To Appear in Proc. Int. Workshop on Program Comprehension (IWPC 2003), 2003.

# Understanding Change-proneness in OO Software through Visualization

James M. Bieman
*Computer Science Department*
*Colorado State University*
*Fort Collins, CO 80523*
*bieman@cs.colostate.edu*

Anneliese A. Andrews
*School of EE and CS*
*Washington State University*
*Pullman, WA 99164*
*aandrews@eecs.wsu.edu*

Helen J. Yang
*Computer Science Department*
*Colorado State University*
*Fort Collins, CO 80523*
*yangh@cs.colostate.edu*

## Abstract

*During software evolution, adaptive, and corrective maintenance are common reasons for changes. Often such changes cluster around key components. It is therefore important to analyze the frequency of changes to individual classes, but, more importantly, to also identify and show related changes in multiple classes. Frequent changes in clusters of classes may be due to their importance, due to the underlying architecture or due to chronic problems. Knowing where those change-prone clusters are can help focus attention, identify targets for re-engineering and thus provide product-based information to steer maintenance processes. This paper describes a method to identify and visualize classes and class interactions that are the most change-prone. The method was applied to a commercial embedded, real-time software system. It is object-oriented software that was developed using design patterns.*

## 1. Introduction

During software evolution series of changes are made to software. Changes can be due to a variety of reasons such as enhancements, adaptation, perfective maintenance or fixing defects. Some parts of the software may be more prone to changes than others. Knowing which classes are change-prone can be very helpful; change-proneness may indicate specific underlying quality issues. If a maintenance process can identify what parts of the software are change-prone then specific remedial actions can be taken. Thus, knowing where most changes are made over time can identify key change-prone classes, key change-prone interactions, and the evolution process can focus attention on them. Underlying reasons explaining why a class is change-prone can vary widely. There may be specific quality problems due to code decay (if changes are due to defect repair), the underlying architecture may have problems (again, if changes are due to defect repair) or encourage changes in certain classes (to add functionality or adaptations). Regardless, it is useful to know what these key classes and interactions are so that they get proper attention. Actions may range from stepped up quality assurance efforts to refactoring depending on the underlying cause for change-proneness.

While changes in individual classes can be counted, this does not reveal important aspects of code changes. Changes in response to a single defect report or change request may be local to a class and involve changes in only that single class, or may involve a whole collection of classes. The latter arguably may be more difficult to understand, take more effort to implement and to implement correctly. In addition, collections of classes that experience frequent changes together should be made visible, since they are prime targets for stepped up comprehension and possibly improvement efforts. We thus are not only interested in identifying and making visible classes that see the most lines of code changed, but, more importantly, identifying and visualizing classes that experience frequent changes *together*. That is, showing changes in these classes that are related. We call this *change-coupling* between classes.

In addition, it is important to analyze whether change-prone clusters of classes represent design structures and change-related interactions represent logical design interactions or not. Patterns are examples of clusters of classes that represent design structures (although not the only ones). A prior study of a commercial software development project by Bieman et al. [3] found that classes that play roles in design patterns were more, rather than less change prone. However, this analysis only counted number of changes in classes, but did not analyze whether changes in clusters of classes were related to the same change request, i. e. whether classes were change-coupled. Such an analysis can provide further understanding of the nature of the changes and how developers interact with a system. In this case study, we analyze the data from the prior study, a sizeable C++ system built using patterns, examining ways to visu-

alize and analyze the changes. The research questions we investigate are:

1. Is there a way to identify and visualize the most change-prone collections of classes in an object-oriented system?
2. Can change-proneness distinguish between local change-proneness and change-proneness due to change interactions of classes?
3. Do individual changes made in response to one "change request" affect only classes that are linked in the logical design of a system, or are there implicit connections between system elements that are not part of any design representation?
4. How do we make this information visible?

Section 2 gives background on existing work related to visualing software system evolution and fault-prone and change-prone behavior. Section 3 defines our analysis method. Section 4 describes a case study that applies this method to analyze an industrial software system containing patterns. Section 5 draws conclusions and points out limitations of our study.

## 2. Background

Others have studied the process of evolution and developed ways to visualize changing systems. Holt and Pak [7] developed a graphical tool to visualize design changes in successive versions. They use colors to indicate "hotter" or more recent changes. The system can identify changes in the explicit structural coupling between components. They found that low-level or module-level changes are more frequent than changes in the architecture.

Some prior work examines the growth of systems over time, without addressing the connections between components. Gall et al. [5] tracked changes in number and size of system components at various levels of abstraction — system, subsystem, module, and program-levels. Plots of the percentage of components that are changed and the growth rate of components indicate change-prone entities, which are candidates for re-design. Lehman et al. [10] also track the size and number of modules over time, and look for unusual trends.

Burd and Munro [4] examine the evolution of data clustering, which is the grouping of functionality with associated data. This work tracks design changes based on common data interactions and the calling structure in COBOL programs. In the systems studies, there was a growth in the number of functional units along with a growth in common coupling between functional units due to interactions with the same data.

The evolution of object-oriented software has also been studied. Mattsson and Bosch [11] apply the methods used by Gall et al. [5] in a case study of an object-oriented system. One objective is to identify the stable portion of a system, which can serve as a reusable framework. The other objective is to identify the most change-prone parts, which are candidates for restructuring. Like Mattsson and Bosch [11], they do not study connections between components. This work treats classes as the lowest level system components, and does not address either the internal characteristics of classes or the roles that a class may play in the design at a higher level of abstraction.

Antoniol et al. [1] recover designs from code and compare evolving designs in terms of a similarity measure. They identify links between components in different versions in terms of their similarity. We also recover designs from code. However, our focus is on the specific design roles of individual classes, or their *architectural context*, and both the explicit and implicit links between classes. Our aim is to see how the design structure of a system can affect the change-proneness of individual classes.

Much of the prior work on change-proneness is geared towards identifying fault-prone components (i. e. changes are due to defect repair). It is important to know which software components are stable versus those which repeatedly need corrective maintenance because of decay. Decaying components become worse as they evolve over releases. Software may decay after adding new functionality with increasing complexity, and due to poor documentation of the system. Over time decay can become very costly. Therefore it is necessary to track the evolution of systems and to analyze causes for decay.

Ash et al. [2] provide mechanisms to track fault-prone components across releases. Schneidewind [14] and Khoshgoftaar et al. [9] provide methods to predict whether a component will be fault-prone. Ohlsson et al. [12, 13] and von Mayrhauser et al. [16] combine prediction of fault-prone components with analysis of decay indicators. They also describe a method to identify both fault-prone components and fault-prone component relationships [13, 16]. Components are defined through physical architecture (i. e. based on clustering of files in directories). Components are ranked based on the number of defects in which a component plays a role. The ranks and changes in ranks are used to classify components as green, yellow and red (GYR) over a series of releases. Ohlsson et al. [12, 13] also analyzed corrective maintenance (change) measures via Principal Components Analysis (PCA) [8]. This helps to track changes in the components over successive releases. Box plots are also used to visualize the corrective maintenance measures and to identify how they differ between releases between fault-prone and non fault-prone components. Von Mayrhauser et al. [15, 16] take these measures of fault-proneness to deter-

mine a fault-architecture.

In this paper, like Mattsson and Bosch [11], we are interested not in components, but in classes and clusters of classes (both patterns and non-patterns) and how change-prone they are both locally and as change-coupled clusters. We examine class change proneness in part because the definition of class boundaries are unambiguous, in contrast to the notion of "components", which depend upon varying definitions. Also, we can identify classes directly from source code and do not depend upon design documentation which is often not available. Depending on the cause for the change, our analysis might result in identifying fault-prone classes and fault-prone interactions between classes. We adapt the measures for fault-proneness from von Mayrhauser et al. [15, 16] so that they can be applied to change-proneness of classes and class clusters, and their change-prone interactions. Similar to von Mayrhauser et al. [15, 16], we build a change architecture diagram of the most change-prone classes (rather than a fault architecture diagram).

The prior study by Bieman et al. [3] examined 39 versions of an evolving industrial object-oriented software system that evolved over a three year period to see if there is a relationship between patterns, other design attributes, and the number of changes. They found a strong relationship between class size and the number of changes — larger classes were changed more frequently. They also found two unexpected relationships: (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 held up after accounting for class size, which had the strongest relationship with changes. Figure 1 shows the difference in the distribution of changes per operation for classes that played roles in patterns from the classes that were not part of a pattern. The Mann-Whitney test confirmed with a significance of 0.0003 that, in the case study data, classes that play roles in patterns are more change-prone, as measured by changes per operation. However, this analysis did not address change-coupling between classes, it only measured individual changes in classes and disregarded any relationships between changes.

In this paper, we examine the same case study data focusing on the classes that are the most change-prone, distinguishing between local changes and change coupling between classes. First we identify the most change-prone classes, and then construct a change architecture, which includes only the most change-prone classes. We also examine the prevalence of classes that play roles in design patterns among the change-prone classes.

# 3. Approach

The study involves analyses of both the implementation structure of the case study software system and change logs. The analysis of the implementation structure provides a characterization of the individual classes in the system, and identifies the design patterns. The change logs provide the information needed to develop a change-architecture for the system; they identify individual changes and all classes that were changed for each reported change.

The analysis method consists of five steps:

1. Collect class-level implementation metrics of the system under study. We measure class size, and the properties of a class's relationships to the rest of the system.

2. Identify design patterns in the system. We identified only *intentional design patterns*, design patterns that developers use in a deliberate purposeful manner. These patterns should be documented, and they should have an 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.

3. Identify the changes in each class, and determine whether a change is local to a class or involves multiple classes. We use counts of these categories of changes to generate values of three change-proneness measures: local change-proneness (LCP), pair change coupling (PCC), and the sum of pair coupling (SPC). The measures are defined for a single change report and then aggregated for all change reports.

4. Identify the change-prone classes. A class is considered to be change-prone if each of the change-prone measures falls above a threshold. We determine the thresholds through a simple outlier analysis of the distribution of measured values.

5. Develop a change-architecture diagram for the change-prone classes. When changes are solely due to defects, this the same as the fault-architecture diagram in von Mayrhauser et al. [15, 16].

## 3.1. Class-level Measures

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 (Tatt): includes both instance variables (non-static member data) and class variables (static member data).

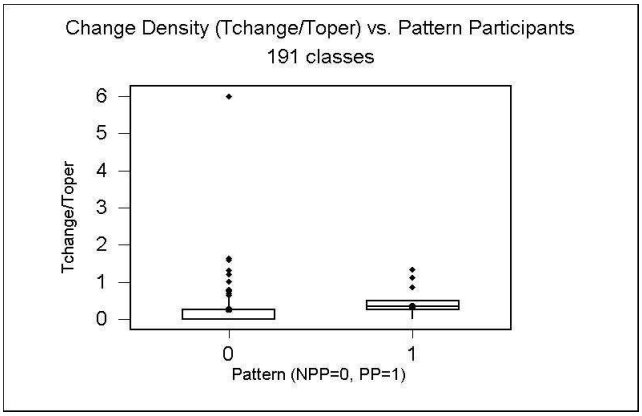


**Figure 1. Box plots of the distribution of change density as measured by Changes per Operation for classes that play roles in patterns (PP = 1) versus classes that do not participate in patterns (NPP = 0). Figure is from a prior report [3].**

- Total number of operations (Toper): 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.

Various tools can produce these measures. We used the *Together* tool and its metamodel, a product of TogetherSoft Corp.

### 3.2. Finding Patterns

We examine program source code, class diagrams generated by *Together* from code, and documentation to find the patterns using the following procedure:

1. Search for pattern names in the documentation of the system.
2. Identify the context of the classes identified in step 1 by analyzing the class diagrams. Once we find the classes whose documentation specifies something relating to a pattern name/role, we can look at the class diagrams to identify all the classes required to constitute a pattern. We look for the links and interactions 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.
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. [6] book as the authoritative reference for this study.

This process will only identify patterns that are documented by using well-known pattern names. We will miss undocumented patterns, and patterns that were used unintentionally. However, to have an effect on ease of adaptation, the developer responsible for making a change must know about the pattern. These patterns should be documented, and they should have an effect on the number of changes, since adaptability is a 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.

### 3.3. Change-proneness Measures

**Local change-proneness (LCP).** A change report associated with changes in a single class adds one point to the local change-proneness measure $LCP_i$. Thus

$$LCP_i = c_i$$

where $c_i$ is the number of change reports that involve only class $i$. The local change-proneness measure is analogous to the defect cohesion measure of von Mayrhauser et al. [15, 16].

**Pair change coupling (PCC).** Pair change coupling is associated with class pairs. Two or more classes are pair-coupled if they are involved in the same change report. For any $n$ classes $C_1$ to $C_n$ changed in the same change report, the pair coupling measure $PCC_{i,j}$ $(i \neq j)$ for each pair of classes $PCC_i$ and $PCC_j$ is increased by one. For $n$ classes in a change report, there will be $n(n-1)/2$ such pairs.

**Sum of pair coupling (SPC).** A class may be involved in many pair couplings. To account for many couplings, we compute the sum of all pair-couplings for each class. The sum of pair coupling of class $C$ is calculated as follows:

$$SPC_C = \sum_{i=1}^{n} PCC_{C,C_i} \; C \neq C_i$$

where $n$ is the number of classes other than $C$ and $PCC_{C,C_i}$ is the pair coupling between $C$ and $C_i$.

### 3.4. Determining Threshold for change-proneness

We use a strategy similar to those used to set thresholds that identify fault-prone components. Various authors have used a variety of approaches to set this threshold. Von Mayrhauser et al. [13, 15, 16] use defect cohesion measures for components and defect coupling measures between components to assess how fault-prone components and component relationships are. If the objective is to concentrate on the most problematic parts of the software architecture, these measures are used with thresholds to identify

- the most fault prone components only (setting a threshold based on the defect cohesion measure);
- the most fault prone component relationships (setting a threshold based on the defect coupling measure).

Von Mayrhauser et al. [15] consider a component fault-prone in a release if it is among the top 25% in terms of defect reports written against the component. In general, one would set the threshold based on available resources, quality, and objectives of the analysis (most problematic versus all components that have problems). The 25% threshold provided a manageable number of problematic components for further analysis. Similarly, a threshold may distinguish between component relationships that are fault-prone and those that are not. The threshold was set to an order of magnitude less than (or 10% of) the maximum value for the defect coupling measure. Setting the threshold is a subjective decision and depends on the objectives of the investigation and the number of fault relationships.

Here we use a different approach to determine the threshold. Rather than treating a component as a collection of files, we treat a class as a component. Box plots can depict the distribution for each change-proneness measure to identify outliers. The "box" represents the central 50% of the values. The outliers are the values that lie outside the box at a distance from the box boundary of more than 1.5 times the size of the box. This simple outlier analysis is used here for determining the threshold for change-proneness. We also adjust the criteria of outliers to consider obvious gaps in the distribution of values.

### 3.5. Diagrams of change-prone classes

Two diagram types can characterize the architecture, both logical and physical, of the change-prone classes. The diagrams depict only the change-prone classes and their relationships. The first diagram, a change-prone class diagram is a standard UML class diagram. However, it only includes the change-prone classes and their design relationships.

The second diagram, a change architecture diagram, shows how classes are linked via changes in the same change report. A change architecture diagram depicts all of the change-prone classes as rectangles, as in UML. The LCP and SPC values are shown inside the rectangles. Each pair of classes that have a PCC value that is greater than zero are linked; the link is annotated with the PCC value.

The two diagrams allow us to compare the connections inherent in the system design, as shown in the class diagram, with the connections implied by changes.

## 4. Case Study and Analysis

The study is conducted on a commercial object-oriented system implemented in C++. The system is a key part of an embedded real-time storage management system. This development project took place while the organization was in the process of adopting object-oriented methods. The system was developed with the support of a version control system over a period of several years. Experienced object-oriented developers developed the system; they also made use of object-oriented 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.

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 and all of the transformations between the versions are the focus of this study. We extracted the class diagrams of versions A and B from the code and used the diagrams for pattern identification as well as metrics collection. The size and relationship measures were generated only for 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.

We count the number of changes to each class that occur in the transitions from version A to version B, through 37 intermediate versions. 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 transitions. Changes can be corrective, adaptive, perfective, or preventive. As is the case with many industrial systems, the system under study had no maintenance history other than the comments in the code, the version control logs, and the recollection of the few system developers that we could find. Our analysis of different classes of changes did not show any differences between the change type. We used two key criteria to group changes to individual classes as elements of one change: matching comments documenting the changes, and matching check-in time stamps — check-ins within one minute of each other.

Version A of the system contained 18 classes that play roles in 16 pattern instances of four pattern types — Singleton, Factory method, Proxy and Iterator patterns.

Fig. 2 shows the distribution in the change-proneness values for each measurement as box plots. Most of the values for each of the three change-proneness measures are quite low — the boxes stay very close to the origin. This means that at least 75% of the measurements were at or very close to zero.

Change-prone classes are identified through box plot outlier analysis as those with LCP values above 10% of the highest LCP value — 36 out of 191 classes, in class pairs with above 50% of the highest PCC values — 29 out of 924 class pairs, and SPC values above 12% of the highest SPC values — 29 out of 191 classes. The SPC threshold was set at a gap in the outliers. Seventeen classes are beyond the threshold for all three criteria; these are the change-prone classes. This data clearly shows that there are clusters of classes whose changes are related, i. e. are change-coupled.

Observe that the outliers for the SPC measure have much higher values than for PCC. That is because several classes are members of many coupled pairs with few changes involving each pair.

Fig. 3 depicts the change-prone class diagram. It uses a UML-like notation to show the relationships between the change-prone classes. Of the change-prone classes, five classes play roles in design patterns. Three are Singleton classes, one is an Iterator class, and one plays a role in a Proxy pattern. Thus, five of the seventeen change-prone classes or 29% are pattern classes, while in all of system version A there are 18 pattern classes out of 191 classes or 9%. Another comparison is that the 5 change-prone pattern classes represent 28% of the pattern classes, while the 12 change-prone non-pattern classes represent 7% of the non-pattern classes. Pattern classes are clearly overrepresented in the set of change-prone classes.

One of our research questions was how to make these change-prone classes, class clusters, and their change relationships visible. We do this through (1) definition of the three change-proneness measures, (2) setting an appropriate threshoold (through boxplot analysis) to identify the relationships that should be visualized, and (3) graphing the remaining change-prone class clusters and the magnitude of change-relationships in a change-architecture diagram. Fig. 4 shows the change architecture diagram. Except for class C1, the change-prone classes form a connected graph. That is, classes C1 through C17 are linked by chains of pair couplings. Class C1 is not pair coupled with any of the other change-prone classes; it meets the change-prone threshold for PCC and SPC through many pair couplings with classes that are not change-prone. Four of the five change-prone pattern classes are linked directly through pair couplings. Class C7 is directly linked to nine of the other change-prone classes.

To answer our third research question, we compare the links in the change architecture shown in Fig. 4 with the logical structure of the implementation shown in Fig. 3. One observation is that the links in the two figures do not match. Several classes, for example C6 and C11, are linked in the change architecture (Fig. 4), but not in the class diagram (Fig. 3). C17, one of the pattern classes, is linked in the change architecture diagram to three other pattern classes — C8, C15, and C16, but C17 is not linked to these classes in the class diagram. These sets of classes may have been changed in response to a non-functional requirement, such as performance. Such a change might not be reflected in the design.

The class-level properties of the change-prone classes differ from those of the classes that are not change-prone. Table 1 gives the measurements for the change-prone classes, while Table 2 are the measurements for the remaining ones. First, the change-prone classes are changed far more often than the non change prone ones. The mean total number of changes (Tchanges) for change-prone classes is more than ten times higher than for non-change prone classes, and the median Tchanges is 18 times higher. The change prone classes tend to be larger — they have more attributes and operations. Neither group makes much use of C++ friends. There seems to be little difference in the num-

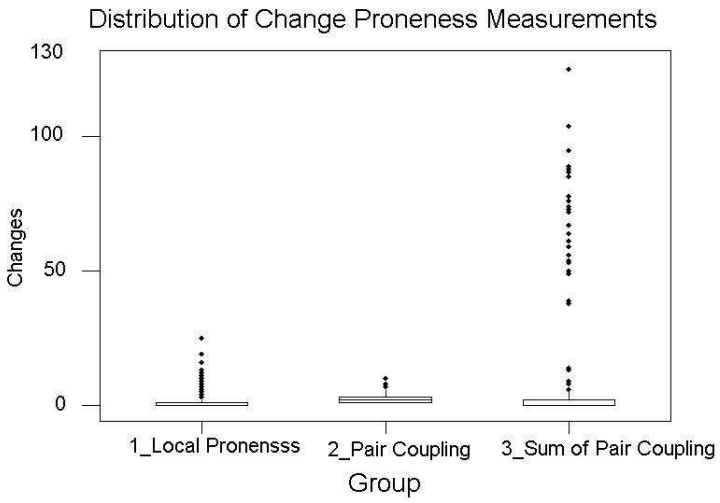


**Figure 2. Box plot showing the distribution of change-proneness measures: local change-proneness (LCP), pair change-coupling (PCC), and sum of pair coupling (SPC).**

ber of methods overridden (MO) for the two groups. The median depth of inheritance (DOI) is two for the change-prone group and one for the non-change prone group. The median values for the number of direct child classes (DCC) and descendents (DCC) are zero for both groups. However, the means are higher for the change-prone classes, especially for DCC. However, the comparatively high value of the mean for change-prone descendants (Desc) can be due to one or two change-prone classes with many descendents in the small sample.

## 5. Conclusions

In this paper, we addressed four research questions. The first was about identification and visualization of change-prone collections of classes in an object-oriented system. The second had to do with distinguishing local change-proneness from change-prone clusters of classes. We showed how to quantify the degree to which classes are change-prone both locally and in their interactions with others. We applied this method to a sizeable case study. For this case study, we found in response to research question 3 that change interactions between classes do not necessarily mimic functional interactions in the design of the system. This can have a variety of reasons. One example would be improvements of qualitative factors like performance. Performance improvements may trigger simultaneous changes in classes that otherwise do not interact with each other.

In response to research question four we visualized local versus cluster change-proneness through the change-architecture diagram and compared it to the design diagram. We also distiguished between change-prone clusters of classes involved in patterns and those which are not. Our visualization was simple and straightforward and driven by the change measures we identified. Future work in this area includes the use of color, overlays of change-architecture versus logical architecture, representation of other measures (e. g. size of box representing size of class). While there are a large number of potential enhancements, the key objective for us was to keep the representation sparse and uncluttered, and to emphasize the answer to the question: which clusters of classes are frequently involved in related changes?

## 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.

## References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-

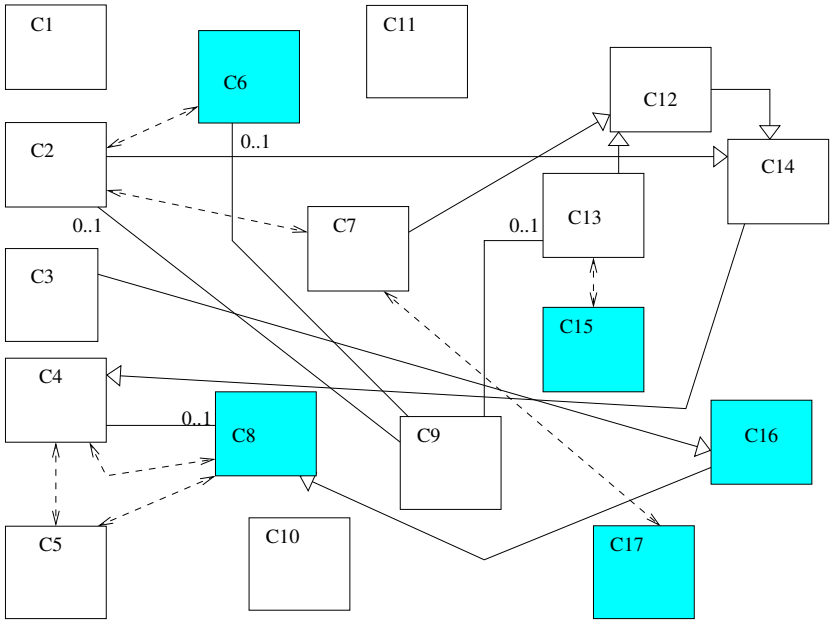


**Figure 3. Class diagram showing links between change-prone classes in the case study system. One directional use dependencies are excluded to provide a more readable figure. Class boxes that play a role in a design pattern are shaded.**

**Table 1. Class-level measurements in version A for change-prone classes.**

| N = 17 change-prone | Mean | Std.Dev. | Sum | Min | Max | Median |
|---|---|---|---|---|---|---|
| Tchanges | 20.82 | 10.27 | 354 | 10 | 50 | 18 |
| Tattr | 6.71 | 4.73 | 114 | 0 | 17 | 6 |
| Toper | 37.71 | 24.77 | 641 | 8 | 97 | 32 |
| Friends | 0.35 | 0.49 | 6 | 0 | 1 | 0 |
| MO | 2.82 | 2.63 | 48 | 0 | 8 | 3 |
| DOI | 2.00 | 1.62 | 34 | 0 | 4 | 2 |
| DCC | 0.82 | 1.33 | 14 | 0 | 4 | 0 |
| Desc | 2.47 | 4.06 | 42 | 0 | 10 | 0 |

**Table 2. Class-level measurements in version A for non change-prone classes**

| N = 174 not change-prone | Mean | Std.Dev. | Sum | Min | Max | Median |
|---|---|---|---|---|---|---|
| Tchanges | 1.91 | 3.22 | 332 | 0 | 20 | 1 |
| Tattr | 2.15 | 4.67 | 374 | 0 | 42 | 0 |
| Toper | 8.43 | 9.64 | 1466 | 1 | 77 | 5 |
| Friends | 0.10 | 0.61 | 17 | 0 | 7 | 0 |
| MO | 1.93 | 2.55 | 336 | 0 | 21 | 2 |
| DOI | 0.88 | 0.67 | 153 | 0 | 4 | 1 |
| DCC | 0.76 | 5.37 | 133 | 0 | 61 | 0 |
| Desc | 0.86 | 5.82 | 149 | 0 | 67 | 0 |

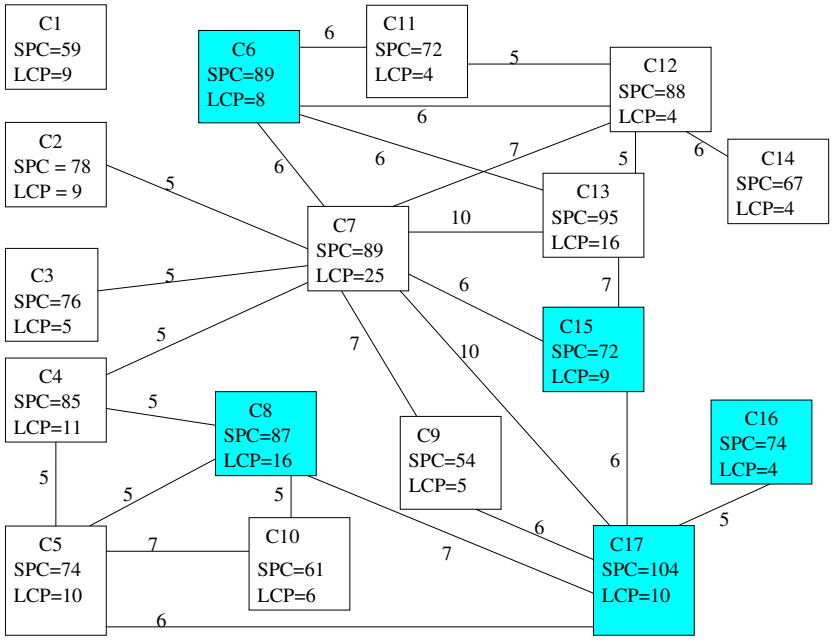


**Figure 4. Change architecture of change-prone classes in the case study system. Numbers along each link indicate the number of pair change couplings (PCC). Class boxes are annotated with the measurements of the sum of pair coupling (SPC) and local change-proneness (LCP). Class boxes that play a role in a design pattern are shaded.**


oriented software evolution. *Software Practice and Experience*, 31:331-355, 2001.

[2] D. Ash, J. Alderete, P.W. Oman, and B. Lowther. Using software models to track code health. *Proc. Int. Conf. on Software Maintenance (ICSM'94)*, pp.154–160,1994.

[3] 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).*, pp. 580-589, 2001.

[4] E. Burd and M. Munro. Investigating component-based maintenance and the effect of software evolution: A reengineering approach using data clustering. *Proc. Int. Conf. on Software Maintenance(ICSM 1998)*, pp. 199–207, 1998.

[5] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth. Software evolution observations based on product release history. *Proc. Int. Conf. Software Maintenance (ICSM 1997)*, pp. 160–166, 1997.

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

[7] R. Holt and J.K. Pak. Gase: Visualizing software evolution-in-the-large. *Proc. Working Conference on Reverse Engineering*, pp. 163–167, 1996.

[8] S. K. Kachigan, *Statistical Analysis – An Interdisciplinary Introduction to Univariate and Multivariate Methods*, Radius Press, 1986.

[9] T. Khoshgoftaar and R. Szabo, Improving code churn predictions during the system test and maintenance phases. *Proc. Int. Conf. on Software Maintenance (ICSM 1994)*, pp. 58–66.

[10] M. Lehman, D. Perry and J. Ramil. Implications of evolution metrics on software maintenance. *Proc. Int. Conf. on Software Maintenance (ICSM 1998)*, pp. 208–217, 1998.

[11] M. Mattsson and J. Bosch. Observations on the evolution of an industrial OO framework. *Proc. Int. Conf. on Software Maintenance (ICSM 1999)*, pp. 139–145, 1999.

[12] M. Ohlsson and C. Wohlin, Identification of green, yellow and red legacy components/ *Procs. International Conference on Software Maintenance (ICSM 1998)*, pp.6–15, 1998.

[13] M. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin. Code decay analysis of legacy software through successive releases. *Proc. IEEE Aerospace Conf.*, Track 7.401, March 1999.

[14] N.F. Schneidewind. Software Metrics Model for Quality Control. *Proc. Int. Software Metrics Symp. (Metrics 1997)*, pp.127–136, 1997.

[15] A. von Mayrhauser, J. Wang, M. Ohlsson, and C. Wohlin. Deriving a fault architecture from defect history. *Int. Symp. on Software Reliability Engineering (ISSRE 1999)*, pp. 295–303, 1999.

[16] A. von Mayrhauser, M. Ohlsson, and C. Wohlin, Deriving fault architectures from defect history. *Journal of Software Maintenance, Research and Practice*, **12**:287–304, 2000.