A Replication Case Study to Measure the Architectural Quality of a Commercial System

Derek Reimanis[1], Clemente Izurieta[1], Rachael Luhr[1], Lu Xiao[2], Yuanfang Cai[2], Gabe Rudy[3]
{derek.reimanis, clemente.izurieta, rachael.luhr}@cs.montana.edu, 01-406-994-3720
{lx52, yfcai}@cs.drexel.edu, 01-215-895-0298
rudy@goldenhelix.com, 01-406-585-8137

ABSTRACT

Context: Long-term software management decisions are directly impacted by the quality of the software's architecture. **Goal:** Herein, we present a replication case study where structural information about a commercial software system is used in conjunction with bug-related change frequencies to measure and predict architecture quality. **Method:** Metrics describing history and structure were gathered and then correlated with future bug-related issues; the worst of which were visualized and presented to developers. **Results:** We identified dependencies between components that change together even though they belong to different architectural modules, and as a consequence are more prone to bugs. We validated these dependencies by presenting our results back to the developers. The developers did not identify any of these dependencies as unexpected, but rather architectural necessities. **Conclusions:** This replication study adds to the knowledge base of CLIO (a tool that detects architectural degradations) by incorporating a new programming language (C++) and by externally replicating a previous case study on a separate commercial code base. Additionally, we provide lessons learned and suggestions for future applications of CLIO.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces*.

General Terms

Management, Measurement, Experimentation.

Keywords

Modularity violations, grime, technical debt, static analysis, architecture quality, case study, replication.

1. INTRODUCTION

Building confidence in previous results helps to increase the strength and the importance of findings. It is especially important to strive for external validation of results by independent researchers, as has been done by the replication study presented herein. To date, the field of Empirical Software Engineering lacks in the number of replication studies. Additionally, most of the existing guidelines found in the literature focus on formal experiments [3] [4] [7] [14]. In this paper, we present the findings of an external replication *case-study*. We present our results by borrowing from the existing experimentation terminology and we have structured our findings consistent with expected sections as delineated by Wohlin et al. [15].

The motivation behind this study stems from a desire to see if the techniques used by Schwanke et al. [13] to uncover architecture-related risks in a Java agile development environment (using architecture and history measures) can also be applied to a commercial C++ development environment. This is important because we wanted to evaluate the deployment of this technology in an industrial setting of a successful company with strict quality controls. We were also interested to see if the observations we make can be used to build consensus in explaining a form of architectural decay, where decay is defined as the structural breakdown of agreed upon solutions [6].

We applied CLIO [16], a tool designed to uncover modularity violations, to a commercial software system developed by a local bioinformatics company –Golden Helix[4]. The latter allowed us access to their software code base to investigate potential architectural disharmonies.

This paper is organized as follows: Section 2 discusses background and related work; Section 3 explains the importance of replication in empirical software engineering and our approach to classifying this study; Section 4 discusses the method followed by our replication; Section 5 explores how the method was carried out, including deviations and challenges encountered from the baseline method, results and developer feedback; Section 6 discusses the relation of our results to the baseline study. Section 7 discusses the threats to validity in our study; and Section 8 concludes with lessons learned from this study and suggestions of future work.

2. BACKGROUND AND RELATED WORK

2.1 Modularity Violations

Baldwin and Clark [1] define a module as "a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units." Identifying violations in modules (hereafter referred to as modularity violations) is important because it allows developers to find code that exhibits bad structural design. Identifying such violations early in the lifecycle leads to proactive module refactoring. However, early detection of modularity violations is difficult because they do not always exhibit negative influences on the functionality of the software system. It is entirely possible

ESEM'14, September 18–19, 2014, Torino, Italy.

http://dx.doi.org/10.1145/2652524.2652581

[1] Software Engineering Laboratory, Computer Science Dept., Bozeman, MT, USA

[2] Computer Science Dept., Drexel University, Philadelphia, PA, USA

[3] Golden Helix Inc., 203 Enterprise Blvd, Bozeman, MT, USA

[4] Golden Helix Inc.; http://www.goldenhelix.com

Table 1: Summary of different treatments between case studies

Factor	*Baseline Project*	*Our Project*
Programming Language	Java	C++
# of Developers	Up to 20	Up to 11
Project Lifetime	2 years	4 years
# Source Files	900	3903 (1569 C++, 267 C, 2067 h)
KSLOC	300	1300

for a system to function as intended, yet still contain modularity violations. If these violations are left uncorrected, they can lead to architectural decay, which would slowly cripple production.

Zazworka et al. [17] used the modularity violations findings from a CLIO case study and compared them to three other technical debt identification approaches. They found that modularity violations contribute to technical debt in the Hadoop open source software system. Technical debt [5] is a well-known metaphor that describes the tradeoffs between making short term decisions (i.e., time to market) at the expense of long term but high software quality (i.e., low coupling). The debt incurred during the lifetime of a software system can be measured as a function of cost (monetary or effort) with added interest. Often, debt happens because of quick and dirty implementation decisions –usually occurring when a development team is trying to meet a deadline. Technical debt is dangerous if not managed because it can result in a costly refactoring process. Techniques to slow down the accumulation of technical debt can benefit from early detection of modularity violations.

2.2 CLIO

CLIO was developed by Wong et al. [16] as a means to identify modularity violations in code. Wong et al. evaluated CLIO by running it on two different open source Java projects, Eclipse JDT[5] and Hadoop Common[6]. The results showed that hundreds of violations identified by CLIO were fixed in later versions of the software. CLIO finds violations within modules by looking not only at the source code of a project, but also at its version history. It helps developers identify unknown modular level violations in software. Although developers will identify some violations, specifically if the violations prove to be bothersome, the difficulty of finding all modularity violations is quite great. CLIO validates that its reports are useful by confirming that previously detected violations are indeed fixed in later versions of the software. The results that Wong et al. [16] obtained showed that CLIO could detect these modularity violations much earlier than developers who were manually checking for them. This means that CLIO can be used in software systems to identify modularity violations early in the development process to save time and money by not having to check for them manually.

Schwanke et al. [13] expanded upon this work by using CLIO on an agile industrial software development project. They looked specifically at the architectural quality of the software. They used a clustering algorithm to observe how files changed together without developer knowledge, and the impact that change had on the quality of the architecture, as measured by source code changes because of bugs. They reported several modularity violations to developers. The developers issued a refactoring because the modularity violations were (1) unexpected and (2) possibly harmful to their system. CLIO allowed them to see the exact number of files that were dependent on one another, and how those changes were affecting the structure of their project.

3. Replication in Software Engineering

Literature in the field concerning guidelines of replication studies only addresses experimental replication, not case study replication [4] [14]. Therefore, we have borrowed terminology from this literature to inform our work.

3.1 Importance of Replicating Case Studies

Experiment replication plays a key role in empirical software engineering [4] [14]. While many other domains construct hypotheses *in vitro*, software engineers are generally not granted that luxury. Empirical software engineering frequently involves humans, directly or indirectly, as experimental subjects, and human behavior is unpredictable and not repeatable in a laboratory setting. Coupled with the prohibitive costs of formal experimentation, software engineering empiricists must look for alternatives. Instead, we must rely on repeated case studies in various contexts to construct a knowledge base suitable for a scientific hypothesis. This process, while requiring exhaustive work, allows for consensus building that can provide the necessary support to generate scientific claims.

3.2 Categories of Replication

Shull et al. [14] discuss two primary types of replications; *exact replications* and *conceptual replications*. Exact replications are concerned with repeating the procedure of a baseline experiment as closely as possible. Conceptual replications, alternatively, attempt to use a different experimental procedure to answer the same questions as the baseline experiment. The study presented in this paper utilizes an exact replication method.

Shull et al. [14] divide exact replications into two categories: *dependent replications* and *independent replications*. In dependent replications, researchers keep all elements of the baseline study the same. In independent replications, researchers may alter elements of the original study. An independent replication follows the same procedure as the original study, but tweaks experimental treatments to come to the same or a different result. If treatments are changed and the same result is found, researchers can conclude that the treatment in question probably has little or no effect on the outcome. However, if changing a treatment leads to different results, that treatment needs to be explored further.

Using Shull's terminology, we categorized this study as an independent replication, with five major treatment differences from what would be considered a dependent replication. These differences are illustrated in Table 1. First, the baseline study used a software project written in Java as their only treatment to the programming language factor. In our case, the treatment is the C++ programming language. In other words, our study lies in the context of a C++ programming language, which may provide different results from the baseline. Second, the comparative sizes of the development groups differed. The baseline study featured a development group of up to 20 developers working on the project at any given point in time [13].The C++ system analyzed in this paper has had a total of eleven contributing developers in its four year lifetime. Third, the software project in the baseline study had been in development for two years, while the project covered in

[5] The Eclipse Project; http://www.eclipse.org

[6] Apache Hadoop Common; http://hadoop.apache.org

our study has been in development for four years. Finally, the project in the baseline study features 300 kilo-source lines of code (KSLOC) in 900 Java files. The project in our study has 1300 KSLOC across 3903 source files, of which 1836 have a .cpp/.c extension, and 2067 are header files. Surprisingly, both projects have a similar ratio of LOC per source file (333 LOC per source file).

3.3 Replication Baseline

In the selected baseline study, Schwanke et al. [13] reported on a case study that measured architecture quality and discovered architecture issues by combining the analysis of software structure and change history. They studied three structured measures (file size, fan-in, and fan-out) and four history measures (file change frequency, file ticket frequency, file bug frequency, and pair of file change frequency). Their study included two parts: 1) Exploring different software measures; and 2) Uncovering architecture issues using those measures.

1) Exploring different software measures: First, they explored the relationship between each pair of measures (structure and history) using Kendall's *tau-b* rank correlation [8], which showed the extent to which any two measures rank the same data in the same order. This study provided an initial insight on whether those measures were indicative of software quality, which was approximated by the surrogate file bug frequency. Then they studied how predictive those measures were of software faults. The data they used spanned two development cycles of the subject system, release 1 (R1) and release 2 (R2). They illustrated how predictive the calculated measures from R1 were for faults that appeared in R2 using Alberg diagrams [9].

2) Uncovering architecture issues: After validating the measures, they were used to discover architecture issues using three separate approaches. First, Schwanke et al. ranked all files by different measures –worst first. They found that the top ranked files (outliers) were quite consistent for different measures. They showed those outliers to the developers to obtain feedback about potential architecture issues; however, the developers gave little response because they could not visualize these issues. To generate responses from developers, they used a static analysis tool named *Understand*™[7] to visualize the position of those outliers in the architecture. Using this method, they were able to discuss many of the outlier files with the developers. In some cases, the developers pointed out how severe the problems were. Finally, they used CLIO to investigate the structure and history of pairs of files and grouped structurally distant yet historical coupled files into clusters. For each cluster, its structure was visualized using *Understand*™ in a structure diagram, which illustrated how clusters which cross-cut different architecture layers could be severe, and gave hints about why they were coupled in history.

3.4 Major Findings of the Baseline

Schwanke et al. found that by using CLIO they could identify, predict, and communicate certain architectural issues in the system [13]. They found that a few key interface files contributed to the majority of faults in the software. Additionally, they discovered that the file size and fan-out metrics are good predictors of future fault-proneness. In the absence of historical artifacts, files that contain high measures of these metrics typically have a higher number of architectural violations. Finally, unknown to the developers, some of these files violated modularity in the system by creating unwanted connections between layers. These violations were visualized and presented to the developers who issued a refactoring thereafter.

[7] Understand; http://www.scitools.com

4. PROCEDURE

Following the procedure outlined in [13], our case study consisted of the following steps:

1) Data collection: The source code, version control history, and ticket tracking history of the software system in question were gathered.
2) Structure and history measurements: Measurements for common metrics were computed/collected across all versions of the software.
3) Validation: Measurements from the second-most recent release are correlated with fault measurements from the most recent release.
4) Prediction: Measurements from the most recent release are used to predict faults in upcoming future releases of the project.
5) Uncovering architecture problems: Measurements were sorted according to future fault impact and visualized. Outlier measurements present the most concern to system architecture quality, and were selected for further exploration.
6) Present findings to developers: Visualizations of the architecture of outlier modules were presented to developers with the intent of helping to realize the architectural quality of the system.

5. CASE STUDY

5.1 Setting

The project analyzed in this case study is named SNP & Variation Suite (SVS), and is the primary product of the bioinformatics company Golden Helix. We analyzed seven major releases of SVS.

SVS features 1.3 million lines of C++ source code spread out across 3903 source files. The project's structure is spread out across a total of 22 directories. In this study, we have chosen to define module as a directory, based on Parnas el al.'s definition [12]. We use the term directory and module interchangeably.

Eleven developers have contributed to this project over its four-year lifetime. The organization of the development group has an interesting hierarchy. The lead developer is also the Vice President of Product Development at Golden Helix. He plays a major role in not only developing SVS, but also in managing product development from a financial perspective. This means he has comprehensive knowledge of the software system when he makes management-related decisions, and therefore, is more aware of the technical debt present in the software than business-oriented managers.

5.2 Motivation

This project was chosen for three reasons. First, Golden Helix is a local software company with its developing team in close proximity to the authors, and is well known for their generous contributions to the community. The process presented in this study is a great opportunity to inform Golden Helix of the architectural quality of their flagship software. Second, applying the CLIO tool in different commercial settings will help future applications of CLIO. By clearly outlining the strengths, weaknesses, and lessons learned at the end of the study, we hope

to improve future applications of CLIO. Finally, no previous study that follows this methodology to detect modularity violations has considered a C++ project. Previous studies such as [16] [17] only looked at non-commercial Java projects. Using the C++ programming language as a treatment in this sense builds on the knowledge base of CLIO, extending what we know about this method.

5.3 Data Collection

Golden Helix strongly encourages developers to commit often, and keep commits localized to their section of change. These commits are stored in a Mercurial[8] repository, and the FogBugz[9] tool is used to track issues. Golden Helix switched repositories, from Apache Subversion (SVN)[10] to Mercurial, and ticket tracking tools, from Trac[11] to FogBugz, during the lifetime of SVS. Because this study focuses on the entirety of the project's lifetime, both the SVN repository and Trac ticket logs have been recovered and treated in the same manner as the current system. Each developer is expected to include references to ticket cases in their commits.

Similar to [13], the repository logs and issue tracking logs were extracted into a PostgreSQL[12] database. This allowed us to search for historical data using simple SQL queries.

We have grouped C/C++ source files and header source files together in this study. That is, for each C/C++ source file and its corresponding header file(s), the files are considered one and the same. For the remainder of this case study, we refer to the C/C++ source and corresponding header file pairs as a *file pair*. Measurements made in both files are aggregated together. There is a reason for doing this. Developers of SVS demand that source files and their corresponding header files be kept together in the same directory. When either a source file or a header file changes, the developers are expected to update the signatures in the corresponding file. This implies that any changes made to the latter are expected and hence do not constitute modularity violations. Our study is concerned with locating *unexpected* changes in modules of code. Therefore, including any information about header/source pairs changing together will lead to useless information.

5.4 Structure and History Metrics

Following the work of Schwanke et al. [13], the following metrics were gathered for all file pairs (*u)* across all seven versions of the software:

1. *File size*: The aggregated file size on disk of both elements in *u*, measured in bytes.
2. *Fan-in*: Within a project, *fan-in* of *u* is the sum of the number of references from any *v* (where *v* is defined identically similarly to *u*) pointing to *u*.
3. *Fan-out:* Within a project, *fan-out* of *u* is the sum of the number of references from *u* that point to any *v* (where *v* is defined identically similarly to *u*).
4. *Change frequency*: The number of times that any element in *u* is changed, according to the commit log. Commits where both elements of *u* are changed are only counted once.
5. *Ticket frequency*: The number of different FogBugz or Trac issue tickets referenced for which either element in *u* is modified. If both elements in *u* are modified with a reference to the same issue ticket, it is only counted once.
6. *Bug change frequency*: The number of different FogBugz or Trac **bug** issue tickets referenced for which either element in *u* is modified. If both elements in *u* are modified with a reference to the same **bug** issue ticket, it is only counted once.
7. *Pair change frequency*: For each file pair, *v*, in the project, the number of times in which *u* and *v* are modified in the same commit.

5.5 Validation

In an effort to validate the significance of our metric choices, several exploratory data analysis techniques were utilized. These include histogram inspection, scatter plot analysis, and correlation analysis. Although the system in question has gone through seven releases, in this paper we only present the results from the most recent release (release 7.5) and the release immediately preceding the most recent release (release 7). Hereafter, we refer to release 7.5 as the *present* state of the software, and release 7 as the *past*.

Similar to the baseline study, we found that data analysis across all other releases showed very similar results. The baseline study chose to focus their work on the most recent releases, because it is more representative of the system in the present time, and may provide better predictive power. We have followed suit because of the same reasons.

1) Histogram analysis

Histograms were generated for each metric in question. We focused on identifying distributions of each metric across releases. From the distributions, we identified outlier file pairs which Schwanke et al. [13] states are more prone to unexpected changes. For example, Figure 1 illustrates the change frequency metric across all releases of the software. The y-axis is shown as a logarithmic scale in base 4 to preserve column space. There is a typical exponential decay curve, suggesting that the majority of file pairs experienced few changes. However, there exist outliers with more than 180 changes per file (not shown, but aggregated to form the bin at x=180). This suggests that a surprising number of pairs (about 60) experience more than 180 changes. This is congruent with findings from [13] and their histogram analysis.

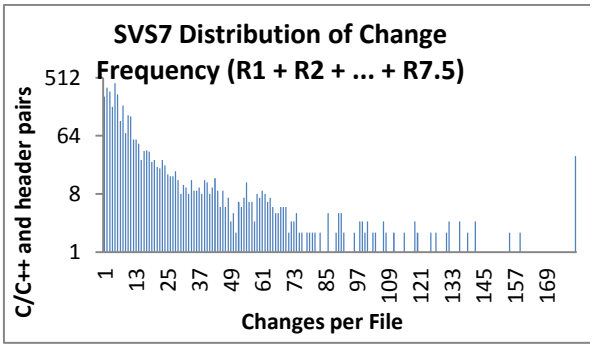

Figure 1: Histogram of change frequency across all releases. The x-axis shows change frequency. The y-axis shows a count of the number of pairs. Any pair with 180 or more changes was considered to be an outlier, and likely to contribute to many unexpected dependencies.

[8] Mercurial SCM; http://mercurial.selenic.com/

[9] FogBugz Bug Tracking; https://www.fogcreek.com/fogbugz/

[10] Apache Subversion; http://subversion.apache.org/

[11] Trac; http://trac.edgewall.org/

[12] PostgreSQL; http://www.postgresql.org/

2) Scatter Plot Analysis

Scatter plots were constructed for each metric gathered. When constructing scatter plots, we plotted the measure in release 7.5 on the y-axis and the measure of other metrics from release 7 on the x-axis. This gave us the opportunity to identify a possible relationship between past and present measurements. Figure 2 shows a scatter plot of change frequency in release 7.5 versus fan-out in release 7. There appears to be a slight linear correlation between the two, suggesting that change frequency in future releases can be predicted from fan-out in current or past releases.

This graph suggests that the fan-out of current or past file pairs may be used to predict the change frequency of the pair in the future. Our scatter plot analysis provided similar results as the baseline study by Schwanke et al [13].

3) Correlation Analysis

Rank-based correlation analysis was performed on the data to identify possible relationships between measurements in one release and fault measurements in a future release. Per the baseline study, we used the Kendall's *tau-b* rank correlation measure [8]. This non-parametric test was chosen instead of a Spearman or the parametric Pearson test because many of the values fall near zero. The Ordinary Least Squares (OLS) method of Spearman or Pearson performs poorly when many values fall near zero.

Kendall's *tau-b* value is found in a two-step process. First, the measurements taken from two metrics are ordered according to their values. Second, a calculation is performed which counts the number of values which appear in the same order. The calculation is shown below:

$$\tau_B(F,G) = \frac{concord(F,G) - discord(F,G)}{concord(F,G) + discord(F,G)}$$

Where *F* and *G* are two orderings of values taken from a file pair. *concord(F,G)* is a count of the number of times values appear in the same order. Alternatively, *discord(F,G)* is a count of the number of times values appear in different order. For this test, values of 0 in either *F* or *G* are ignored; that is, they are not counted by either *concord* or *discord*. The value produced falls in range [-1, 1], corresponding to the correlation between the orderings. A value of 1 indicates a perfect linear correlation. For the purpose of this study, and in agreement with [11], we consider values at 0.6 or greater to be strong. Because this is a non-parametric statistical test, we cannot assume a normal distribution fits the data. Therefore, we cannot find an associated *p*-value for each *tau-b* value.

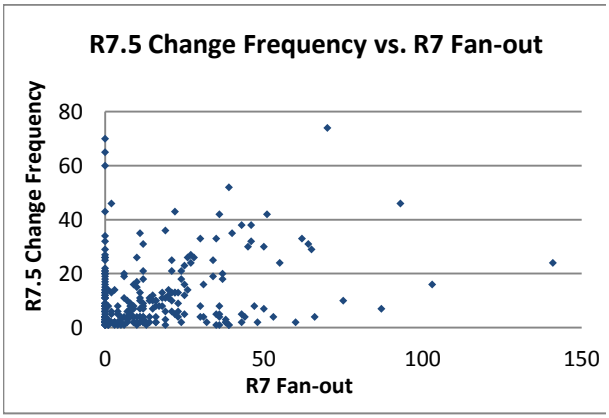

Figure 2: Scatter plot of release 7.5 change frequency and release 7 fan-out. Each data point represents a C/C++ and header pair. The x-axis plots the fan-out of pairs in release 7. The y-axis plots the change frequency of each pair in release 7.5. $R^2 \approx 0.2658$

Table 2: *Tau-b* values for metric pairs

Tau-b table of metrics for svs7 + svs7.5						
R7+R7.5	fan-in	fan-out	file size	changes	tickets	bugs
fan-in	1	0.257	0.301	0.331	0.328	0.464
fan-out	0.257	1	0.441	0.417	0.416	0.637
size	0.301	0.441	1	0.293	0.273	0.510
changes	0.331	0.417	0.293	1	0.972	0.858
tickets	0.328	0.416	0.273	0.972	1	0.857
bugs	0.463	0.637	0.510	0.858	0.857	1

Table 2 shows the *tau-b* value calculated for each metric pair in release 7 and release 7.5. Each cell corresponds to the *tau-b* value as found by the previously described equation. The table is symmetric because the comparison of two ranked metric values is a symmetric property. Highlighted cells indicate a strong correlation.

The highlighted values in the bottom right quadrant of the table are expected correlations. The values report that, for example, as ticket frequency increases, bug change frequency increases as well. This is logically consistent because as developers add more tickets to their commits, more of these tickets will contain bug references. However, the correlation value for bugs vs. fan-out is an unexpected result. This number tells us that as the fan-out of a file pair increases, the number of bugs associated with that pair increases as well. Similar results were found by [13], adding more power to hypothesis that fan-out and number of bugs increase together.

Using these three methods of exploratory data analysis, we identified likely correlations between metrics. In the validation step we analyze these correlations to see if they are indicative of bug-related changes in the future.

5.6 Prediction

Ostrand and Weyuker [10] introduced *accuracy*, *precision*, and *recall* measures from the information retrieval domain. We use various *recall* metrics to validate our prediction of future bugs. Recall is defined as the percentage of faulty files that are correctly identified as faulty files. As in the baseline case study, we calculate recall in three different ways. For every file pair *u,*

Faulty file recall: An instance occurs when either element in *u* is changed at least once in the release representing the future due to any bug ticket.

Fault recall: An instance is a tuple defined as <*u*, bug ticket reference>, where *u* is changed at least once due to the same bug ticket.

Fault impact recall: An instance is a triple defined as <*u*, commit number in the source control logs where *u* is changed, bug ticket reference> where the bug ticket is referenced in the same commit where *u* is changed in.

These three recall measures apply different emphasis to future fault prediction. *Faulty file recall* emphasizes future fault prediction least, because it treats all future bug-related changes to

u, regardless of the number of instances, as one. This fails to capture instances where *u* is associated with more than one bug ticket. However, *Fault recall* does take this into account, because it considers multiple bug ticket references in an instance. Furthermore, *Fault impact recall* provides the highest granularity to allow for future fault prediction because it takes into account all changes *u* goes through. All three recall measures form an implied subsumption hierarchy.

Using these recall measures, we use Alberg diagrams [9] to plot release 7 measurements vs. release 7.5 faults. Alberg diagrams are based on the pareto principle, that roughly 20% of the files in a system are responsible for 80% of the faults. In this context, we use this same principle to estimate the accuracy of prediction models [9].

Figure 3 illustrates one Alberg diagram for this system. The x-axis shows 60 C/C++ source and header pairs, *u*, ordered in descending order according to their metric values from release 7. These 60 file pairs are selected based on their contribution to bug-related changes in release 7.5. The bug change frequency for *u* in release 7.5 is plotted on the y-axis. Any given point on the curve represents a C/C++ source and header pair. The oracle curve is a perfect predictor of release 7.5 bug change frequency for all *u*. As other curves get nearer to the oracle curve, their accuracy for predicting release 7.5 bug change frequency increases.

The oracle curve from this Alberg diagram states that roughly 20% (actually 23.3%) of C/C++ source and header pairs contribute to 80% of bug change frequency in release 7.5. The values of fan-out and change frequency in release 7 for these pairs contributed from 40% to 50% of bug changes in release 7.5. These findings are slightly less than Schwanke et al.'s findings, yet are still noteworthy. This validates that selected metrics from earlier releases can be used to predict bug change frequency in future releases.

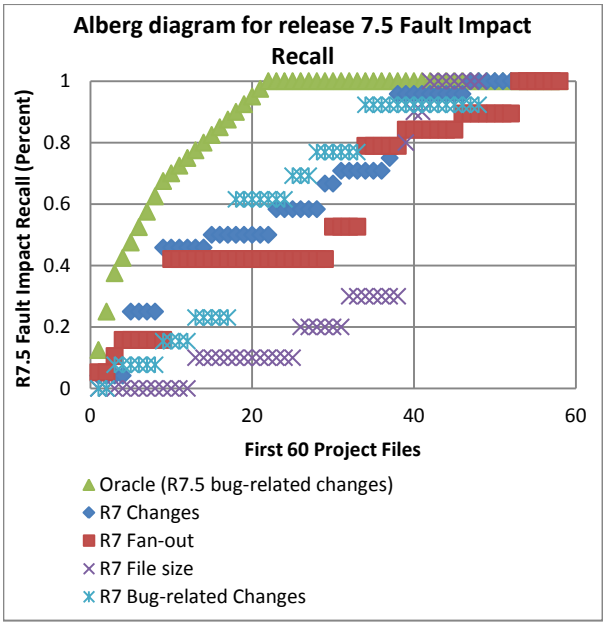

Figure 3: Alberg diagram of release 7.5 Fault Impact Recall. Each data point is a C/C++ source and header pair. The x-axis represents the rank of a data point in release 7, sorted according to its type. The y-axis represents the percentage of bugs in release 7.5.

5.7 Uncovering and Visualizing Architecture Problems

Once these measures have been validated as capable of predicting future faults, the problem of identifying file pairs which are more prone to unexpected changes arises. Next, we study the extent to which these pair affects other quality measures.

We utilized the static code analysis tool *Understand*™ to visualize graphs of interdependent components. *Understand*™ is a commercial product developed by Scientific Tools, Inc.[7]. *Understand*™ can find many structural features of code, including dependency listings of how pairs of C++ files depend on one another. Through visualization, we can analyze the extent to which these dependencies affect other pairs in the software system.

These graphs help differentiate expected and unexpected dependencies. If dependencies occur between two pairs that are in the same module, we treat them as *expected* dependencies, consistent with the baseline study. This is based on the assumption that developers group files or classes together based on similar functionality. *Unexpected* dependencies are treated as dependencies that occur across different modules, also consistent with our baseline study. Our definitions of expected and unexpected dependencies were validated by the developers at Golden Helix.

Because we are concerned with how these dependencies are changing together, we define a "distant" and "local" change pair. Using Schwanke et al.'s [13] definitions, a pair of file pairs that change together, *change pair,* <*u,v*> is local if (1) *u* directly depends on *v*, (2) *v* directly depends on *u*, or (3) *u* and *v* belong to the same module. Any change pair which does not fit under this definition is a *distant change pair*.

Figure 4 illustrates a high level view of the dependencies between modules in SVS. Nodes in the graph represent modules, and edges represent dependencies between modules. The number on the edge refers to the exact number of dependencies. The modules shown contain the ten most frequent distant change pairs. This graph is nearly a complete graph, suggesting that modules have high coupling when distant change pair frequency is high.

Once change pairs have been classified as either local or distant, CLIO is used to (1) identify change pairs which historically have changed together frequently, and (2) cluster these pairs according to the scope of their change pair (local or distant). To identify frequent historic change pairs, we mine the PostgreSQL database built in the procedure described by section 5.1. To cluster the pairs, a "single link" clustering algorithm is used [13].

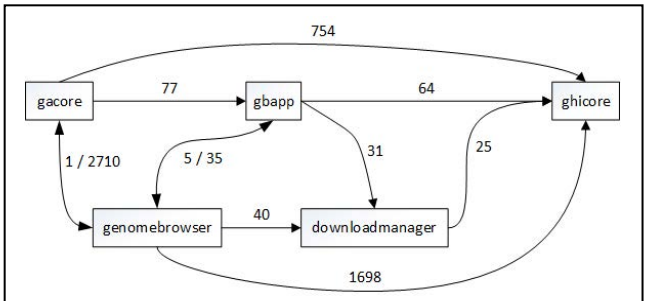

Figure 4: High level view of the dependencies between modules containing the *ten* most frequent distant change pairs. The numbers on the edges represent the number of times all file pairs in a module depend on another module.

The clustering algorithm groups distant change pairs as follows: For each frequent, distant change pair <*u*, *v*>, cluster *u* and *v* together. Then, add all the local dependencies which contain either *u* or *v* to the cluster. We generated visualizations of these clusters that illustrate the number of dependencies across distant change pairs and presented these visualizations to developers.

5.8 Presenting Results to Developers

Visualizing architectural dependencies with graphs provided us with a convenient and intuitive medium that could be validated with developers. We presented all our data to the lead developer at Golden Helix. In summary, the lead developer at Golden Helix was not surprised by our findings. He indicated that several outlier file pairs were contributing to the majority of modularity violations in the code base. It was these pairs that also contributed to a large number of bugs in the most current releases. The lead developer was well aware of this, and more or less the extent to which this affected other files.

The majority of modularity violations and bugs occurred in packages representing highly customizable components of the SVS executable. These packages include the UI component, the core component, and a component that is concerned with reading in a large variety of complex file formats. We noticed that file pairs in these packages both heavily depend on and were depended upon by many others (i.e., they have high efferent and afferent coupling). However, the structure observed was the choice of the developers. The developers utilized these pairs as access points, or common files to reference when one component needed to be used. When these access point pairs were changed, they incurred a slew of changes in other modules in the system because of numerous, propagating dependencies. The developers saw this method as a necessary step in their development lifecycle.

6. DISCUSSION

The process of using CLIO to detect and measure architectural quality of software needs to be matured further. Developers were not surprised by the findings of CLIO, primarily because the findings pointed out known problems. Many of these problems are due to the many connections that exist between modules. From an academic sense this is a problem, because it is preferable to have few connection points between modules (coupling). Lower coupling between modules is indicative of better design, and helps localize possible future changes as well as allows for increased quality attributes (such as understandability) [2]. However, from the developers' perspective, familiarity with the code base was more important than traditional good design. The developers are content leaving the coupling between modules as is, because it makes the most sense for the SVS system. This finding is very interesting because it gives the impression that the results from tools such as CLIO should be system dependent. That is, although the results may appear useful, nothing can be learned unless an in-depth assessment of the software system in question has been made. These conclusions cannot be reached without evaluating and deploying laboratory tools in commercial grade environments.

We did find very similar results to the baseline, which is promising in helping extend power of the hypothesis that certain metrics can be used as better predictors of software quality. We found that a select few files contributed to many modularity violations, and greatly influenced the number of bugs. While in our case the developers were not surprised by the results, the results are promising in that they clearly identify problem files in code. The baseline found that developers were not always aware of these modularity violations. In cases where developers may not be fully familiar with the structural connections across modules in their code base, this procedure provided significant insights.

We also identified and validated cases where structural metrics can be used as quality predictors for future releases. Both this study and Schwanke et al. [6] concur that the fan-out metric is a good predictor of future faults, as verified by correlation analysis and Alberg diagrams.

7. THREATS TO VALIDITY

There are several threats that threaten the validity of this study. One developer brought up the argument that, "If a developer prefers to commit files more frequently than other developers, it would show up in the commit logs as having few change pairs. This would give misleading results because it would provide cases where too few files are being committed to account for changes across modules, or too many files are being committed which would make it appear that more dependencies exist." This is a direct threat to the construct validity of our study. Although the developer's observation is correct, it did not have a large impact on our results. We identify files showing up in the commit logs together with a high frequency, and ignore cases where paired changes happen infrequently. This reinforces that such cases as described by the developer are unlikely to occur often. Regardless, the observation does shed light into a situation that will be mitigated in future studies.

A second threat to the construct validity is the fact that we grouped C/C++ source file and corresponding header files together. These file pairs consist of the aggregated information from their combined elements. Although a threat, it is mitigated by the following reason. The developers brought to our attention that both elements in the file pair are expected to belong to the same package, and are expected to change together. That is, if a C++ source file is updated, the developers expect to make changes to the signature of the header source file as well. Because both of these cases are expected changes, including both files separately in the study would be spurious information. Thus, we chose to group every C++ source and corresponding header file together.

A third threat to the construct validity of this study is the assumption that developers tag bugs correctly in the commit messages. As an external observer, the only method we have of identifying past-bugs in the software project is through analyzing historical artifacts. Therefore, we need to rely on the discipline of developers to (1) tag the bugs they focused on in a commit and (2) tag the bugs correctly. We have no way of knowing if either of these two conditions is not met.

External validity represents the ability to generalize from the results of a study. In this instance, we cannot generalize the results we found to other contexts. In other words, the results found in this study and the baseline only hold true for our specific contexts, however they helped in building consensus around our findings across different programming languages in commercial agile development environments. More replication studies are necessary to increase the power of these results.

8. CONCLUSION

This replication case study was performed to help us analyze how structural file metrics could be correlated with system quality, and to help us comprehend if similar observations performed in a Java commercial product could also be observed in its C++ counterpart. We have gathered structural metrics and identified correlations between them and future bug problems. We identified a select few outlier files which contribute to the majority of future bug problems. From these, we collected dependencies and

visualized how extensively problems may propagate. We showed this information to the developers of Golden Helix and they were not surprised by the results. Rather than attempt to entirely eliminate distant-modules with frequently-changing dependencies, the developers preferred to keep a select-few files as connection points. When asked why, the lead developer explained that these connection points offer a single point of entry into a module. Any changes between modules would be reflected in the connection points only. The developers would rather be aware of a few files that are frequently problematic than issue a refactoring.

9. Challenges

Herein we describe some of the challenges we encountered while trying to perform this study.

1) Specific Tools: The baseline study featured the use of the commercial tool *Understand*™ for static analysis of code to gather metrics as well as to visualize results. Although the static analysis and visualizations provided high quality analysis, it is nearly impossible to replicate this case study without the use of this specific tool. Alternatives were considered, but the mechanistic formula used for analyzing files needed to be used as is, as other approaches would have constituted (in the opinion of the authors) a significantly large deviation from the baseline method that we would not have been able to call this a replication study.

2) Understanding the System: While we hope that manually performing the CLIO process eventually leads to an automated approach, this study suggests that such a hope may be far-fetched. Ultimately, a complete understanding of the system in question is necessary before any significant value can be taken from this tool. Our results mean very little unless the developers actually make use of them.

3) Literature Coverage: The majority (entirety) of literature covering replications in Empirical Software Engineering refers to formal experiments, not case studies. We have borrowed the terminology from such literature in this study. This situation is not ideal because case studies have less power than formal experiments and therefore should be approached differently. Peer-reviewed literature needs to be published which outlines case study replication guidelines.

10. ACKNOWLEDGMENTS

We would like to thank Golden Helix for allowing us access to their software and providing us with the necessary resources to carry out this study. We would especially like to extend our gratitude to Gabe Rudy for his generosity and devotion to this project.

11. REFERENCES

[1] Baldwin, C. and Clark, K. 2000. *Design Rules: The power of Modularity*. Vol. 1. MIT Press., Cambridge, MA.

[2] Bansiya, J. and Davis, C. G. 2002. A hierarchical model for object oriented design quality assessment. In *IEEE Transactions on Software Engineering* 28, 1 (Aug. 2002), 4-17. DOI=http://dx.doi.org/10.1109/32.979986.

[3] Basili, V. R., Selby, R. W., and Hutchens, D. H. 1986. Experimentation in Software Engineering. In *IEEE Transactions on Software Engineering* 12,7 (July 1986), 733-743. DOI=http://dx.doi.org/10.1109/TSE.1986.6312975.

[4] Brooks, A., Roper, M., Wood, M., Daly, J., and Miller, J. 2008. Replication's Role in Software Engineering. In *Guide to Advanced Emprirical Software Engineering*, Shull, F., Singer, J., and Sjøberg, D. I. K. Springer London, Springer, 365-379. DOI=http://dx.doi.org/10.1007/978-1-84800-044-5_14.

[5] Cunningham, W. 1992. The Wycash portfolio management system. In *OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications* (Dec. 1992). OOPSLA '92. SIGPLAN ACM, New York, NY 29-30. DOI= http://dx.doi.org/10.1145/157709.157715.

[6] Izurieta, C. and Bieman, J. 2013. A multiple case study of design pattern decay, grime, and rot in evolving software systems. In *Software Quality Journal*, 21, 2 (June 2013), 289-323, DOI=http://dx.doi.org/10.1007/s11219-012-9175-x.

[7] Juristo, N. and Moreno, A. M. 2010. *Basics of Software Engineering Experimentation* (1st ed.). Springer Publishing Company, Incorporated.

[8] Kendall, M. G. 1938. A new measure of rank correlation. In *Biometrika*, 30 (1938), 81-93.

[9] Ohlsson, N. and Alberg, H. 1996. Predicting fault-prone software modules in telephone switches. In *IEEE Transactions on Software Engineering*, 22, 12 (Dec. 1996), 886-894, DOI= http://dx.doi.org/10.1109/32.553637.

[10] Ostrand, T. J. and Weyuker, E. J. 2007. How to measure success of fault prediction models. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting* (SOQUA '07). ACM, New York, NY, USA, 25-30. DOI= http://doi.acm.org/10.1145/1295074.1295080.

[11] Ott, R. and Longnecker, M.1993. *An introduction to statistical methods and data analysis*. Vol. 4. Duxbury Press, Belmont, CA.

[12] Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (December 1972), 1053-1058.

[13] Schwanke, R., Xiao, L., and Cai, Y. 2013. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software Engineering (ICSE)* (San Francisco, CA, May18 - 26 2013). ICSE '13. IEEE, San Francisco, CA, 891-900. DOI= http://dx.doi.org/10.1109/ICSE.2013.6606638.

[14] Shull, F. J., Carver, J. C., Vegas, S., and Juristo, N. 2008. The role of replications in Empirical Software Engineering. In *Empirical Software Engineering* 13, 2 (April 2008), 211-218. DOI= http://dx.doi.org/ 10.1007/s10664-008-9060-1.

[15] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. 2012 *Experimentation in software Engineering*. Springer Berlin Heidelberg. DOI= http://dx.doi.org/10.1007/978-3-642-29044-2.

[16] Wong, S., Cai, Y., Kim, M., and Dalton, M., 2011. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, New York, NY, USA, 411-420. DOI= http://doi.acm.org/10.1145/1985793.1985850.

[17] Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seamon, C., and Shull, F. 2013. Comparing four approaches for technical debt identification. In *Software Quality Journal* (April 2013), 1-24, Springer US.