

Design Patterns and Change Proneness: An Examination of Five Evolving Systems.

James M. Bieman*, Greg Straw*, Huxia Wang*, P. Willard Munger**, Roger T. Alexander*

*Software Assurance Laboratory
Computer Science Department
Colorado State University
Fort Collins, CO 80523 USA

{bieman,straw,wanghu,rta}@cs.colostate.edu

**School of Computing
Southern Adventist University
Collegedale, TN 37315 USA
wmunger@southern.edu

Abstract

Design patterns are recognized, named solutions to common design problems. The use of the most commonly referenced design patterns should promote adaptable and reusable program code. When a system evolves, changes to code involving a design pattern should, in theory, consist of creating new concrete classes that are extensions or subclasses of previously existing classes. Changes should not, in theory, involve direct modifications to the classes in prior versions that play roles in a design patterns. We studied five systems, three proprietary systems and two open source systems, to identify the observable effects of the use of design patterns in early versions on changes that occur as the systems evolve. In four of the five systems, pattern classes are more rather than less change prone. Pattern classes in one of the systems were less change prone. These results held up after normalizing for the effect of class size — larger classes are more change prone in two of the five systems. These results provide insight into how design patterns are actually used, and should help us to learn to develop software designs that are more easily adapted.

Keywords: Design patterns, change proneness, software evolution, adaptability, maintainability, case study, object-oriented methods, size measurement.

1. Introduction

Methods to assess object-oriented (OO) design quality can help developers choose between alternative designs. We are interested in raising the level of abstraction of OO design measures to include *architectural context*, the role that a program unit plays within a larger design structure. For

example, a class may play a role in a design pattern and/or an inheritance hierarchy. This role represents its architectural context.

Design patterns promote reuse of solutions to recurring design problems by naming and cataloging these solutions. Many design patterns, for example, those described in the popular Gamma et al. [7] book, promote adaptability, by supporting modifications through specialization. Developers can adapt a system built using these patterns by creating new concrete classes with desired functionality rather than by direct modifications to existing classes.

There are numerous publications describing design patterns, including references on patterns for designing software architectures [3, 19], patterns for describing problem analysis [6], patterns for assigning operations to classes [13], etc. The Patterns Home page has links to pattern definitions, conferences, and tutorials [18].

This work primarily examines the use of the patterns described in Gamma et al. [7], because these are the most commonly referenced patterns, and are most likely to be applied in the development of existing systems. Gamma et al. describe 23 patterns that solve common software design problems. In particular, they describe through examples and discussions how these patterns support adaptability. They do not demonstrate the benefits to real software development projects. The Gamma et al. patterns were supplemented by those defined by Grand [8] for the four case studies implemented in Java.

A specific objective is to see how design patterns are applied to real software development projects. We want to see if patterns are used in a manner that is consistent with the expectations of the pattern references. In particular, we want to determine if software with patterns tends to be adapted by creating new concrete classes that are extensions of existing pattern classes, interfaces, or abstract classes as

we would expect, or by modifying existing pattern classes.

In a prior study [2], we analyzed 39 versions of an evolving industrial system implemented in C++ to see if there was a relationship between patterns, other design attributes, and the number of changes. We found a strong relationship between class size and the number of changes — larger classes were changed more frequently. We also found two relationships that we did not expect: (1) classes that participate in design patterns were not less change prone — these pattern classes are among the most change prone in the system, and (2) classes that were reused the most through inheritance tend to be more change prone. These unexpected results held up after accounting for class size, which had the strongest relationship with changes.

The prior study represented only one software project. This paper partially replicates the prior study by examining four additional systems: two industrial systems and two open source systems. Each of these additional systems was implemented using Java.

2. Overview of Study Design

We examine five evolving systems and look at the relationship between design structure and software changes. Design structure is characterized by class-size, and class participation in inheritance relationships and design patterns. Changes is measured in terms of a count of the number of times that a class is modified over a period of time. We quantify the design structure of an early version of each system, and study the relationship between the design attributes of this version and future system changes.

2.1. Systems Under Study

Table 1 provides high-level information about the systems that we studied. For each system, we examined the design structure in detail of a base system — an early version of the system. We extracted the object models of an early versions of each system using the Together tool from TogetherSoft. Together generated code metrics; we identified patterns by analyzing the object models. Then we examined changes as the system evolved through several later versions.

2.2. Initial Commercial C++ System

The initial study was conducted on a commercial OO system implemented in C++. This development project took place while the organization was in the process of adopting OO methods. The system was developed with the support of a version control system over a period of several years. Our focus has been the transformations between two specific versions of the system: version A, which is the first

Table 1. System Level Measurements.

| System | Version | Num. of Classes | Lines of Code |
|--------------------------|---------|-----------------|---------------|
| C++ Commercial System | A | 199 | ~24,000 |
| | B | 227 | ~32,000 |
| Commercial Java System A | 2 | 384 | ~23,000 |
| | 18 | 404 | ~23,000 |
| Commercial Java System B | 2 | 101 | ~7,500 |
| | 18 | 201 | ~17,000 |
| Netbeans | 1.0.x | 4,138 | ~327,000 |
| | 3.3.1 | 7,573 | ~753,000 |
| JRefactory | 2.6.24 | 546 | ~43,000 |
| | 2.6.38 | 699 | ~47,000 |

stable version of the system, and version B, which is the final version in our data set. We studied a total of 39 versions of the system including versions A and B, and all of the intermediate versions.

2.3. Commercial Java Systems

The commercial Java systems were analyzed in a similar manner to that of the initial C++ system studied. Both are real-time embedded system controllers.

System A. System A is an early development in Java by the organization. Version 2, the base version for design analysis, contains 384 classes and around 23,000 lines of code. We examined 17 versions of System A.

System B. System B is a more mature Java development project. Version 2, the base system for analysis, contains 101 classes and around 7,500 lines of code. We examined 17 versions of System B.

2.4. Open Source Systems

The open source systems were collected via the internet. System versions are managed for these systems using the Concurrent Version System (CVS), an open source version management system. We tracked changes using the diff program.

Netbeans. Netbeans is an open source software development environment for Java program development; it was developed and is maintained by Sun Microsystems. We classify it as a hybrid open source system. It was developed by a commercial organization, but the entire source is available as open source code at Netbeans.org. Netbeans is the largest system that we studied; it contains more than 7,500 classes. We used Netbeans version 1.0.x as the base version, and studied the changes through versions 3.2.x, 2.0, 3.0, 3.1, 3.3 and 3.3.1.

JRefactory. JRefactory is an open source system designed to refactor or restructure Java programs. It was developed and maintained by the Open Source community and is available through SourceForge.net. Version 2.6.38 of the system contains 699 classes and more than 47,000 lines of program code. We used JRefactory release 2.6.24 as the base version and studied the changes through release 2.6.27, 2.6.30, 2.6.31, 2.6.32, 2.6.34, 2.6.35, and 2.6.38. These were all the publicly released versions available at the time of the study.

2.5. Case Study Hypotheses

Our major objective is to test whether the architectural design context of a class can predict future changes to a class. We want to demonstrate that the architectural design context affects class change proneness after accounting for the effect of single class properties such as class size. Thus we also need to examine the relationship between single class properties and change proneness. Here, we concentrate on class size.

We tested the following hypotheses on the five case studies:

- H1: Larger classes will be more change prone. A larger class has more functionality, thus there is a greater likelihood that some functionality in the class will need to be corrected or enhanced.
- H2: Classes participating in design patterns are less change prone. Patterns are designed so that changes are made via subclasses or by adding new participant classes rather than modifying already present classes. Patterns promote ease of change, hence the classes participating in patterns should require fewer changes.

The initial case study of the C++ system also examined whether or not classes that are reused through inheritance more often tend to be less change prone.

We tested these hypothesis by analyzing the relationship between measurements both class design context and size class properties of early versions of each system and a count of the number of changes to each class that occurred during the transition from the early version to a later version.

2.6. Determining Size

In the initial study, we used the number of operations and the number of attributes as class size measures. Because these two size measures were closely correlated, and we found that the number of operations to be a better predictor of changes, we used only the number of operations in the other case studies. We used the number of operations rather than lines of code, because we wanted the analysis to be at the design-level rather than at code-level.

2.7. Identifying Intentional Patterns

Although, in the worst case, finding patterns in object models is intractable, several researchers show that patterns can be identified quickly. For example, systems by Kramer and Prechelt [11], Antoniol et al [1], and Keller et al [9] demonstrate the feasibility of finding patterns in automated design pattern recognition.

A manual approach to finding patterns is an alternative to automated pattern recognition. For example, Shull, Melo and Basili [20] use an inductive approach to identify custom patterns in domain-specific systems.

In this research, we are looking for intentional patterns, patterns that developers use in a deliberate, purposeful manner. These patterns should be documented, and they should have a effect on the number of changes, since adaptability is the primary reason for using patterns — the indirection inherent in design patterns should reduce the number of changes to existing classes. Changes should be limited to adding new subclasses or other new classes that were not part of the original pattern. Because we seek to find only intentional patterns, we adopted a manual approach for pattern recognition with the following steps:

1. Search for pattern names in the documentation of the system. Developers are likely to document the pattern functionality/role of the class or method so that a pattern can be treated as a pattern during later development or maintenance.
2. Identify the context of the classes identified in step 1 by analyzing the object models. Once we find the classes whose documentation specifies something relating to a pattern name/role, we can look at the object models to identify all the classes required to constitute a pattern. We look for the links between classes that implement the pattern.
3. Verify that the candidate pattern is really a pattern instance. We examine the pattern implementation to look for lower level details, for example, required delegation constructs.
4. Verify the purpose of the pattern. We examine each group of classes that represent a pattern candidate to confirm that the classes and relations have the same purpose as described by an authoritative pattern reference. We use the Gamma et al. [7] and Grand [8] books as the authoritative references for this study.

2.8. Determining Changes

In the initial study, we used the version control system to obtain multiple versions of the system and collect data on the transformations between 39 versions.

For the two commercial Java systems (Java Systems A and B) we examined the structure of the first substantially complete version of each system (version two of each system) and tracked changes over 17 subsequent versions. We tracked changes for seven versions of Netbeans and eight versions of JRefractory.

In the study of the initial system, we tracked changes as recorded by the version control system. We used diff to track changes in the remaining systems. We count the number of changes to each class that occur in the transitions from the first to the last version. Changes can be corrective, adaptive, perfective, or preventive. Design patterns should aid in the last three types of maintenance. As is the case with many industrial systems, the systems under study had no maintenance history other than the comments in the code, and the recollection of the few system developers that we could find (for the commercial systems). An initial analysis of different classes of changes did not show any differences between the change type. In this work, we do not classify the types of changes performed on the classes.

3. Results

Table 2 lists the patterns identified in each system and number of instances of each pattern; In the original case study C++ system, 18 classes play roles in 16 pattern instances of four pattern types — Singleton, Factory method, Proxy and Iterator patterns.

In System A, 49 classes out of a total of 384 classes played roles in six pattern instances of five design pattern types — Adapter, Factory Method, Singleton, State, and Strategy patterns. In System B, 38 of the 102 classes played roles in nine pattern instances of four pattern types — Builder, Factory Method, Singleton, and Strategy patterns. In Netbeans, we found 141 pattern classes out of a total of 4138 classes played roles in 42 pattern instances of seven pattern types — Adapter, Builder, Command, Creator, Factory Method, Iterator, and Singleton patterns.

We found 176 pattern classes in JRefractory out of a total of 699 classes that played roles in 26 pattern instances of six pattern types — Adapter, Builder, Filter, Singleton, State, Visitor patterns. Now we examine each hypothesis.

3.1. H1: Are larger classes more change prone?

In the first case study, we found that size, as measured by the number of operations and number of attributes correlates to the number of changes. The impact of the total number of operations is much greater than that of attributes. Class size had a much smaller relationship with changes in three of the four other case studies. We also found that the number of operations correlates closely with the number of lines of program code.

Table 2. Patterns Identified in the Base Version of each System. C++Sys denotes the original case study system; SysA is Java System A; SysB is Java System B. All patterns are from Gamma et al. [7], except for Filter which is from Grand [8].

| Pattern | Number of Instances | | | | |
|----------------|---------------------|------|------|----------|-------------|
| | C++Sys | SysA | SysB | Netbeans | jRefractory |
| Adaptor | | 1 | | 1 | 16 |
| Builder | | | 1 | 4 | 2 |
| Command | | | | 8 | |
| Creator | | | | 1 | |
| Factory Method | 1 | 1 | 4 | 18 | |
| Filter | | | | | 2 |
| Iterator | 4 | | | 1 | |
| Proxy | 1 | | | | |
| Singleton | 10 | 1 | 3 | 2 | 1 |
| State | | 2 | | | 3 |
| Strategy | | 1 | 1 | | |
| Visitor | | | | | 2 |

The number of operations in a class is the size metric with the strongest significant relationship with change proneness. Figure 1 displays a scattergram and a fitted line plot based on a regression analysis for the initial C++ system, which shows that classes with more operations tend to require more changes. The significance of this relationship is indicated by low α -values from the correlation analyses. In the first case study, we can reject the null hypothesis for H1, and accept H1. In the C++ system, larger classes, measured by the number of operations, are more change prone.

Figures 2, 3, 4, and 5 show the relationship between the number of operations and changes for classes in Systems A, B, Netbeans, and JRefractory. These scattergrams are displayed in a log-log scale to better show the distribution of values. Of these four systems, only Netbeans (see Figure 4) shows a strong relationship between class size and changes. Systems A, B, and JRefractory have little or no relationship between class size and changes. Table 3 shows that the fairly strong relationship between class size and changes that we found in the first case study is repeated only in Netbeans. Note that the Spearman rank correlation is the strongest (.60) for Netbeans, while the Pearson correlation is strongest (0.839) for the C++ system. Since our data is not normally distributed, the Spearman correlation is the most meaningful. To summarize, results from three of the four additional case studies conflict with the strong relationship found in the first case study. We now do not have enough evidence to reject the null hypothesis, and we cannot conclude that class size can predict the number of changes.

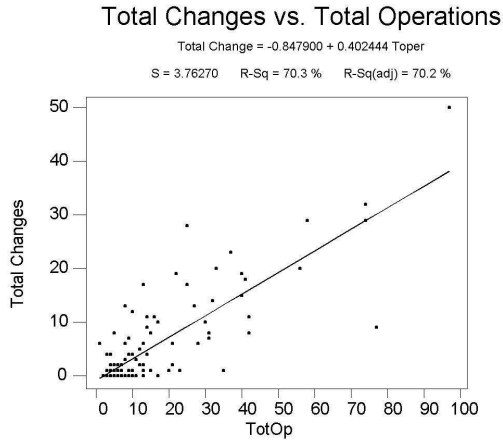


Figure 1. Initial C++ case study system: Scattergram of the number of operations per class (TotOp) versus the total number of changes per class with regression analysis results and fitted line plot.

Table 3. Correlation coefficients for the number of operations per class as a predictor of the number of changes.

| System | Pearson Correlation | | Spearman Rank Correlation | |
|-------------|---------------------|-----------------|---------------------------|-----------------|
| | Co-efficient | α -Value | Co-efficient | α -Value |
| C++ System | 0.839 | <.0001 | 0.50 | <.0001 |
| Java Sys. A | 0.26 | 0.0001 | 0.33 | <.0001 |
| Java Sys. B | 0.45 | <.0001 | 0.09 | 0.3161 |
| Netbeans | 0.40 | <.0001 | 0.60 | <.0001 |
| Jrefactory | 0.351 | <.0001 | 0.281 | <.0001 |

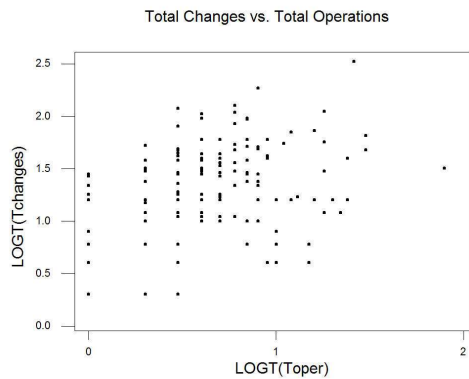


Figure 2. Java System A: the number of operations vs. changes per class (log-log scale).

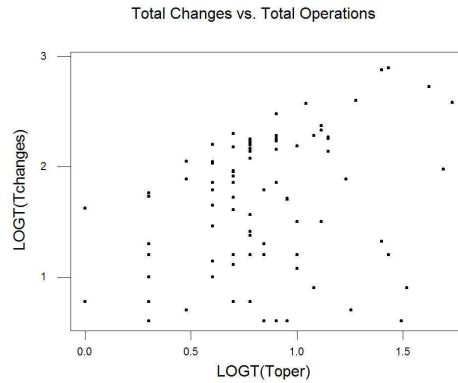


Figure 3. Java System B: number of operations vs. changes per class (log-log scale).

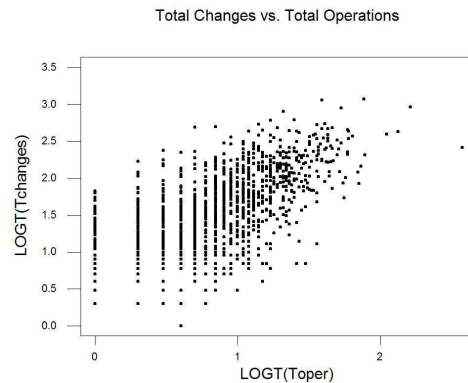


Figure 4. Netbeans: number of operations vs. changes per class (log-log scale).

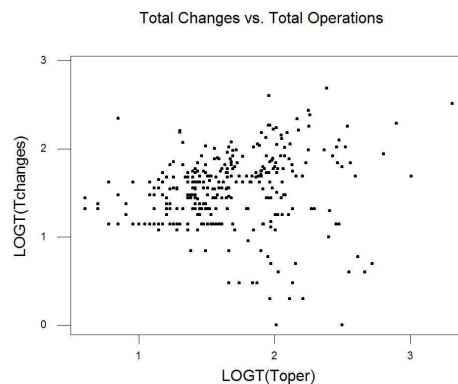


Figure 5. JRefactory: number of operations vs. changes per class (log-log scale).

3.2. H2: Are pattern classes less change prone?

H2 and the initial C++ system case study. Figure 6 clearly shows that most of the classes that do not participate in a pattern require very few changes — 75% of the non-pattern classes are changed at most once, while classes that take part in patterns tend to require comparably many more changes.

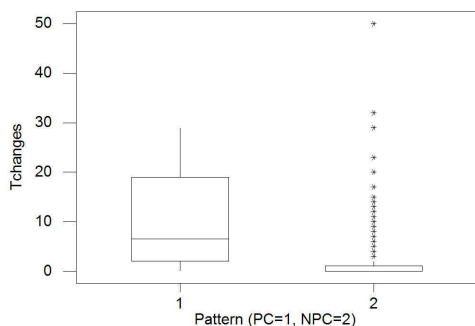


Figure 6. Initial C++ case study system: Box plots of the distribution of Changes for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPC = 2).

Although pattern classes in this system appear to be more change prone, we need to see whether this is a result of a third factor. In the C++ system, larger classes — classes with more operations — are more change prone, and we found that pattern classes are larger. To control for the effect of class size, we used change density rather than the total number of changes in our analysis. Change density is the changes per operation (Changes/TotOp). Figure 7 shows that the difference in change density between pattern classes and non-pattern classes is less than when we used the total number of changes. However, pattern classes still show more changes per operation than non-pattern classes.

We clearly cannot reject the null hypothesis for H2; the pattern classes are more change prone than non-pattern classes. We reverse the hypothesis to see if there is support for the null hypothesis:

–H2: Classes participating in design patterns are **more** change prone.

First we analyze the data to determine the appropriate statistical methods. We determined that the data is not normally distributed, and we were unable to find a better fit

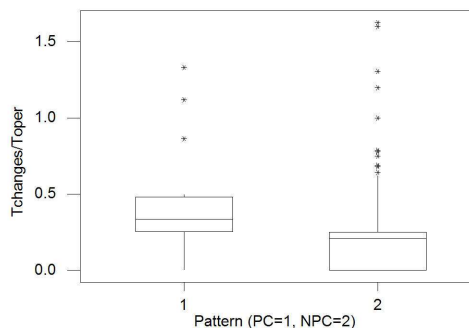


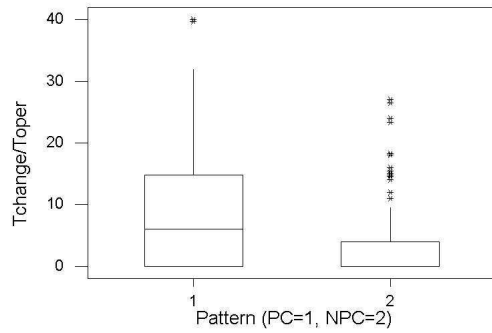
Figure 7. Initial C++ case study system: Box plots of the distribution of change density for pattern classes (PP = 1) versus non-pattern classes (NPC = 2). One outlier was not included in the figure.

using a logarithmic transformation. Thus we applied non-parametric methods to test the hypothesis.

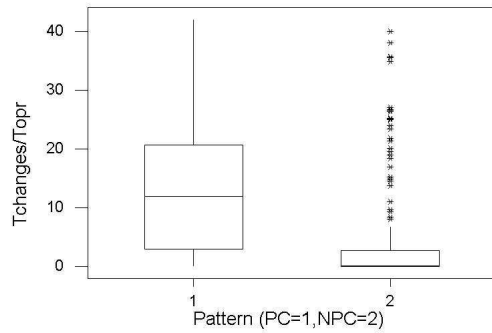
We applied the Mann-Whitney test, a non-parametric two sample rank test of the equality of two population medians, and the corresponding point estimate and confidence interval [14]. This test allows us to reject the null hypothesis of $\neg H2$, our original H2, with a significance of 0.0003. We conclude that classes that participate in design patterns are **more** change prone.

Additional Case Study Results. We evaluated the change proneness of the pattern and non-pattern classes in the additional case study data. Although these changes were only minimally related to class size in these systems, we report the results here in terms of changes per operation to be consistent with our results from the initial case study. An analysis without normalizing for class size does not change the results significantly.

The results were similar to our initial findings in three of the four systems. Pattern classes in System A, System B, and Netbeans were more change prone, as shown in Figures 8.A, 8.B, and 9. In contrast, JRefractory pattern classes were less change prone than non-pattern classes as shown in Figure 10. All of these results were significant at the 0.0001 level. We also obtained similar results when we examined the relationship between pattern roles and the total number of lines of code changed rather than a count of changes. That is, the results for pattern roles versus change size matched the results for a count of changes.



A. System A Results



B. System B Results

Figure 8. Java Systems A and B: Box plots of the distribution of change density as measured by Changes/TotOpp for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPC = 2). Fig. 8.A is the data for System A and Fig. 8.B is System B data.

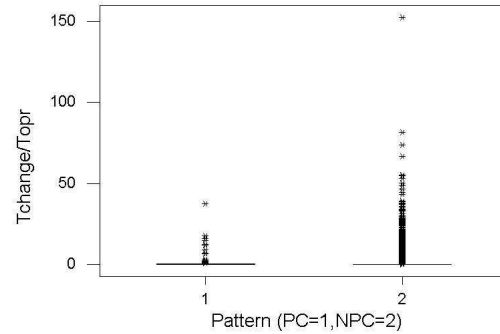


Figure 9. Netbeans: Box plots of the distribution of change density as measured by Changes/TotOpp for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPC = 2).

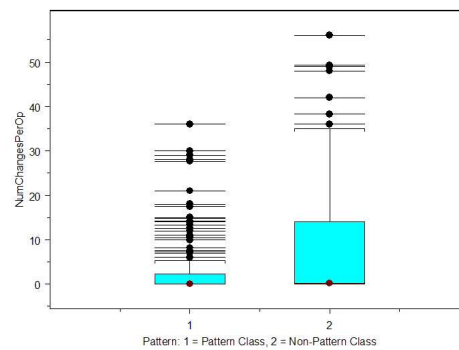


Figure 10. JRefractory: Box plots of the distribution of change density as measured by Changes/TotOpp for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPC = 2).

3.3. Are classes that are reused through inheritance more often less change prone?

In the initial case study, we examined the relationship between classes with more children or descendents via inheritance and changes. Our hypothesis was that classes with more and/or more descendents would be less change-prone. Our rationale was that classes with many subclasses would cause a large ripple effect if they were changed — the subclasses would be effectively changed. The results in this prior study were in conflict with the hypothesis. Classes with more children or more descendents were changed more, not less often. However, the Spearman Rank Correlation showed that the strength and/or significance of these relationships was not strong. Classes that were reused more through inheritance were not less change prone. We found no additional evidence in the new case study data to reevaluate this issue.

4. Discussion

The new case studies demonstrate that we should not rely upon the results from a single case study. The first case study made it appear that (1) larger classes are more change-prone, an expected result, and (2) classes that play roles in design patterns are more change prone, an unexpected result.

The new case studies indicate a relationship between class size and change-proneness, but it is a much weaker relationship than in the first study. We can speculate on reasons for the differences. First, the original case study was implemented in C++, while all of the others are Java systems. Perhaps, there is something about larger C++ classes — classes with many operations — that make them more change-prone than larger Java programs. Another possible explanation is that the C++ system was one of the first OO development projects in the organization. The developers may not have been making the best use of object technology.

The result that pattern classes are more change-prone from the first case study was supported by three of the four additional studies. The exception was JRefactory, which was the only system that is being maintained by the Open Source community. (The source for Netbeans is publicly available, but the system is maintained by Sun Microsystems.) Is that the reason for the difference? We need to study additional “pure open source” systems to find out.

To answer these questions, we also need more semantically meaningful ways to evaluate design quality. One area that we are exploring is the notion of coding concerns — anomalies in the code that may represent potential trouble spots, either defects or difficult to understand program constructs [16]. Such concerns can be detected by commercial

tools. We have begun to examine the concerns in the Java case study code. All of the systems contain many concerns. A further analysis may provide an explanation for the differences in the case study systems.

Another area to explore is the nature of the changes, and their root causes. Further analysis of the changes themselves will require both tools and insight. Determining the root causes of program changes requires additional data, such as change requests and change logs, which are often very difficult to obtain.

Clearly, there are factors that we did not include in the study that affect change-proneness. Our study was, however, designed to identify specific relationships: class size and change-proneness, and class pattern roles and change-proneness.

5. Threats to Validity

An adequate study should be valid for the population of interest [22]. We assess four types of threats to the validity of this empirical study: construct validity, content validity, internal validity and external validity. Construct validity refers to the meaningfulness of measurements [10, 17] — do the measures actually quantify what we want them to? To validate the meaningfulness of measurements, we need to show that the measurements are consistent with an empirical relation system, which is an intuitive ordering of entities in terms of the attribute of interest [5, 12, 15]. The dependent variable in this study, a count of changes is an intuitive measure of an aspect of maintenance effort. However, not all changes are equal, but a large number of changes over a many versions should minimize the impact of change effort variability. Further study can determine the distribution of effort per changes; actual change effort data was not available for this study.

Content validity refers to the “representativeness or sampling adequacy of the content... of a measuring instrument” [10]. The content validity of this research depends on whether the individual measures of design structures and maintainability adequately cover the notion of design quality and maintainability respectively. The count of changes quantifies only one aspect of maintenance effort in our empirical study. We only look at one structural attribute, size, as quantified by the number of operations. The development of measures that completely capture the design quality of object-oriented software is an ongoing research activity.

Internal validity focuses on cause and effect relationships. The notion of one thing leading to another is applicable here and causality is critical to internal validity. The statistical results show that design structures like operations, number of descendants, pattern use, etc. are related to changes in the system. Such statistical results only provide empirical evidence, they do not account for causality.

Demonstrating causality requires more than illustrating statistically significant relationships. We need to show temporal precedence — evidence that cause precedes effect, and demonstrate a theory that defines a mechanism for the relationships [4, 21]. In our study the design measures were collected from software developed before the change activity, and there are causal explanations for the effect on changes, which were expressed as hypothesis. The relationship between class size and change-proneness is not strong enough to allow us to reject the null hypothesis for H1. Our analysis rejects hypothesis H2 and shows strong support for the null hypotheses. Before we accept this null hypothesis, we need to demonstrate internal validity for the relationship. We will need a good theory and additional data to support them. Thus, we are conducting more in-depth studies of these systems to identify if possible the real connection between class roles and changes.

External validity refers to how well the study results can be generalized beyond the study data. The initial C++ study was based on a single system, undocumented patterns in the system are not identified and developers might have had inadequate knowledge about design patterns. This system represents an industrial development project during the developers transition to OO methods.

We have added four additional systems implemented in Java, rather than C++. Two of the systems were developed by engineers in the same company as the first C++ system. The remaining systems were open source: one was developed and is maintained by Sun Microsystems; the other is a pure open source system. Thus, we have examined systems that represent three developments in one company, a development in a second, and one pure open source system. Virtually all case studies exhibit weak external validity. Additional case studies conducted in different environments are necessary to determine the generality of the relationships. The added case studies demonstrated that the results concerning class size and change-proneness from the first case study do not appear to generalize. The added case studies do provide further support for the relationship between class pattern roles and change-proneness. However, the one exception, the JRefactory case study, shows that the results do not generalize to all software systems.

6. Conclusions

The goal of this research was to use case studies, both from industry and from the Open Source community, to explore the relationship between design structures in object-oriented software and development and maintenance changes. The study included several design factors that can potentially affect maintenance efforts. We investigate the general notion of design structure, and examine the relationship between class size, inheritance, design pattern

use and changes. Our main results are as follows:

1. The relationship between class size and number of changes is equivocal. The results from the first case study of a C++ system and the study of Netbeans indicate that class size can predict the number of changes. In those systems, classes with a greater number of operations were changed the most. The relationship between class size and changes was minimal at best in the three other systems studied. Thus, we do not have strong support for the common belief that component size is a dominant factor in predicting change effort.
2. Classes that play roles in design patterns are changed more often than other classes in four of the five case studies. The case study data does not show that design patterns support adaptability. An informal analysis suggests that pattern participant classes provide key functionality to the system, which may explain why these classes tend to be modified relatively often. However, the rationale given for using design patterns is that changes to a design pattern will be made by adding new concrete classes to a pattern rather than by modifying existing pattern classes.

These results are from five case studies, which represent five development environments, applications, and groups of developers. Further studies are needed to compare results in different domains.

Perhaps the most valuable result from this work is the clear demonstration of the dangers of a generalization based on one case study. The results from three of the additional four case study did not match the first case study's finding on the relationship between class size and class fault-proneness. In addition we found one system, JRefactory, where the relationship between class pattern roles and change-proneness was quite different from that in the other systems.

Our results do show that commonly held beliefs about either class size or class pattern roles, and change-proneness are not clearly supported by the case study data. We are now working to identify key reasons for the counter-intuitive results. One key objective will to determine why pattern classes in JRefactory are less change prone, which contradicts the results from the other case studies.

Future work should provide evidence to guide developers to use design structures in a manner to increase the adaptability of systems. We also plan to investigate the change-proneness of classes in non-intentional patterns — patterns that cannot be identified by pattern names and documentation; this work will involve automated design pattern recognition technology.

Acknowledgements

This material is based on work supported by the U.S. National Science Foundation under grant CCR-0098202, and by a grant from the Colorado Advanced Software Institute (CASI). CASI is sponsored in part by the Colorado Commission on Higher Education (CCHE), an agency of the State of Colorado. Storage Technology Corporation provided software, tools, and computer resources for this study. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The Statistics Dept. of Colorado State University providing valuable help with our statistical analyses. We thank Giulio Antoniol for his insights on design pattern recognition. Finally, we thank the reviews for their comments which greatly improved our presentation of results.

References

- [1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using metrics to identify design patterns in object-oriented software. *Proc. IEEE-CS Software Metrics Symp. (Metrics'98)*, 1998.
- [2] J. Bieman, D. Jain, and H. Yang. Design patterns, design structure, and program changes: an industrial case study. *Proc. Int. Conf. on Software Maintenance (ICSM 2001)*, 2001.
- [3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [4] D. Campbell and J. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Co., Boston, 1966.
- [5] N. Fenton and S. Pfleeger. *Software Metrics - A Rigorous and Practical Approach Second Edition*. Int. Thompson Computer Press, London, 1997.
- [6] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading MA, 1997.
- [7] E. Gamma, R. Helm, J. R., and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA, 1995.
- [8] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd Edition, Volume 1*. John Wiley and Sons, New York, 2002.
- [9] R. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design concepts. *Proc. Int. Conf. on Software Engineering (ICSE'99)*, pages 226–235, 1999.
- [10] F. Kerlinger. *Foundations of Behavioral Research, Third Edition*. Harcourt Brace Jovaonvich College Publishers, Orlando, Florida, 1986.
- [11] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Proc. Working Conf. on Reverse Engineering*, pages 208–215, 1996.
- [12] D. Krantz, R. Luce, P. Suppes, and A. Tversky. *Foundations of Measurement*, volume I Additive and Polynomial Representations. Academic Press, New York, 1971.
- [13] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process Second Edition*. Prentice-Hall, 2002.
- [14] J. McLave and T. Sincich. *Statistics, Eighth Edition*. Prentice-Hall, 2000.
- [15] J. Michell. *An Introduction to the Logic of Psychological Measurement*. Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1990.
- [16] W. Munger, J. Bieman, R. Alexander. Coding concerns: do they matter? *Proc. Workshop on Empirical Studies of Software Maintenance (WESS 2002)*, 2002.
- [17] J. Nunnally. *Psychometric Theory, Second Edition*. McGraw-Hill, New York, 1978.
- [18] The Patterns Home Page.
URL <http://hillside.net/patterns/patterns.html>.
- [19] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.
- [20] F. Shull, W. Melo, and V. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report UMCP-CSD CS-TR-3597 or UMIACS-TR-96-10, University of Maryland, Computer Science Dept., 1996.
- [21] L. Votta and A. Porter. Experimental software engineering: A report on the state of the art. *Proc. 17th Int. Conf. Software Engineering (ICSE'95)*, 1995.
- [22] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.