OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study

James M. Bieman Dolly Jain Helen J. Yang
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523 USA
970-491-7096
bieman@cs.colostate.edu

Abstract

A primary expected benefit of object-oriented (OO) methods is the creation of software systems that are easier to adapt and maintain. OO design patterns are especially geared to improve adaptability, since patterns generally increase the complexity of an initial design in order to ease future enhancements. For design patterns to really provide benefit, they must reduce the cost of future adaptation. The evidence of improvements in adaptability through the use of design patterns and other design structures consists primarily of intuitive arguments and examples. There is little empirical evidence to support claims of improved flexibility of these preferred structures.

In this case study, we analyze 39 versions of an evolving industrial OO software system to see if there is 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 are **not** *less change prone — these pattern classes are among the most change prone in the system, and (2) classes that are reused the most through inheritance tend to be more change prone. These unexpected results hold up after accounting for class size, which had the strongest relationship with changes.*

Keywords Design patterns, object-oriented design, software changes, adaptability.

1. Introduction

Inheritance, abstraction, and polymorphism are mechanisms that should make adapting software easier and thus lower the cost of reuse, maintenance, and enhancement. Object-oriented (OO) design patterns represent a way to structure groups of classes to solve commonly recurring design problems. Each pattern allows some aspect of the system structure to change independently of other aspects. Thus, a system that is built out of patterns should be adaptable.

Evidence for the benefits of particular OO design strategies tends to consist of intuitive arguments, examples, and often the enthusiasm of a method's proponents. For example, the primary claim of Gamma et al's popular book [9] is that design patterns support reuse. The book describes through discussion and examples how each of the 23 listed patterns supports adaptability. The descriptions of the benefits of design patterns are compelling, accounting, in part, for the popularity of OO design patterns. However, there is no demonstration of the adaptability benefits on actual commercial software projects.

Although design patterns and other recommended design structures are intuitively appealing, their actual use on real systems determines their true benefits. And there is very little empirical evidence of the claimed benefits of design patterns and other design practices when applied to real development projects.

Case studies or controlled experiments can demonstrate the connections between design structure and external quality attributes such as maintainability or reliability. Briand et al studied the relationship between particular design attributes and fault proneness in both student and commercial projects [2, 5]. The studies of student projects were controlled experiments, while the studies of commercial projects were case studies. Results indicate that classes that are tightly coupled tend to be fault prone, while class cohesion is not related to fault proneness. Briand et al did not study maintenance issues such as changes or design pattern use.

Prechelt and Unger et al conducted controlled exper-

iments involving student programmers. One experiment found that pattern documentation increased programmer maintenance productivity and reduced errors, while another experiment was inconclusive [18]. Another experiment involving professional programmers found that, in most cases, maintenance time and/or maintenance effort was reduced when design patterns were used [19].

Our work focuses on assessing the quality of software designs, as well as developed software. We want to determine the relationship between design structures and external quality factors such as reusability, maintainability, testability, and adaptability. OO methods provide great flexibility in modeling both requirements and designs. This flexibility is a major reason for the popularity of OO methods, but flexibility also increases the number of design choices.

Methods to assess 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.

Most currently used OO design measures quantify properties of lower level design units such as classes, attributes, methods etc. to evaluate the designs. Briand et al [2, 4, 3, 5] studied coupling and cohesion measures that quantify OO software design quality. Their investigations of coupling measures do not directly address the coupling between classes that represent specific roles within architectural contexts such as design patterns [4]. Chidamber and Kemerer's OO metrics suite consist of class level measures, also without addressing roles with specific context [7]. Our investigation includes quantification of design structures, constructs used to design a system, ranging from the programming language implementation entities (classes, methods, variables etc.), and the design architecture and patterns used to connect lower level units such as classes. Using these design structures, we will be able to reason about the design and hence software system in greater detail. Hence, in our study we have used design patterns as a mechanism to quantify design structure and the architectural context of OO components.

This case study examines an evolving commercial OO system and looks at the relationship between design structure and software changes. Design structure is characterized by a set of class-level measurements, 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 measure the design structure of an early version, and study the relationship between the design attributes of this version and future system changes.

Table 1. System Level Measurements. 191 classes in Version A are also in Version B.

Version	Num. of Classes	Lines of Code
A	199	∼24,000
B	227	∼32,000

2. System Under Study

The study is conducted on a medium size 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. Experienced OO developers developed the system; they also made use of OO design patterns. The version control system allowed us to obtain multiple versions of the system and collect data on the transformations between 39 versions. Our focus has been the transformations between two specific versions of the system: version A, which is the first stable version of the system, and version B, which is the final version in our data set. Table 1 provides high-level information about versions A and B. Version A consists of 199 classes and approximately 24,000 lines of source code. Version B has 227 classes with approximately 32,000 lines of code. Of the 199 classes in Version A, 191 also appear in Version B. The 191 classes that appear in both Version A and Version B are the focus of this study. We extracted the object models of versions A and B and used the object models for pattern identification as well as metrics collection.

3. 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 effects 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. The specific hypotheses that we test via the case study are as follows:

- H1: Larger classes will be more change prone. A larger class has more functionality, thus there is a greater likelyhood 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.

- H3: Classes that are reused through inheritance more often will be less change prone. That is, classes with more direct subclasses and more descendants will be changed less often. We expect descendents to be added or modified more frequently than ancestor classes. This is because of the difficulty of modifying a class with many descendants and subclasses — any change to a superclass potentially affects a descendent.

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

4. Metrics Collection

We collected metrics on the entire system, individual classes in the system, and the number of changes to each class from version A to version B.

Assorted class-level metrics indicate internal properties of a class and relationships between classes. Two metrics are measures of class size:

- Total number of attributes (TotAtt): includes both instance variables (non-static member data) and class variables (static member data).
- Total number of operations (TotOp): includes both instance methods (non-static member functions) and class methods (static member functions).

Five metrics indicate properties of a class's relationship with other classes, either a property of an inheritance relationship or visibility through the C++ friends construct:

- Number of friends methods (Friends).
- Number of methods that are overridden (MO).
- Depth of inheritance (DOI): indicates a class's level in a class hierarchy. A base class — a class with no superclasses — has a DOI of zero.
- Number of direct child classes (DCC): a count of the number of immediate subclasses.
- Number of descendents (Desc): a count of all classes that are derived from the class either directly or indirectly.

The size and relationship measures were applied only to the classes in version A, since we are trying to identify the properties of the earlier version that can predict the number of changes that will later be applied. The *Together* tool and its metamodel, a product of TogetherSoft Corp., produced the class-level measurements.

Table 2. Measured Values of Class Properties of Version A. N = 191 classes.

Variable	Mean	Std. Dev.	Sum	Min	Max	Median
Changes	3.59	6.89	686	0	50	1
TotAtt	2.55	4.84	488	0	42	0
TotOp	11.03	14.36	2107	1	97	5
Friends	0.12	0.60	23	0	7	0
MO	2.01	2.56	384	0	21	2
DOI	0.98	0.86	187	0	4	1
DCC	0.77	5.14	147	0	61	0
Desc	1.00	5.70	191	0	67	0

We count the number of changes to each class that occur in the transitions from version A to version B. This count is a tally of the number of changes that are logged on the version control system for each class during the 39 version transition. 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 system 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. Our initial analysis of different classes of changes did not show any differences between the change type. In this paper, we do not classify the types of changes performed on the classes. Future work will report on the effects on our results, if any, between the change types.

Table 2 displays a quick view of the distribution of the measured values for each of the metrics. The maximum numbers of operations for a class in Version A of the system is 97 and minimum is 1; the maximum number of attributes is 42 and minimum is 0. Version A had maximum depth of inheritance of 4. The values of most of the metrics are not normally distributed, since the medians do not even approximate the means. As a result, we must either transform the data or apply non-parametric statistical tests in our analysis.

5. 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 [12], Antoniol et al [1], and Keller et al [10] 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 our 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 [9] book as the authoritative reference for this study.

Table 3 lists the patterns identified in version A of the system and number of instances of each pattern; 18 classes play roles in 16 pattern instances of four pattern types — Singleton, Factory method, Proxy and Iterator patterns. The identified patterns were not implemented exactly as specified by Gamma et al [9]. For example, we found four different implementations of singleton pattern. Though three of the implementations were not the standard implementations for singleton, they were used in the system to provide the functionality of singleton pattern.

Table 3. Patterns Identified in Version A of the System.

Pattern Name	Number of Instances
Singleton	10
Factory Method	1
Proxy	1
Iterator	4

Table 4. Correlation coefficients of class metrics with respect to the number of class changes (Changes).

	Pearson Correlation		Spearman Rank Correlation	
Metric	**Co-efficient**	**α-Value**	**Co-efficient**	**α-Value**
TotAtt	**0.495**	**<.0001**	0.15	**0.0356**
TotOp	**0.839**	**<.0001**	**0.50**	**<.0001**
Friends	0.178	**0.014**	**0.38**	**<.0001**
MO	-0.043	0.555	-0.14	0.0606
DOI	**0.354**	**<.0001**	0.05	0.5219
DCC	0.158	0.029	0.29	**<.0001**
Desc	0.210	0.004	0.29	**<.0001**

6. Evaluating The Hypotheses

We first examine the data looking for trends, before a detailed analysis of support for the hypotheses. Table 4 shows the correlation relationships between the class metrics and the number of changes to a class (Changes) from version A to version B of the system. The Pearson Correlation is a parametric statistical test which requires normally distributed interval or ratio data, while the Spearman Rank Correlation is a nonparametric test which compares variable rankings, and can be applied to ordinal data and to data that is not normally distributed [14]. The coefficients describe how a variable moves with Changes, and the α-values indicate significance. The coefficients with magnitude greater than 0.35, and α-values of less than 0.05 are in bold. They represent values of greatest interest. A significance level of less than 0.05 is a typical cutoff.

The size measure TotAtt clearly has coefficients of interest. Classes with more operations appear to be more change prone using either the Pearson or Spearman correlations.

Size, as measured by TotOp and TotAtt correlates to the number of changes; the impact of TotOp is much greater than that of TotAtt, especially when analyzed by the Spearman Rank Correlation. Since the data is not normally distributed, the Spearman Rank Correlation as the most relevant analytical tool. Friends has the next highest significant relationship, although Friends is not relevant to the original hypotheses. The correlation analysis on the measures related to reuse through inheritance (H3) shows some weak relationships; further analysis will determine whether or not

to accept the hypothesis. This initial analysis provides no information concerning effects of pattern use on Changes. We now examine support for the each of the hypotheses.

6.1. H1: Are larger classes more change prone?

The total number of operations in a class (TotOP) 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 produced by Minitab. This analysis clearly shows that classes with more operations tend to require more changes. The significance of this relationship is indicated by the low α-value from the Spearman Rank Correlation analysis indicated in Table 4. We can reject the null hypothesis for H1, and accept H1. Larger classes, measured by the number of operations, are more change prone.

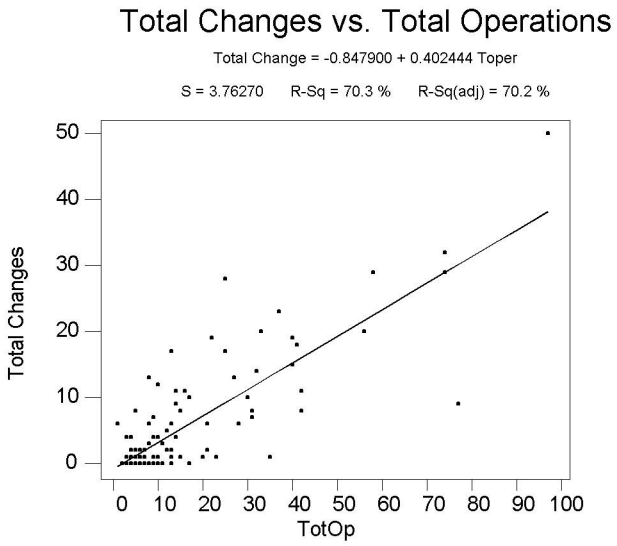

Figure 1. Scattergram of the total number of operations (TotOp) versus the total number of changes (Changes) with regression analysis results and fitted line plot.

6.2. H2: Are pattern classes less change prone?

Figure 2 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.

Although pattern classes appear to be more change prone, we need to see whether this is a result of a third factor. We know that larger classes — classes with more operations —

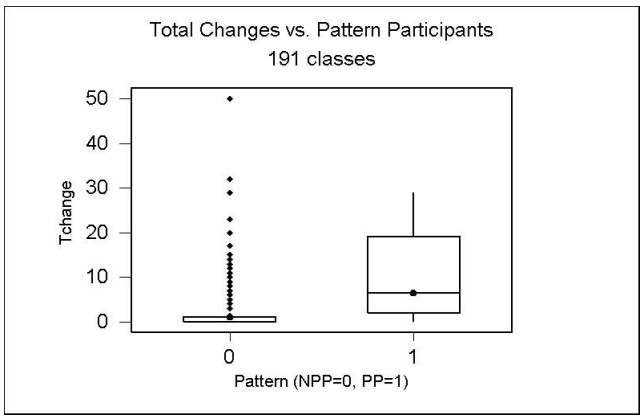

Figure 2. Box plots of the distribution of Changes for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPP = 0).

are more change prone; we need to see if pattern classes are larger. Figure 3 shows that pattern classes are larger. Thus, we need to adjust our analysis to account for the influence of size.

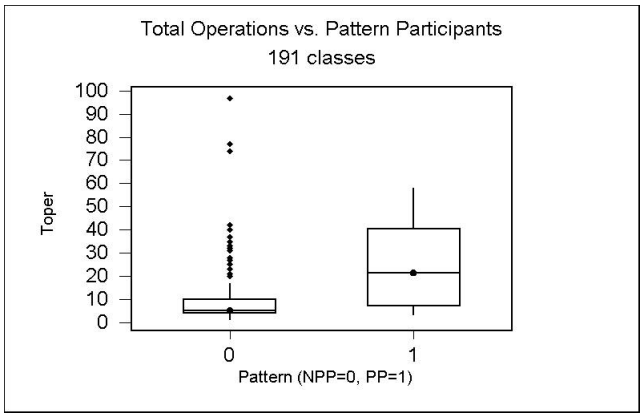

Figure 3. Box plots of the distribution of class size as measured by TotOpp for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPP = 0).

We control for the effect of class size by using change density rather than the total number of changes. Change density is the changes per operation (Changes/TotOp). Figure 4 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. This relationship is more visible when we remove one outlier in Figure 4, a pattern class with unusually large number of changes per operation. Figure 5 shows

distributions of change density without the outlier.

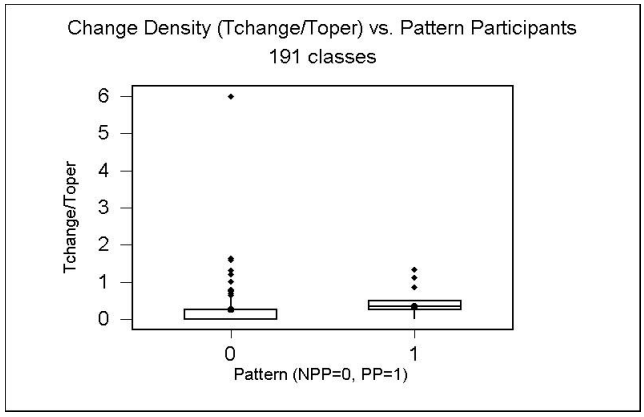

Figure 4. Box plots of the distribution of change density as measured by Changes/TotOpp for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPP = 0). There is one obvious outlier — a non-pattern class with 6 changes per operation.

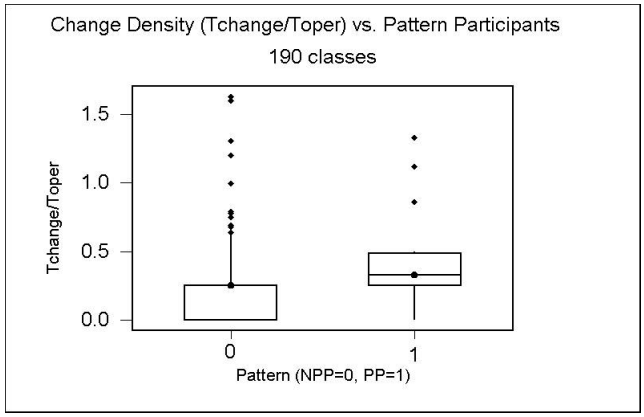

Figure 5. Box plots of the distribution of change density for pattern classes (PP = 1) versus non-pattern classes (NPP = 0) without the outlier shown in Figure 4.

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 determine if the data is normally distributed to see whether to apply parametric or non-parametric methods to test the hypothesis.

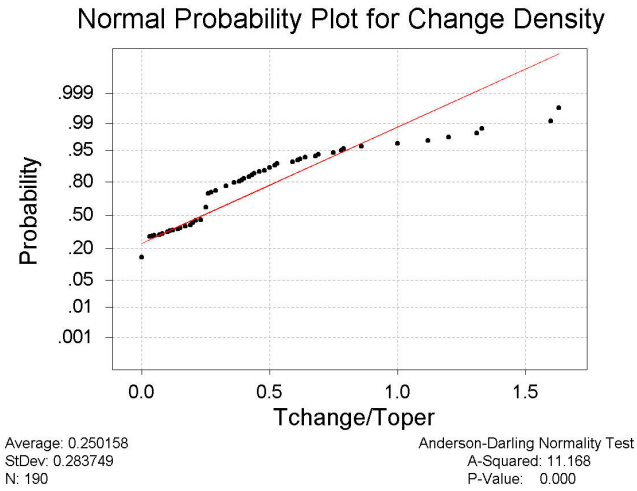

Figure 6. Anderson-Darling Normality test for Change Density.

Figure 6 shows the results of applying the Anderson-Darling Normality Test to determine whether the observed values of change density follows a normal distribution. The Anderson-Darling test measures how far the plot points fall from the fitted line in a probability plot. The observed values of change density do not follow a normal distribution. We applied the logarithmic transformation $\log(\textit{Change Density} + 1)$ to better fit a normal distribution. However the improvement is not great enough to warrant the use of parametric tests such as a T-Test to compare the change density of pattern classes to that of non-pattern classes. Rather, we apply non-parametric methods.

We apply 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 ¬H2, our original H2, with a significance of 0.0003. We conclude that classes that participate in design patterns are **more** change prone.

6.3. H3: Are classes that are reused through inheritance more often less change prone?

The correlation analysis, shown in Table 4, on the measures related to reuse through inheritance (H3) shows some weak relationships. However, the most significant and relevant relationships conflict with H3: classes with more children (DCC) or more descendents (Desc) are changed more, not less; the coefficient of 0.29 is not strong but it is very significant. The relationship between depth of inheritance

(DOI) and Changes is very small and not significant according to the Spearman Rank Correlation. In summary, the significant relationships support rather than refute the null hypothesis. Thus, **we reject the hypothesis: classes that are reused more through inheritance are not less change prone.**

7. Further Analysis

7.1. Relative Impact of Design Properties on Changes

The hypothesis tests indicate that classes with more operations and classes that participate in design patterns are more change prone. Classes with higher DCC and Desc also appear to be more change prone, with apparently less effect.

We evaluate the relative impact of these design properties on the number of changes by applying a regression analysis, which is an analysis of covariance [16]. We first ran a regression analysis using Changes as the dependent variable. The independent variables include the metrics with significant Spearman rank correlations in Table 4 — TotOp, Friends, DCC, and Desc — along with a variable Pattern, which is set to 1 for classes that play a role in a pattern, and set to 0 for classes that do not play a role in patterns. The standardized residuals from this model is not normally distributed, so we transformed the data using a logarithmic transformation to both TotOp and Changes. Using the transformed data, TotOp and Pattern are significant predictors. Friends, DCC, and Desc are not significant predictors with P-values of 0.530, 0.094, and 0.095, which are well above the 0.05 cutoff.

We provide further details describing our analysis of the relative effect of the two significant variables — pattern participation (Pattern) and TotOp. This analysis uses Pattern as the factor in the analysis, TotOp is the covariant, and Changes is the dependent variable. A regression model represents the relationship; Figure 7 shows the distribution of standardized residuals from the model. It is clearly not normally distributed, thus we again perform logarithmic transformations to both TotOp and Changes and run another regression model. Figure 8 shows normally distributed standardized residuals after the transformation. The regression equation is

$$\log(\textit{Changes}+1) = -0.453 + 0.910\log(\textit{TotOp}+1) + 0.174\textit{Pattern}$$

with $R^2 = 58\%$, Pattern = 1 for pattern classes, and Pattern = 0 for non-pattern classes. Clearly class size as indicated by the number of operations has a much greater effect on Changes than pattern participation.

We added the interaction factor $\textit{Pattern} \times \log(\textit{TotOp}+1)$ in another model to see if the affect of the interaction

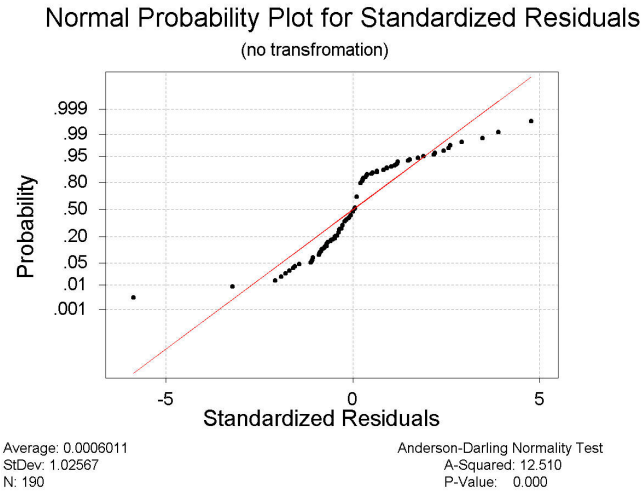

Figure 7. Anderson-Darling Normality test for standardized residuals for factors Pattern and TotOp.

was significant. The P-value for the interaction factor was greater than the cutoff significance level of 0.05. Thus we ignore the interaction factor.

By fixing the value for Pattern at either 0 or 1, we can create two equations — one for non-pattern classes and one for pattern classes:

Non-Pattern Classes:

$$\log(\textit{Changes}+1) = -0.453 + 0.910\log(\textit{TotOp}+1)$$

Pattern Classes:

$$\log(\textit{Changes}+1) = -0.279 + 0.910\log(\textit{TotOp}+1)$$

Both of the equations have a slope of 0.910 with an intercept difference of 0.174. We transform back to the original scale to get the following equations:

Non-Pattern Classes:

$$\textit{Changes} = 0.352(\textit{TotOp}+1)^{0.91} - 1$$

Pattern Classes:

$$\textit{Changes} = 0.526(\textit{TotOp}+1)^{0.91} - 1$$

The ratio between the number of changes in pattern participants versus the number of changes in non-pattern participants is constant as the number of operations increase.

7.2. What About Friends?

The Spearman Rank Correlation in Table 4 indicates that classes with more friend functions (Friends) tend to be more

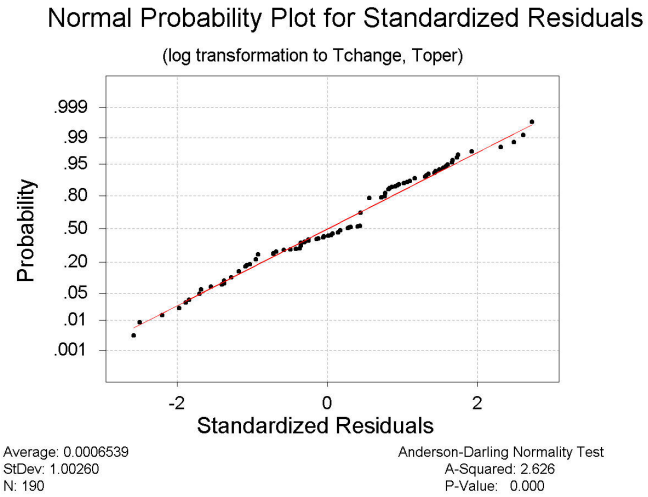

Figure 8. Anderson-Darling Normality test for standardized residuals for factors Pattern and TotOp after log transformations to TotOp and Changes.

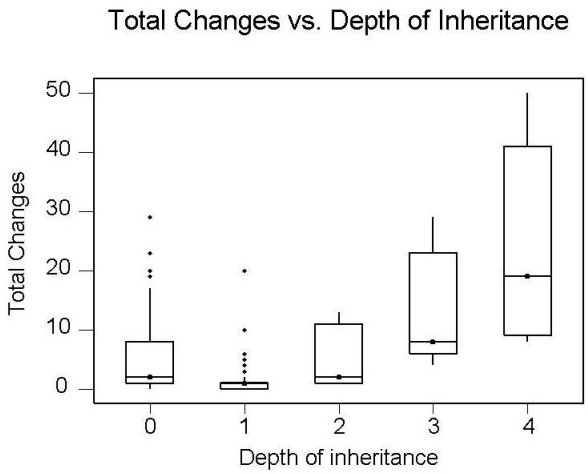

Figure 9. Distribution of Changes at different inheritance depths. The apparent relationship disappears after accounting for class size.

change prone. This relationship is significant, but the coefficient is only 0.38. Further, Friends did not contribute to the regression analysis when compared to TotOp or pattern participation. One reason is that there were few classes with friend operations. Note that Briand et al [2, 5] found the number of friend operations to be a predictor of *fault* proneness. Additional data is needed to evaluate the effect of friend functions on change proneness.

7.3. Trends That Disappear

An initial look at the data for depth of inheritance (DOI) as displayed in Figure 9 shows an interesting trend. Base classes, classes with DOI = 0, are changed more than classes at depth 1 or 2. The number increases as inheritance depth reaches 4 or 5. However, this trend disappears when we account for the effect of class size by using change density indicated by the number of changes per operation. Figure 10 displays the distribution of changes per operation for classes at different inheritance depths. This interesting trend was just a side-effect of the much greater effect of class size.

7.4. Discussion

The case study data includes many instances of the singleton pattern, and classes that play roles in the singleton pattern outnumber the classes that play roles in other patterns. The singleton pattern represents design reuse, the reuse of a solution to the problem of insuring that there can be only one instance of a particular class. Perhaps, singleton pattern use does not affect the change proneness of its participants. We also found that the designers used variants of Singleton that may have made it more difficult for maintenance programmers to adapt these classes.

Each design pattern is aimed to make specific changes easier. A design pattern can also make other changes more difficult. Thus, the benefit of using design patterns is realized only if the actual changes match the ones supported by the patterns in the system. Thus, one reason for our results may be that the use of design patterns was detrimental, because the actual changes did not match the patterns that were put in place in the earlier version. Determining if one can use the right patterns — patterns that actually reduce future maintenance effort — is an open research question. Only by looking at the actual use of design patterns in real projects can we determine if they have a positive or negative effect on maintainability.

8. 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 [11, 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 [8, 13, 15]. The dependent variable in this study, a count of changes is an intuitive

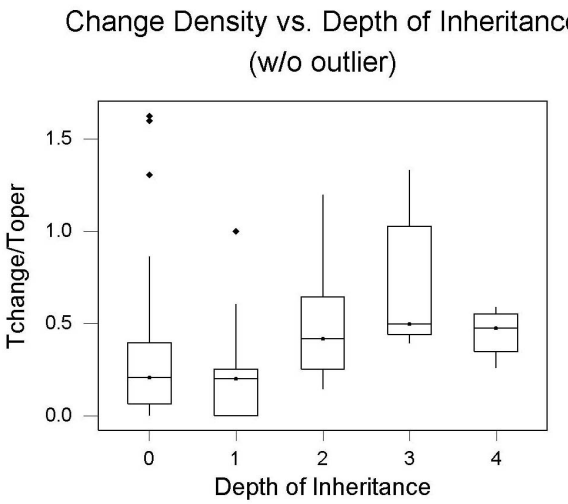

Figure 10. Distribution of change density (Changes per operation) at different inheritance depths. The outlier shown in Figure 4 is excluded. The trend in Figure 9 is nearly gone after accounting for class size.

measure of an aspect of maintenance effort. However, not all changes are equal, but a large number of changes over a series of 39 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" [11]. 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. The various structural metrics quantify various aspects of the system and should capture the notion of design quality. 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 [6, 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. Our analysis rejects hypotheses H2 and H3 and shows strong support for the null hypotheses. Before we accept these null hypotheses, we need to demonstrate internal validity for the relationship. We will need a good theory to support them and further data.

External Validity refers to how well the study results can be generalized beyond the study data. The study is based on a single system, undocumented patterns in the system are not identified and developers might have had inadequate knowledge about design patterns. It does represent one sample of an industrial development project during their transition to OO methods. Virtually all case studies exhibit weak external validity. Additional case studies conducted in different environments are necessary to determine the generality of the relationships.

9. Conclusions

The goal of this research was to use an industrial case study 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:

- Class size can predict the number of changes. Classes with a greater number of operations are changed the most. This result reinforces the common belief that component size is a dominant factor in predicting change effort.

- Classes that play roles in design patterns are changed more often than other classes. 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.

- Classes that are reused through inheritance tend to be changed more, rather than less, frequently. A change to a class with many descendants is effectively a change to all descendant classes. Thus, future changes are likely to become more difficult and regression test effort will surely be increased, since classes that descend from a changed class will need re-testing.

These results are from one case study, which represents one development environment, one application, and group of de-

velopers. Further studies are needed to compare results in different domains.

Although limited to one case study, the results do have some practical consequence for maintenance practice. Our result that size can predict changes supports common beliefs concerning class size and can help in predicting the number of changes. Our other results show that commonly held beliefs about patterns and inheritance are not supported by the case study data. A practical consequence is that we provide good reason to be skeptical about maintenance effects of new practices. We are now working to identify key reasons for the counter-intuitive results. Future work should provide some very practical guidelines to improve program maintainability when using design patterns and inheritance.

10. Acknowledgements

This work is partially supported by U.S. National Science Foundation 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. The Statistics Dept. of Colorado State University providing valuable help with our statistical analyses. Finally, we thank Giulio Antoniol for his insights on design pattern recognition.

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] L. Briand, J. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. *Proc. Int. Software Metrics Symp. (Metrics'98)*, pages 246–257, 1998.

[3] L. Briand, J. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[4] L. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1):91–121, 1999.

[5] L. Briand, J. Wüst, S. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. *Proc. Int. Conf. Software Engineering (ICSE'99)*, pages 345–354, 1999.

[6] D. Campbell and J. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Co., Boston, 1966.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[8] N. Fenton and S. Pfleeger. *Software Metrics - A Rigorous and Practical Approach Second Edition*. Int. Thompson Computer Press, London, 1997.

[9] E. Gamma, R. Helm, J. R., and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA, 1995.

[10] 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.

[11] F. Kerlinger. *Foundations of Behavioral Research, Third Edition*. Harcourt Brace Jovaonvich College Publishers, Orlando, Florida, 1986.

[12] C. Krämer and P. L. Design recovery by automated search for structural design patterns in object-oriented software. *Proc. Working Conf. on Reverse Engineering*, pages 208–215, 1996.

[13] D. Krantz, R. Luce, P. Suppes, and A. Tversky. *Foundations of Measurement*, volume I Additive and Polynomial Representations. Academic Press, New York, 1971.

[14] J. McLave and T. Sincich. *Statistics, Eight Edition*. Prentice-Hall, 2000.

[15] J. Michell. *An Introduction to the Logic of Psychological Measurement*. Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1990.

[16] J. Neter, M. Kutner, C. Nachtshein, and W. Wasserman. *Applied Linear Statistical Models, Fourth Edition*. Irwin, 1996.

[17] J. Nunnally. *Psychometric Theory, Second Edition*. McGraw-Hill, New York, 1978.

[18] L. Precheld and B. Unger. A series of controlled experiments on design patterns methodology and results. *Proc. Softwaretechnik '98.*, 1998.

[19] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy. Two controlled experiments assessing the usefulness of design pattern information in program maintenance. Submitted to *IEEE Trans. Software Engineering*, March 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.