries such Libxml2, LibSoup and GMime. Chart 7 shows a histogram of the percentage of singletons in a system-version for all system-versions in our dataset.

![Histogram of Percentage of Singletons](chart7.png)

*Chart 7: Percentage of files that are singletons*
Chapter 4

Results

As part of our analysis, we use standard multiple regression to assess the ability of a set of measures to predict defects. We first analyze defects in general and show a set of statistical models that predict defects from a set of architectural variables. We then take a look at reopened defects and find that doing the same type of multiple regression analysis is difficult given that we lose statistical power due to the small number reopened defects. We therefore perform logistic regression analysis with good results and then follow that with qualitative data from interviews and peruses of the defect reports in the GNOME defect-tracking system. In the end, we compare the results of both types of defects.

4.1 Preliminary analysis

Preliminary analysis is conducted to ensure that there is no violation of the assumptions of normality, linearity and multicollinearity. Inspecting the histograms and Normal P-P plots revealed a slight to moderate positive skewness in a subset of the variables. Log-transforming them resolved the skewness, but had little impact on our models, hence the raw data values are used for that. The P-P plot for the dependent variable defects per kloc is shown in chart 8.

All of the independent variables have acceptable levels of correlation between them, that is, less than 0.7 (Pallant 2011), and there appears to be no correlation between any of them and the model’s residuals, satisfying the assumptions of Ordinary Least Squares analysis (OLS) for the independence of the residuals.
Doing a correlation analysis of propagation cost against the residual of the model where first-order density explains propagation cost reveals that there is a significant correlation between the two, thus breaking a fundamental assumption of OLS\(^{10}\). It is for that reason that we take out propagation cost in our final model (model IV). Note that propagation cost and first-order density remain positively correlated, as has been the case in prior work: \(r=.335\), \(n=494\), \(p < .01\) (using Pearson product-moment correlation coefficient).

The correlations of our variables to the dependent variable *defects per kloc* is shown in table 3. The complete correlation table is available in appendix C.

<table>
<thead>
<tr>
<th>defects per kloc</th>
<th>Pearson Correlation</th>
<th>prop. cost</th>
<th>first-order density</th>
<th>% in core</th>
<th>% in shared</th>
<th>% in periph.</th>
<th>% in control</th>
<th>% change in kloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop. cost</td>
<td>-.096**</td>
<td>.112(^*)</td>
<td>.108(^*)</td>
<td>-163**</td>
<td>-.023</td>
<td>.126**</td>
<td>.304**</td>
<td></td>
</tr>
</tbody>
</table>

\(^*\) Correlation is significant at the .05 level (2-tailed)

\(^\text{**}\) Correlation is significant at the .01 level (2-tailed)

Table 3: Correlation coefficients for defects per kloc (see appendix C for full table)

\(^{10}\) Propagation cost appears to be highly correlated (at the .01 level) with the residual of the model where first-order density explains propagation cost: -.397.
The removal of extreme points and potential outliers is as discussed in section 3. With respect to cyclomatic complexity, our dataset includes two measures of it:

- *mccabe sum*, which is a sum of all independent paths in a system-version
- *mccabe per kloc*, which control for the system-versions’ size

*Mccabe sum* is highly correlated to the variables *kloc, kloc code*, *files, defects* (.960, .98, .89, .65, respectively), however, *mccabe per kloc* appears to have no significant correlation with *defects per kloc* and a negative correlation with *defects per kloc code* (-.128). Furthermore, it appears to have a smaller correlation coefficient than *loc* (.647 vs .674) as shown in the correlation table in appendix C. The *mccabe per kloc* variable satisfies all the assumptions of normality, linearity and multicollinearity. Its histogram is shown in chart 9.

![Chart 9: Histogram for mccabe per kloc](image

### 4.2 Statistical models

Starting with defects, i.e. all types of defects, our statistical models are as shown in table 4. In model IV, we include the set of variables that we expect to be good explainers of defects. The model gives an adjusted R² value of 16.2% and shows all variables to be

---

11 Recall that *kloc code* captures just the executable lines of code and excludes all other lines of code, such as comments.
highly significant. *Percentage change in kloc* proves to be the greatest contributor causing an increase of about 9% in the $R^2$ value\(^\text{12}\).

<table>
<thead>
<tr>
<th></th>
<th>Model I</th>
<th>Model II</th>
<th>Model III</th>
<th>Model IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2$</td>
<td>.127</td>
<td>.146</td>
<td>.128</td>
<td>.176</td>
</tr>
<tr>
<td>Adj $R^2$</td>
<td>.119</td>
<td>.137</td>
<td>.119</td>
<td>.162</td>
</tr>
<tr>
<td>Constant</td>
<td>1.083***</td>
<td>.797***</td>
<td>1.111***</td>
<td>.685*</td>
</tr>
<tr>
<td>Files</td>
<td>.001***</td>
<td>.001***</td>
<td>.001***</td>
<td>.001**</td>
</tr>
<tr>
<td>KLOC</td>
<td>-.002***</td>
<td>-.003***</td>
<td>-.002***</td>
<td>-.001*</td>
</tr>
<tr>
<td>McCabe per KLOC</td>
<td>-.003†</td>
<td>-.004†</td>
<td>-.003</td>
<td>-.003</td>
</tr>
<tr>
<td>% change in KLOC</td>
<td>1.003***</td>
<td>1.020***</td>
<td>1.002***</td>
<td>1.001***</td>
</tr>
<tr>
<td>First-order density</td>
<td></td>
<td>6.779***</td>
<td></td>
<td>7.737***</td>
</tr>
<tr>
<td>Propagation cost</td>
<td></td>
<td></td>
<td>-.186</td>
<td></td>
</tr>
<tr>
<td>Core size</td>
<td></td>
<td></td>
<td></td>
<td>1.105*</td>
</tr>
<tr>
<td>Shared size</td>
<td></td>
<td></td>
<td></td>
<td>-.928**</td>
</tr>
<tr>
<td>Control size</td>
<td></td>
<td></td>
<td></td>
<td>.779*</td>
</tr>
</tbody>
</table>

*Table 4: Summary of the models developed*

*Dependent variable = defects per kloc, († p < .10, *p < .05, **p < .01, ***p < .001)*

As a robustness check, we take out Evolution, which stands out both in terms of size and defect activity; we still get similar models with similar $R^2$ values – in fact, we get a slightly higher adjusted $R^2$ of 16.9% for model IV. Taking out systems that appear to not show much growth between their first and last releases doesn’t impact our models either, resulting in similar coefficients and $R^2$ values. Putting in a dummy variable for said systems reveals that the six systems that exhibit a decline in growth during their lifetime are significantly different from those that exhibit an increase in growth at the .05 level (slope=.198, $R^2 = 16.8\%$).

With regards to the cutoff points between levels, we define that as the median visibility fan-in and median visibility fan-out. Interestingly, that has never been a problem in previous studies, possibly because the systems that they looked at were much larger in

\(^{12}\) When added, the lagged variable *defects per kloc* (t-1) proves to be a very significant explainer of defects, as one might expect, explaining approximately 48% of the dependent variable.
size. Here, looking at a subset of our sample reveals that a sizable number of systems have files that lie right on the median. Although the results are in line with those of previous studies, a better method may be necessary to deal with such files.

Cyclomatic complexity controlled for system size appears to have no significance, neither in terms of correlation nor as a predictor, with respect to defects per kloc, though it does appear to show significant positive correlation with reopened defects, as discussed in the following section.

Considering model IV from table 4 and the mean system-version from table 5, our linear regression equation becomes:

\[
DEFPKLOC = .685 + .001(FILES) + 1.001(CIKLOC) + 7.737(FODENS) + 1.105(CORE) \\
- .928(SHARED) + .779(CONTROL)
\]

That is to say, an increase of one unit in percentage change in kloc from one version to another results in an increase of 1.001 in defects per kloc, given that all other variables remain constant. Similarly, an increase of one unit in first-order density, that is, the density of direct dependencies, results in an increase, on average, of 7.737 in defects per kloc given that all other variables remain constant. The fact that the core and control components have positive coefficients and shared has a negative one, meaning that increases in core and control result in more defects and increases in shared result in fewer defects, is consistent with previous work (MacCormack and Sturtevant 2011).

Furthermore, we find that the coefficient of the core indicates a much more significant impact of having core files in a system than one might expect when considering just the interaction of shared (high visibility fan-in) and control (high visibility fan-out) variables.
<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILES</td>
<td>Files</td>
<td>307.46</td>
</tr>
<tr>
<td>KLOC</td>
<td>KLOC</td>
<td>128.53</td>
</tr>
<tr>
<td>CIKLOC</td>
<td>% change in KLOC</td>
<td>.07</td>
</tr>
<tr>
<td>FODENS</td>
<td>First-order density</td>
<td>.03</td>
</tr>
<tr>
<td>PROPCOST</td>
<td>Propagation cost</td>
<td>.27</td>
</tr>
<tr>
<td>CORE</td>
<td>Core size</td>
<td>.08</td>
</tr>
<tr>
<td>SHARED</td>
<td>Shared size</td>
<td>.31</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Control size</td>
<td>.30</td>
</tr>
<tr>
<td>DEFPKLOC</td>
<td>Defects per KLOC</td>
<td>.84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILES</td>
<td>Files</td>
<td>203</td>
</tr>
<tr>
<td>KLOC</td>
<td>KLOC</td>
<td>76.90</td>
</tr>
<tr>
<td>CIKLOC</td>
<td>% change in KLOC</td>
<td>.02</td>
</tr>
<tr>
<td>FODENS</td>
<td>First-order density</td>
<td>.03</td>
</tr>
<tr>
<td>PROPCOST</td>
<td>Propagation cost</td>
<td>.27</td>
</tr>
<tr>
<td>CORE</td>
<td>Core size</td>
<td>.05</td>
</tr>
<tr>
<td>SHARED</td>
<td>Shared size</td>
<td>.31</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Control size</td>
<td>.31</td>
</tr>
<tr>
<td>DEFPKLOC</td>
<td>Defects per KLOC</td>
<td>.63</td>
</tr>
</tbody>
</table>

Table 5: Properites of the mean and median system-versions

Using the mean values from table 5 in the linear regression equation produces:

\[
DEFPKLOC = .685 + .001(307.46) + 1.001(.07) + 7.737(.03) + 1.105(.08) - .928(.31) \\
+ .779(.30) = 1.33
\]
The mean system-version has approximately one defect per 1,000 lines of code, or put differently, around 171 defects, over a period of 6 months.

To see the effects of increases in our independent variables on defects in the mean system-version, we increase each variable at a time by its respective standard deviation and then multiply the result by the mean size, which is 128.53 kloc\textsuperscript{13}. The results for the mean system-version for the same time period of six months are shown in table 6.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Value</th>
<th>SD</th>
<th>Value+SD</th>
<th>DEFPKLOC</th>
<th>Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 FILES</td>
<td>307.46</td>
<td>285.94</td>
<td>593.40</td>
<td>1.62</td>
<td>208</td>
</tr>
<tr>
<td>2 CIKLOC</td>
<td>.07</td>
<td>.2493</td>
<td>.3193</td>
<td>1.58</td>
<td>203</td>
</tr>
<tr>
<td>3 FODENS</td>
<td>.03</td>
<td>.0224</td>
<td>.0524</td>
<td>1.50</td>
<td>193</td>
</tr>
<tr>
<td>4 CORE</td>
<td>.08</td>
<td>.0822</td>
<td>.1622</td>
<td>1.42</td>
<td>183</td>
</tr>
<tr>
<td>5 SHARED</td>
<td>.31</td>
<td>.1160</td>
<td>.4260</td>
<td>1.22</td>
<td>157</td>
</tr>
<tr>
<td>6 CONTROL</td>
<td>.30</td>
<td>.1129</td>
<td>.4129</td>
<td>1.42</td>
<td>182</td>
</tr>
<tr>
<td>7 1 and 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.86</td>
<td>240</td>
</tr>
<tr>
<td>8 1, 2 and 3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.04</td>
<td>262</td>
</tr>
</tbody>
</table>

Table 6: The effect of changing independent variables on defects

Here, we see that an increase in the number of source-code files has the most impact on defects as it results in 37 more defects. An increase in the percentage change also has a high impact on defects, resulting as it does in an increase of 32. Looking at likely combinations, we find that increases in both files and percentage change per kloc result in 69 more defects and increases in those two variables plus first-order density result in 91 more defects. What these values show is that our regression equation is useful in that

\textsuperscript{13} We use the total number of lines-of-code here, as outputted by our static analysis tool, to remain consistent, though we also record lines of code, excluding non-executed lines such as comments. Using that variable gives us similar models with a slightly lower $R^2$. For that variable, the mean is 8.33 and the median is 47.68.
increases in our independent variables have significant impact on the dependent variable rather than resulting in marginal increases of, say, one or two defects, which in reality would not be very significant.

4.3 Taking a look at reopened defects

The analysis for reopened defects begins with the same multiple regression analysis done for defects in general. The set of regression models where the dependent variable is reopened defects per kloc are as shown in table 7 below. They appear to show that a multiple regression model can explain very little of reopened defects. First-order density, although significant at the .10 level, has a much smaller coefficient than the same variable in the multiple regression model where defects per kloc is the dependent variable. Also, although reopened defects and mccabe per kloc appear to be moving together, mccabe per kloc does not appear to explain reopened defects within a linear model.

---

14 The following additional filter is used to exclude extreme points: defects reopened per kloc < 0.1 resulting in a sample size of 465, down from 494.
Table 7: Summary of the models developed

<table>
<thead>
<tr>
<th></th>
<th>Model I</th>
<th>Model II</th>
<th>Model III</th>
<th>Model IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2$</td>
<td>.023</td>
<td>.028</td>
<td>.025</td>
<td>.031</td>
</tr>
<tr>
<td>Adj $R^2$</td>
<td>.015</td>
<td>.017</td>
<td>.014</td>
<td>.014</td>
</tr>
<tr>
<td>Constant</td>
<td>.013***</td>
<td>.010*</td>
<td>.014***</td>
<td>.009</td>
</tr>
<tr>
<td>Files</td>
<td>.00002***</td>
<td>.00002</td>
<td>.00001**</td>
<td>.00002**</td>
</tr>
<tr>
<td>KLOC</td>
<td>-.00003**</td>
<td>-.00004</td>
<td>-.00003**</td>
<td>-.00003*</td>
</tr>
<tr>
<td>McCabe per KLOC</td>
<td>-.00002</td>
<td>-.00003</td>
<td>-.00002</td>
<td>-.00003</td>
</tr>
<tr>
<td>% change in KLOC</td>
<td>-.001</td>
<td>-.001</td>
<td>-.001</td>
<td>-.001</td>
</tr>
<tr>
<td>First-order density</td>
<td>.075</td>
<td></td>
<td>.093†</td>
<td></td>
</tr>
<tr>
<td>Propagation cost</td>
<td></td>
<td>-.005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core size</td>
<td></td>
<td></td>
<td>.015</td>
<td></td>
</tr>
<tr>
<td>Shared size</td>
<td></td>
<td></td>
<td>-.002</td>
<td></td>
</tr>
<tr>
<td>Control size</td>
<td></td>
<td></td>
<td>-.001</td>
<td></td>
</tr>
</tbody>
</table>

*Dependent variable = defects reopened per kloc († $p < .10$, * $p < .05$, ** $p < .01$, *** $p < .001$)

In reality, we lose a lot of statistical power due to the small number of reopened defects in the dataset – just 951 defects out of 44,278 fixed defects – and therefore end up with a large number of system-versions with zero reopened defects. For this reason, we choose to perform a direct logistic regression analysis and look at the likelihood that a reopened defect appears in a system-version, i.e. we create a new dependent variable called has reopened and assign it a binary value depending on whether the system-version has reopened defects. A little over half of the system-versions do not have reopened defects, while the others have one or more (242 vs. 223).

The model, containing the same predictors as before, is statistically significant at the .001 level ($\chi^2 = 43.54$). Hence, the model is able to differentiate between system-versions that include reopened defects and those that don’t include them. It explains between 13.6% and 18.1% (Cox and Snell $R^2$ and Nagelkerke $R^2$, respectively) of the total variation in the existence of reopened defects and correctly classifies 64.5% of cases, which is an
improvement over the baseline model, with none of the predictors included, which correctly classifies 52% of cases. The variable shared size appears to be the only independent variable that is statistically significant; its Exp(B) value of .117 indicates that for every unit increase in shared size, system-versions are .1 times less likely to have reopened defects. The results of the logistic analysis are shown in table 8.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>S.E.</th>
<th>Wald</th>
<th>df</th>
<th>p</th>
<th>Exp(B)</th>
<th>95% C.I. for EXP(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lower</td>
</tr>
<tr>
<td>Files</td>
<td>.001</td>
<td>.001</td>
<td>1.955</td>
<td>1</td>
<td>.162</td>
<td>1.001</td>
<td>.999</td>
</tr>
<tr>
<td>KLOC</td>
<td>.003</td>
<td>.002</td>
<td>1.642</td>
<td>1</td>
<td>.200</td>
<td>1.003</td>
<td>.999</td>
</tr>
<tr>
<td>McCabe per KLOC</td>
<td>.003</td>
<td>.005</td>
<td>.369</td>
<td>1</td>
<td>.544</td>
<td>1.003</td>
<td>.993</td>
</tr>
<tr>
<td>% change in KLOC</td>
<td>.466</td>
<td>.403</td>
<td>1.336</td>
<td>1</td>
<td>.248</td>
<td>1.593</td>
<td>.723</td>
</tr>
<tr>
<td>First-order density</td>
<td>-7.444</td>
<td>6.707</td>
<td>1.232</td>
<td>1</td>
<td>.267</td>
<td>.001</td>
<td>.000</td>
</tr>
<tr>
<td>Core size</td>
<td>.576</td>
<td>1.555</td>
<td>.137</td>
<td>1</td>
<td>.711</td>
<td>1.778</td>
<td>.084</td>
</tr>
<tr>
<td>Shared size</td>
<td>-2.141</td>
<td>.954</td>
<td>5.034</td>
<td>1</td>
<td>.025</td>
<td>.117</td>
<td>.018</td>
</tr>
<tr>
<td>Control size</td>
<td>-.105</td>
<td>.997</td>
<td>.011</td>
<td>1</td>
<td>.916</td>
<td>.901</td>
<td>.128</td>
</tr>
<tr>
<td>Constant</td>
<td>-.285</td>
<td>.771</td>
<td>.137</td>
<td>1</td>
<td>.711</td>
<td>.752</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Logistic regression predicting likelihood of a system-version having reopened defects (n=465)

Doing a correlation analysis for reopened bugs reveals the results shown in table 9. Most intriguingly, reopened bugs appear to have a significant positive correlation with cyclomatic complexity, and a significant negative correlation with first-order density, a result that was not particularly expected given the results obtained for defects per kloc. This perhaps indicates that reopened defects are perhaps due to a different type of complexity than normal defects, that is, the type of complexity captured by cyclomatic complexity, which is independent paths in the codebase, rather than that captured by first-order density and propagation cost, where it is coupling. The direction and magnitude of its correlation coefficient for core size remains similar to that of defects per kloc.
<table>
<thead>
<tr>
<th>defects per kloc</th>
<th>reopened def. per kloc</th>
<th>mccabe per kloc</th>
<th>prop. cost</th>
<th>first-order density</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.576**</td>
<td>-</td>
<td>0.137**</td>
<td>-0.085</td>
<td>0.037*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>defects per kloc</th>
<th>% in core</th>
<th>% in shared</th>
<th>% in periph.</th>
<th>% in control</th>
<th>% change in kloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.100*</td>
<td>-0.129**</td>
<td>-0.047</td>
<td>0.042</td>
<td>0.242*</td>
<td></td>
</tr>
</tbody>
</table>

**Correlation is significant at the .01 level (2-tailed)
*Correlation is significant at the .05 level (2-tailed)

Table 9: Correlation coefficients for reopened defects per kloc using Spearman’s correlation coefficient (see appendix C for full correlation table)

The result brings about a suspicion as to whether the count of reopened defects truly captures said type of defects, leading to three possible outcomes:

1. The measure is wrong, i.e. triaging isn’t done well and some reopened defects aren’t marked as duplicates and end up as new defects.
2. Reopened defects capture not only defects that are reopened due to technical reasons, but also those that are reopened due to the issues afforded by a distributed collaboration environment.
3. Reopened defects actually are caused by a different type of complexity.

Investigating the first two points involved conducting an interview with a member of GNOME’s Bugsquad, Andre Klapper, as well as perusing the comments section of a subset of defect reports and observing the discussions therein. The interview was conducted via email and was primarily made up of open-ended questions that were meant to elicit tacit knowledge from a prominent member of the GNOME community. Questions such as “What factors would you say influence the rise and fall of bugs in
different releases of Evolution\textsuperscript{15} proved enlightening; they uncovered a set of idiosyncratic process and project-related quality-influencing factors within Evolution such as missing manpower at times, the dynamics of having both paid and volunteer developers, inconsistent bug reporting policies in GNOME, the fact that Ubuntu, a Linux-based operating system, dropped Evolution as their default email client in favor of Mozilla’s Thunderbird and the impact of that on the user-base and hence the number of bug reports, to name a few. During a discussion of modules’ proneness to defects, modules with legacy code, some of it dating back to 1999, were noted as being among the more challenging parts of the codebase to deal with because of their fragility and downright ugliness.

Another open-ended question on triaging practices, which made reference to our current assumptions, also proved enlightening as it further illuminated the notion that in distributed, volunteer-based, open-source development, consistency is a difficult aspiration. Nevertheless, within GNOME, there do appear to exist notions of process, guidelines and documentation that collectively help regulate quality.

Yet another question about the effect of GNOME’s distributed development environment on quality lead to an interesting discussion about culture and the different interpretations of urgency and quality in different cultures – following Novell’s takeover of Ximian, the company that started Evolution, development was outsourced to Novell India. Such a shift in manpower lead to temporal issues with quality, not because said developers are necessarily less skilled, but because of the change in project dynamics due to the shift in culture.

With regards to the reopened bug reports, we proceeded to categorize the reasons for a subset of 210 out of the 951 reopened defects in our dataset by going through their

\textsuperscript{15} Recall that Evolution is an email client and is part of the set of projects that make up GNOME.
comments sections and making note of the type of discussions that took place. That exercise revealed the categories and statistics shown in chart 10.

![Chart 10: Reasons for defect reopenings](image)

For 68% of reopenings, the reason appears to be due to the defect reappearing, either in a subsequent release or within the same release. For 7% of reopening, the reason appears to be regressions, which one could argue may be traced to coupling, i.e. breaking something in another component or a change in another component causing the defect to reappear. One comment, for example, reads:

"...going to have to reopen this. The fix I committed fixed saving-as-vcard but broke some other (arguably more important) features, like being able to edit lists at all. Have backed out the change..."

Only around 16% of reopenings are due to collaboration-type issues or human error, such as forgetting to attach a patch, forgetting to commit a patch, not realizing that the problem is with the reporter's personal environment or misunderstanding the fix or the component
being fixed due to inadequate documentation. Hence, the majority of said defects appear to be due to genuine issues that one could argue may be traced to software complexity.

As a result of both activities, it would appear that a good triaging does in fact happen and with a high level of success as is evident from the number of defects that are, for example, marked as duplicates. Projects do benefit from the work of the GNOME Bugsquad and their comments and actions are noticeable in a lot of the discussions.

We then looked at another statistical technique, Pareto analysis, which states that 80% of problems come from 20% of possible sources (Jalote 2002)\textsuperscript{16}. In software, one can apply it to quality management in order to see which components have the most number of defects and using that information to drive the defect prevention process. It is useful to do such an analysis to see if our two types of defects are more or less driven by the same components. Doing such an analysis reveals that indeed around 80% of defects lie in around 20% of components. For both types of defects, the 20% of components appear to be the same, as shown in charts 11 and 12 for a sample system: Evolution. This appears to indicate that at least in terms of components, reopened defects don’t appear to be fundamentally different than normal defects, that is, we do not find a component with an unusually high number of reopened defects and a low number of defects in general.

\textsuperscript{16} In other domains, it can be redefined any number of ways and the ratio has been shown to hold.
Chart 11: Reopened defects in Evolution per component (v2.0 to v2.32)

Chart 12: Fixed defects in Evolution per component (v2.0 to v2.32)
Chapter 5

Discussion and practical implications

Our results show that architectural variables such as coupling not only correlate to defects but also have predictive power and can hence explain defects. Our measure of direct dependencies proves a strong predictor indicating that even for medium-sized systems, modularity remains an important quality attribute. The practical implication of this finding is that even in a cross-sectional dataset such as ours, more modularity can be shown to lead to fewer defects. Furthermore, source-code files that have cyclic dependencies between them (core components) remain one of the main predictors of defects, with a smaller core being indicative of fewer defects. Utility files that predominantly have inward dependencies are a more desirable class of files given that our statistical models show that increasing them leads to fewer defects.

Our measure of percentage-change between releases proves one of the strongest predictors of defects as well as the most stable variable. Increasing percentage-change by one standard deviation increases defects by around 120% and increasing it along with files and first-order density increases defects by over 150%. Viewed within the context of process, the result indicates that the more lines of code one adds or removes between releases, the more likely it is for one to introduce defects. Perhaps it is intuitive that the more new code one introduces, the more likely it is that one will consequently introduce defects, but also, perhaps the practical lesson is that a system must be given a chance to breathe and stabilize between releases. This phenomenon may be more of a side effect of the agile-type processes used in open-source projects, which involve little preemptive planning and a lot of iterative refining and refactoring. As some studies have observed,
more work needs to be done in looking into the relationship between agile-type processes and quality (Brown, et al. 2010). The take-away here is that if one sees that the process being used results in a high number of code being added or removed in every release and notices too that the oscillation is impacting quality, perhaps it is time to take a step back and make the development effort more predictable, or perhaps it is time to put in more stringent QA processes such as automated regression testing, combining it with other testing techniques as well. It may be useful for a future study to look at reopened defects in closed-source systems, or more generally, systems that use a more rigid development process.

Though propagation cost has been used in previous studies of software quality, this is the first study to use it with a cross-sectional dataset. What we find is that although breaking up the measure into four measures based on visibility fan-in and fan-out reveals similar results to previous studies, namely that a smaller core results in fewer defects, the measure of propagation cost itself appears to be insignificant when put in a multiple regression model. Perhaps the systems' sizes are still too small for propagation cost, despite our attempts to work with a small subset of the original set of GNOME projects that met our criteria for adequate complexity. With smaller systems, it is really the talent of the small set of contributors that zero-out the effects of complexity. It could also be that for systems with fewer core components and more shared components, as is the case here, propagation cost does not behave the same and may in fact be a negative predictor of defects due to it being driven by a high number of desirable components such shared and periphery components. Doing a stepwise linear regression analysis while splitting the data file based on projects reveals that for three of the projects, however, propagation cost is significant and positive and for five of them, it is significant and negative. Looking closely at the various metrics for those groups of systems reveals no clear similarities between them. For example, with Gnome Control Center, GnomeVFS and Gtranslator, neither are they of similar sizes (59,000, 103,000 and 29,000 lines of code, respectively) nor are their dependency data particularly different than the other projects.
Also, no clear patterns can be discerned from the systems’ inclusion patterns into GNOME releases or their defect and file activity.

In terms of reopened defects, the logistic regression model shows that between 13.6% and 18.1% of reopened defects are driven by our predictors, with the number of shared files in a system, i.e. files that don’t depend on a lot of files, but have a lot of files depend on them, being the only significant contributor. So in terms of dependencies, we can say that an increase in shared files results in fewer reopened defects. Although the odds-ratio is quite small (.117), it is in the same direction as defects in general. What the other analyses suggest is that the reappearance of defects is the main drivers of reopenings. Lack of documentation, full clarity about the component being worked on and the code being modified and the lack of consistent regression testing practices and QA in general are in turn the main drivers of defects reappearing. Collaboration-type issues, such as misunderstandings and misinterpretations of the defect report, defect-tracking tool and the fix itself do contribute to reopenings, but to a lesser extent. The existence of a triaging document with guidelines about how to triage defects is helpful in limiting the number of new defect reports that are created for existing or previously closed issues. Good tacit knowledge of the system by contributors and bugmasters saves a lot of time and effort by ensuring that if, say, a newcomer reopen a defect claiming it to be a duplicate of another, someone else will quickly point out that it is not due to some subtle difference.

The correlation we see between reopened defects and cyclomatic complexity suggests that perhaps construction errors are the main drivers of said type of defects rather than architectural drivers. This appears to be in line with previous work that found that for smaller projects, construction errors can make up about 75% of errors (Jones 1998, McConnell 2004). Given that our dataset is made up of small to moderately sized projects (mean kloc=128.53, median kloc=76.90), such a conclusion makes intuitive sense. It would be interesting to see the effect of larger systems on the number of
reopened defects. The take-away is that for projects of this size, perhaps focusing on program construction rather than design and architectural factors can lead to fewer reopened defects. By that, one means following coding standards and conventions, using best-practices for writing classes, functions, branches, variable names and data types, etc. as well as making use of techniques such as defensive programming and the use of unambiguous logic. This can also mean avoiding code that is too clever. A recent article by one of YouTube’s original engineers, Mike Solomon, provided several suggestions for achieving scalability, one of which was “Dummer code is easier to grep for and easier to maintain” (Hoff 2012).

For managers, our work highlights the importance of managerial decisions made within a software system’s lifetime, particularly with respect to deciding what to put in a codebase for a given release. Given a tight schedule, it would appear, based on our results and on anecdotal evidence, that choosing to exclude features is a less costly path to take than that of putting in features and then reworking them post-deployment. Our work highlights the critical importance of being cognizant of the types of dependencies that exist within a codebase: using a wrong criterion to modularize a system may in fact lead to no fewer dependencies at the code level. A non-technical manager is typically more aware of a system’s business components and workflows and may therefore delegate technical decisions to developers. It is for this reason that visualization tools such as DSMs can prove useful in practice, as a tool for managers to see the codebase in a different light. Furthermore, we find through our qualitative results that the existence of processes and of good communication channels helps in unifying a team’s technical vocabulary and reducing misunderstandings and misinterpretations. We find that collaboration is done with good success in GNOME through its defect-tracking system and posit that discussions are as important in driving quality as the previously mentioned architectural variables. We also find that fluctuating man-power can be problematic and may indeed impact the quality of the product and process.
Consistency in QA remains an important attribute, given that inconsistencies can cause comparing data from different systems to become more challenging, as it requires one to codify and control for more variables. In this study, we looked at a set of systems that loosely coexisted under the auspices of GNOME, but in reality, were independently developed. We found that not only did projects use the defect-tracking system, which had clear guidelines, in different ways, but they also produced architectures that were in some cases significantly different, with some being very modular and some being close to monolithic. Furthermore, knowledge of the idiosyncrasies of a project allows an outside observer to make more accurate observations: a decrease in defects may be interpreted as being due to an improved QA process, but in fact, it may be due to a decreased user-base, as was the case with Evolution when Ubuntu dropped it as its default mail application in favor of Thunderbird.

The existence of historical data is also a critical part of improving process and product. Without defect-tracking and an archive of previous releases, it is difficult if at all possible to do much analysis of the state of a system’s health and argue for how best to improve it. The lack of a dedicated QA team can reduce the quality of the defect data, given that it can result in badly triaged defects and inconsistent uses of a defect report’s fields. This may be a challenge in open-source projects where contributors may be volunteers, hence it is important for one to be cognizant of this limitation when doing any kind of analysis on open-source data.

Avenues for future work are plenty. In particular, the area of reopened defects appears to be uncharted with only a few studies done on specific systems. It would be interesting for a future study to look at reopened defects in more structured, perhaps even closed-source, projects and see if the architectural measures used herein prove more predictive of reopened defects.
References


—. *The Mythical Man-Month*. Addison-Wesley, 1975.


IBM. *IBM SPSS Software*. https://www-01.ibm.com/software/analytics/spss/.


Appendices

Appendix A: Data-processing steps
Appendix B: First-order matrix to visibility matrix
Appendix C: Full correlation table
Appendix D: Subset of dataset showing dependencies
Appendix A: Data-processing steps

Source-code data
The source-code for any set of systems-versions has to be imported to individual Understand projects (.udb files) in order to be able to do any kind of static analysis on them. The data are, hence, processed for each system-version as follows:

1. Create a project (.UDB file) in Understand and import the codebase
2. Generate a [system-version name].csv file in Understand of the codebase’s dependencies
3. Run extract_files_and_deps.pl on (2)
4. Run metrics_for_data_file_generator.m on the .cvs.deps file generated in (3)
5. Run project_metrics.pl in Understand on the .udb file
6. Add data generated in (4) and (5) to the dataset
7. Run necessary analyses on the dataset within SPSS

Defect activity data
For defect activity data, a script takes care of making the necessary http calls and then retrieving and consolidating the data as follows:

1. Run bugzilla_data_gatherer, which retrieves a set of .csv files generated by GNOME’s Bugzilla for all system-versions
2. Consolidate those values with the dataset by adding them to the defects variable for all system-versions

Removing singletons
The process of removing singletons is done via a Perl script as follows:

1. Run extract_files_and_deps_batch_processor.sh on each system-version’s directory to get a list of files that have inward or outward dependencies
2. Run `remove_singletons_batch_processor.sh` on all system-version directories, taking the `.cvs.files` file from (1) as input. This script removes all files that are in the directory or any of its subdirectories and not in the `.cvs.files` file.

3. Update the files, kloc and kloc code variables in the dataset.

### Scripts

More generally, a set of scripts help facilitate the processing of the data. These are described herein and elaborated on in the table below:

- **project_metrics.pl**: extracts code-related metrics such as file counts, lines of code, cyclomatic complexity, etc.
- **metrics_for_data_file_generator.m**: generates a first-order DSM showing the direct dependencies between files in a codebase; generates a visibility DSM showing both direct and indirect dependencies; calculates first-order density, propagation cost and percent of files in {core, periphery, shared and control} variables.
- **extract_files_and_deps.pl**: takes a map of direct dependencies in a codebase and processes it by replacing file names with integers and collapsing repeat calls between the same pair of files into a single line; the map is finally sorted; the set of file names is saved in a separate file.
- **remove_singletons.pl**: goes through a system-release’s directory and removes files that have no inward or outward dependencies from or to other files in the system.
- **bugzilla_data_gatherer**: retrieves defect activity data for all system-versions from GNOME’s Bugzilla and records the metrics.
- **remove_singletons_batch_processor.sh**, **extract_files_and_deps_batch_processor.sh**: batch process the two tasks for all system-versions.
<table>
<thead>
<tr>
<th>Script</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Understand</strong></td>
<td><code>project_metrics.pl</code></td>
<td>A <code>.csv</code> file containing metrics generated by Understand</td>
</tr>
<tr>
<td></td>
<td><code>.udb</code> file(^{17})</td>
<td>(e.g. complexities, file counts)</td>
</tr>
<tr>
<td><strong>MATLAB</strong></td>
<td><code>metrics_for_data_</code></td>
<td>Prop cost, density, percent in</td>
</tr>
<tr>
<td></td>
<td><code>file_generator.m</code></td>
<td>{core, periphery, shared, control} and related metrics</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td><code>extract_files_</code></td>
<td>A <code>.csv.deps</code> file containing a</td>
</tr>
<tr>
<td></td>
<td><code>and_deps.pl</code></td>
<td>sorted list of dependencies, a</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>.csv.files</code> file containing the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name of all files(^{18})</td>
</tr>
<tr>
<td></td>
<td><code>remove_</code></td>
<td>Adjusted system-release directory containing only</td>
</tr>
<tr>
<td></td>
<td><code>singletons.pl</code></td>
<td>files with dependencies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bugzilla_data_</code></td>
<td>A set of <code>.csv</code> files generated by Bugzilla that include</td>
</tr>
<tr>
<td></td>
<td><code>gatherer</code></td>
<td>all defect activity data</td>
</tr>
</tbody>
</table>

\(^{17}\) A system-version codebase imported into Understand

\(^{18}\) Based on a script written by Dan Sturtevant

*Appendix table 1: Scripts used, along with their respective inputs and outputs*
Appendix B: First-order matrix to visibility matrix

The set of DSMs below shows the matrices for GTK+ version 2.28, raised to successive powers of n, where the first matrix shows only direct dependencies and the last one shows indirect dependencies for all path lengths. The first-order density for the system is .89% and the propagation cost is 42.10%.
### Appendix C: Full correlation table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. defects per kloc</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. reopened defects per kloc</td>
<td>.599*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. propagation cost</td>
<td>-.096*</td>
<td>-.082</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. first-order density</td>
<td>.112'</td>
<td>-.002</td>
<td>.335*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. core size</td>
<td>.116'</td>
<td>.113'</td>
<td>-.707*</td>
<td>-.383*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. periphery size</td>
<td>-.015</td>
<td>.002</td>
<td>.575*</td>
<td>.264*</td>
<td>-.352*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. shared size</td>
<td>-.161*</td>
<td>-.068</td>
<td>-.170*</td>
<td>-.075</td>
<td>-.155*</td>
<td>-.637*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. control size</td>
<td>.121*</td>
<td>-.016</td>
<td>-.230*</td>
<td>-.070</td>
<td>-.007</td>
<td>-.689*</td>
<td>.103*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. percent change in kloc</td>
<td>.304*</td>
<td>.064</td>
<td>-.025</td>
<td>.011</td>
<td>.031</td>
<td>-.016</td>
<td>-.025</td>
<td>.028</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. kloc</td>
<td>-.137*</td>
<td>-.015</td>
<td>.084</td>
<td>-.474*</td>
<td>-.092*</td>
<td>.085</td>
<td>.035</td>
<td>-.106*</td>
<td>-.088</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. kloc code</td>
<td>-.110*</td>
<td>-.002</td>
<td>.099*</td>
<td>-.499*</td>
<td>-.045</td>
<td>.064</td>
<td>-.026</td>
<td>-.043</td>
<td>-.086</td>
<td>.972*</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. files</td>
<td>-.053</td>
<td>.049</td>
<td>.067</td>
<td>-.610*</td>
<td>.144*</td>
<td>-.071</td>
<td>-.025</td>
<td>.034</td>
<td>-.074</td>
<td>.860*</td>
<td>.893*</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13. mccabe per kloc</td>
<td>-.057</td>
<td>.003</td>
<td>.094*</td>
<td>-.187*</td>
<td>.069</td>
<td>.070</td>
<td>-.102*</td>
<td>-.058</td>
<td>-.055</td>
<td>.158*</td>
<td>.276*</td>
<td>.320*</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14. mccabe sum</td>
<td>-.130*</td>
<td>-.016</td>
<td>.100*</td>
<td>-.495*</td>
<td>-.040</td>
<td>.088</td>
<td>-.025</td>
<td>-.086</td>
<td>-.081</td>
<td>.960*</td>
<td>.982*</td>
<td>.891*</td>
<td>.363*</td>
<td>1</td>
</tr>
</tbody>
</table>

---

The significant positive correlation coefficient of .137 between *reopened defects per kloc* and *mccabe per kloc* in section 4.3 is using Spearman's correlation coefficient.
Appendix D: Subset of dataset showing dependencies

A sample set showing a comparison of core, periphery, shared and control sizes for first-order and visibility matrices.

<table>
<thead>
<tr>
<th>project</th>
<th>release</th>
<th>prop. cost</th>
<th>density</th>
<th>Using indirect dependencies</th>
<th>Using direct dependencies</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% in core</td>
<td>% in periph.</td>
<td>% in shared</td>
</tr>
<tr>
<td>anuta</td>
<td>2.2</td>
<td>33%</td>
<td>2%</td>
<td>1%</td>
<td>27%</td>
<td>26%</td>
</tr>
<tr>
<td>anjuta</td>
<td>2.10</td>
<td>13%</td>
<td>1%</td>
<td>22%</td>
<td>22%</td>
<td>28%</td>
</tr>
<tr>
<td>anjuta</td>
<td>2.12</td>
<td>9%</td>
<td>1%</td>
<td>28%</td>
<td>31%</td>
<td>20%</td>
</tr>
<tr>
<td>balsa</td>
<td>2.10</td>
<td>37%</td>
<td>4%</td>
<td>1%</td>
<td>22%</td>
<td>48%</td>
</tr>
<tr>
<td>balsa</td>
<td>2.12</td>
<td>39%</td>
<td>4%</td>
<td>1%</td>
<td>18%</td>
<td>47%</td>
</tr>
<tr>
<td>banshee</td>
<td>2.14</td>
<td>8%</td>
<td>1%</td>
<td>16%</td>
<td>24%</td>
<td>32%</td>
</tr>
<tr>
<td>banshee</td>
<td>2.16</td>
<td>8%</td>
<td>1%</td>
<td>16%</td>
<td>24%</td>
<td>32%</td>
</tr>
<tr>
<td>beast</td>
<td>2.10</td>
<td>24%</td>
<td>1%</td>
<td>10%</td>
<td>27%</td>
<td>40%</td>
</tr>
<tr>
<td>beast</td>
<td>2.12</td>
<td>24%</td>
<td>1%</td>
<td>10%</td>
<td>27%</td>
<td>40%</td>
</tr>
<tr>
<td>dia</td>
<td>2.10</td>
<td>27%</td>
<td>2%</td>
<td>3%</td>
<td>20%</td>
<td>31%</td>
</tr>
<tr>
<td>dia</td>
<td>2.12</td>
<td>26%</td>
<td>2%</td>
<td>3%</td>
<td>19%</td>
<td>31%</td>
</tr>
<tr>
<td>eog</td>
<td>2.10</td>
<td>29%</td>
<td>5%</td>
<td>5%</td>
<td>22%</td>
<td>30%</td>
</tr>
<tr>
<td>eog</td>
<td>2.12</td>
<td>29%</td>
<td>5%</td>
<td>5%</td>
<td>22%</td>
<td>30%</td>
</tr>
<tr>
<td>eog</td>
<td>2.10</td>
<td>11%</td>
<td>3%</td>
<td>22%</td>
<td>29%</td>
<td>28%</td>
</tr>
<tr>
<td>eog</td>
<td>2.12</td>
<td>11%</td>
<td>3%</td>
<td>19%</td>
<td>30%</td>
<td>24%</td>
</tr>
<tr>
<td>epiphany</td>
<td>2.10</td>
<td>32%</td>
<td>2%</td>
<td>13%</td>
<td>16%</td>
<td>34%</td>
</tr>
<tr>
<td>epiphany</td>
<td>2.12</td>
<td>30%</td>
<td>2%</td>
<td>11%</td>
<td>18%</td>
<td>35%</td>
</tr>
<tr>
<td>evince</td>
<td>2.10</td>
<td>10%</td>
<td>3%</td>
<td>18%</td>
<td>21%</td>
<td>29%</td>
</tr>
<tr>
<td>evince</td>
<td>2.12</td>
<td>11%</td>
<td>2%</td>
<td>22%</td>
<td>25%</td>
<td>26%</td>
</tr>
<tr>
<td>evolution</td>
<td>2.10</td>
<td>19%</td>
<td>0%</td>
<td>13%</td>
<td>21%</td>
<td>29%</td>
</tr>
<tr>
<td>evolution</td>
<td>2.12</td>
<td>28%</td>
<td>1%</td>
<td>5%</td>
<td>26%</td>
<td>25%</td>
</tr>
<tr>
<td>f-spot</td>
<td>2.10</td>
<td>14%</td>
<td>2%</td>
<td>14%</td>
<td>22%</td>
<td>29%</td>
</tr>
<tr>
<td>f-spot</td>
<td>2.12</td>
<td>26%</td>
<td>3%</td>
<td>3%</td>
<td>22%</td>
<td>46%</td>
</tr>
<tr>
<td>gdm</td>
<td>2.10</td>
<td>35%</td>
<td>6%</td>
<td>0%</td>
<td>29%</td>
<td>42%</td>
</tr>
<tr>
<td>gdm</td>
<td>2.12</td>
<td>33%</td>
<td>5%</td>
<td>3%</td>
<td>26%</td>
<td>43%</td>
</tr>
<tr>
<td>gdm</td>
<td>2.8</td>
<td>35%</td>
<td>6%</td>
<td>0%</td>
<td>27%</td>
<td>44%</td>
</tr>
<tr>
<td>gedit</td>
<td>2.10</td>
<td>34%</td>
<td>5%</td>
<td>3%</td>
<td>24%</td>
<td>26%</td>
</tr>
<tr>
<td>gedit</td>
<td>2.12</td>
<td>34%</td>
<td>5%</td>
<td>3%</td>
<td>24%</td>
<td>26%</td>
</tr>
<tr>
<td>Application</td>
<td>Version</td>
<td>Core</td>
<td>Periphery</td>
<td>Shared</td>
<td>Control</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>------</td>
<td>-----------</td>
<td>--------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>glom</td>
<td>2.12</td>
<td>9%</td>
<td>1%</td>
<td>18%</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.14</td>
<td>47%</td>
<td>2%</td>
<td>1%</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>gmime</td>
<td>2.10</td>
<td>15%</td>
<td>4%</td>
<td>21%</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.12</td>
<td>15%</td>
<td>4%</td>
<td>21%</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>gnome-key.</td>
<td>2.10</td>
<td>19%</td>
<td>12%</td>
<td>4%</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>gnome-key.</td>
<td>2.12</td>
<td>19%</td>
<td>12%</td>
<td>4%</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>gnome-net.</td>
<td>2.10</td>
<td>4%</td>
<td>2%</td>
<td>14%</td>
<td>31%</td>
<td></td>
</tr>
<tr>
<td>gnome-net.</td>
<td>2.12</td>
<td>4%</td>
<td>2%</td>
<td>14%</td>
<td>31%</td>
<td></td>
</tr>
<tr>
<td>gnome-vfs</td>
<td>2.10</td>
<td>32%</td>
<td>2%</td>
<td>0%</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>gnome-vfs</td>
<td>2.12</td>
<td>31%</td>
<td>2%</td>
<td>0%</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>gnnumeric</td>
<td>2.10</td>
<td>36%</td>
<td>2%</td>
<td>1%</td>
<td>23%</td>
<td></td>
</tr>
<tr>
<td>gnnumeric</td>
<td>2.12</td>
<td>41%</td>
<td>2%</td>
<td>3%</td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>gthumb</td>
<td>2.10</td>
<td>20%</td>
<td>3%</td>
<td>11%</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>gthumb</td>
<td>2.12</td>
<td>34%</td>
<td>3%</td>
<td>6%</td>
<td>18%</td>
<td></td>
</tr>
<tr>
<td>gtk</td>
<td>2.10</td>
<td>33%</td>
<td>1%</td>
<td>2%</td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>gtk</td>
<td>2.12</td>
<td>33%</td>
<td>1%</td>
<td>3%</td>
<td>18%</td>
<td></td>
</tr>
<tr>
<td>gtranslator</td>
<td>2.10</td>
<td>22%</td>
<td>6%</td>
<td>0%</td>
<td>38%</td>
<td></td>
</tr>
<tr>
<td>gtranslator</td>
<td>2.12</td>
<td>22%</td>
<td>6%</td>
<td>1%</td>
<td>36%</td>
<td></td>
</tr>
<tr>
<td>libsoup</td>
<td>2.10</td>
<td>72%</td>
<td>8%</td>
<td>0%</td>
<td>81%</td>
<td></td>
</tr>
<tr>
<td>libsoup</td>
<td>2.12</td>
<td>72%</td>
<td>8%</td>
<td>0%</td>
<td>81%</td>
<td></td>
</tr>
<tr>
<td>libxml2</td>
<td>2.10</td>
<td>21%</td>
<td>4%</td>
<td>0%</td>
<td>42%</td>
<td></td>
</tr>
<tr>
<td>libxml2</td>
<td>2.12</td>
<td>21%</td>
<td>4%</td>
<td>0%</td>
<td>41%</td>
<td></td>
</tr>
<tr>
<td>metacity</td>
<td>2.10</td>
<td>70%</td>
<td>6%</td>
<td>0%</td>
<td>82%</td>
<td></td>
</tr>
<tr>
<td>metacity</td>
<td>2.12</td>
<td>70%</td>
<td>6%</td>
<td>0%</td>
<td>82%</td>
<td></td>
</tr>
<tr>
<td>nautilus</td>
<td>2.10</td>
<td>32%</td>
<td>2%</td>
<td>5%</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>nautilus</td>
<td>2.12</td>
<td>33%</td>
<td>2%</td>
<td>5%</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>orbit2</td>
<td>2.10</td>
<td>26%</td>
<td>3%</td>
<td>11%</td>
<td>31%</td>
<td></td>
</tr>
<tr>
<td>orbit2</td>
<td>2.12</td>
<td>26%</td>
<td>3%</td>
<td>11%</td>
<td>31%</td>
<td></td>
</tr>
<tr>
<td>planner</td>
<td>2.10</td>
<td>33%</td>
<td>3%</td>
<td>6%</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>planner</td>
<td>2.12</td>
<td>36%</td>
<td>3%</td>
<td>7%</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>rhythmbox</td>
<td>2.10</td>
<td>28%</td>
<td>3%</td>
<td>12%</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>rhythmbox</td>
<td>2.12</td>
<td>29%</td>
<td>3%</td>
<td>5%</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>seahorse</td>
<td>2.10</td>
<td>26%</td>
<td>6%</td>
<td>5%</td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>seahorse</td>
<td>2.12</td>
<td>28%</td>
<td>5%</td>
<td>7%</td>
<td>20%</td>
<td></td>
</tr>
</tbody>
</table>

Appendix table 3: Core, periphery, shared, control sizes for a subset of systems