Discovering New Change Patterns in Object-Oriented Systems

Stéphane Vaucher, Houari Sahraoui and Jean Vaucher
GEODES, Université de Montréal
CP 6128 succ Centre-Ville
Montréal, QC, H3C 3J7 Canada
Email: {vauchers,sahraouh,vaucher}@iro.umontreal.ca

Abstract

Modern software has to evolve to meet the needs of stakeholders; but the nature and scope of this evolution is difficult to anticipate and manage. In this paper, we examine techniques which can discover interesting patterns of evolution in large object-oriented systems. To locate patterns, we use clustering to group together classes which change in the same manner at the same time. Then, we use dynamic time warping to find if a group of classes is similar to another when we ignore the exact moment when changes occur. Groups that exhibit distinctive evolution properties are potential candidates for new evolution patterns. Finally, in a study of two industrial open-source libraries, we identified four new types of change patterns whose usefulness is determined by perusal of the release notes and the architecture.

1. Introduction

An important aspect of understanding evolving software involves detecting change patterns in previously released versions. Consequently, much research has focused on discovering the presence of well-known patterns such as co-change and restructurings.

These patterns are quite important and recovering them can lead to a better understanding of systems; but every system is unique, both in terms of the skills and the experience of its developers, and in terms of its environment (e.g. technological). It is therefore reasonable to assume that each system contains some particular evolution patterns which cannot be known beforehand. These new patterns could be useful if they can be detected and found to be related to maintenance activities.

To discover new patterns, we propose the use of clustering on the change histories of classes in large deployed systems. Clustering will group together classes with similar change histories, each group implicitly representing a potential change pattern. As with most unsupervised data mining techniques, many groups are formed because of noise or coincidence, but the effort in verifying the quality of the grouping is relatively small compared to the cost of analysing each class separately.

We tested our approach on the evolution of two large open-source libraries. Potential patterns are first extracted by clustering. Then, to judge the relevance and usefulness of the groups found, they are analysed both internally, looking at the code, and externally using release notes. Finally, dynamic time warping (DTW) is used to detect cases when similar patterns occur at different points in time. In this study, we identified four distinct and useful change patterns.

The remainder of this paper is organized as follows. The following section presents previous work on change patterns. Section 3 describes the types of change that we use and how we measure change from successive code versions. Techniques to identify groups of classes with similar evolution are presented in Section 4. Then, Section 5 describes and analyses the groups found in two large systems. Finally, we conclude in Section 6.

2 Related Work

In recent years, much research has focussed on understanding evolution and change in software. Two important aspects are the definition of particular change patterns and their detection or recovery in existing systems.

Arguably the first catalog of change comes from the study on refactorings [3]. Refactoring is a particular type of change which restructures a system without affecting its external behaviour. Detection has been done using change in software metrics [2] and multicriteria matching[5]. A comprehensive list of available techniques is published in [7].

Co-change identification is another well-researched problem. It seeks to identify entities that change at the same time [15]. Often this is done by scanning version control systems for files that are checked-in either in the same transaction or within a small time window. This information can
then be used by developers to identify hidden dependencies between files. This was done using association rule mining [14, 15] and dynamic time-warping [1]. Different types of co-change patterns are defined in [4] and are discovered using formal concept analysis.

Change patterns affecting individual classes are defined by Lanza and Ducasse [9] who detect them using a combination of visualisation and metrics. More recently, Xing et al. in [13] use an algorithm to identify changes in design to do the same.

Though we use known techniques like clustering and time-warping, we differ from previous work by focussing on the semi-automatic discovery of new (potentially useful) change patterns; and the clustering relies on code versions only.

3. Modelling Change

Software systems evolve through a series of versions. We model the evolution of a class with an evolution blueprint which is a sequence of changes between successive versions.

For every class evolution, the amount of change is represented by two values: implementation change, IC (Section 3.1) and functional changes, FC (Section 3.2). These changes are included in a vector that defines its evolution blueprint. For a system with $n$ versions, a class contains $n - 1$ implementation changes and $n - 1$ functional changes as follows:

$$\text{Blueprint} = (IC_1 \rightarrow 2, ..., IC_{n-1} \rightarrow n, FC_1 \rightarrow 2, ..., FC_{n-1} \rightarrow n).$$

Blueprints are also used to represent change patterns. In that case, the change values are averages for all the classes in a group that changes in a similar way.

3.1. Implementation Change

Implementation change (IC) is defined as the percentage of code which is changed between two consecutive versions of a class.

Change is measured as a Levenshtein edit distance [10]: the minimum number of code elements that need to be added, removed or changed to transform one version into the next. Since we analyse classes with our department’s PTIDEJ reengineering toolkit which works primarily on byte-code, this metric calculates change in byte-code. Therefore comments are discounted. Furthermore, to discount cosmetic changes, methods in consecutive versions of a class are aligned by signature and the change in a class is the sum of the changes in corresponding methods. When a method is absent in either version, it is treated like a method containing no instructions. Finally, relative change is calculated by dividing by the size of the largest version.

3.2. Functional Change

Whereas IC is a measure of the work involved in modifying a class, functional change (FC) is meant to measure the effect of that work. There are many ways to quantify functional change. For example, one could use methods as the elements of a Levenshtein distance. But, following the example of [6], we take the perspective of an external user of a class and measure change to the public interface (API): and we consider the interface to include both inherited and local public methods.

More exactly we define FC as the number of added or removed public methods divided by the maximum size of the public interface. Modified methods whose signature remains the same are not included.

4. Locating Patterns

To find potential patterns, we use clustering, an automatic learning technique, to extract groups of similarly evolving classes. The classes they contain are further analysed according to their size, their maturity and their rate of change to verify if they constitute a pattern of interest. Finally, these groups are compared with one another in a time independent manner using dynamic time-warping. This section presents both techniques.

4.1. Clustering

The KMeans algorithm is an efficient clustering technique that uses an incremental approach to put similar elements into $K$ distinctive groups where $K$ is supplied as an input parameter. In our case, we use XMeans(eXtended KMeans) clustering [11] to identify potential change patterns. There two advantages to using this technique: first, it determines the number of cluster centers ($K$) automatically. Secondly, it can deal with missing change values (for versions when a class is missing). We consider every change in a blueprint as an independent attribute. Therefore, clustering cannot find similar patterns of change that recur at different times for different classes. To recognize these, we propose to apply dynamic time-warping (DTW) [8] on the clusters uncovered by XMeans.

4.2. Time-independent Cluster Analysis

Dynamic time-warping is a technique used to recognize shape similarity between a pair of sequences. This technique was recently used by Bouktif et al. [1] to find classes changing more or less at the same moment. When comparing sequences, DTW is allowed to stretch (warp) the sequences to achieve optimal matching. This technique cannot be applied directly to individual blueprints since they
often contain missing versions. We can however apply it to the cluster centers which provide complete evolution blueprints.

5. Exploratory Study

We conducted an exploratory study to validate the usefulness of the proposed technique in understanding an evolving code base. Thus, two open-source systems were picked for this study: Xerces\(^1\) and JFreeChart\(^2\). They were selected because of their popularity and maturity. Both are bundled in a multitude of commercial products meaning that this analysis should be representative of the study of real commercial systems. This section present a summary of the results. Complete results are available at [12].

The growth of the systems is presented in Table 1. Over the course of the last seven years and 35 version changes, Xerces has more than doubled in size while JFreeChart has grown from 48 to 800 classes. In this period, many classes have been removed, added and refactored. In total, we have gathered 690 and 1469 class evolutions histories respectively for Xerces and JFreeChart.

5.1. Cluster Descriptions

The XMeans clustering algorithm found a total of 11 clusters in Xerces and 12 in JFreeChart. Table 2 describes every cluster according to its size, to the change probability of its classes and their life expectancy. The size of a cluster is an indication of its importance in the system. The change probability, measured as the total number of versions where classes change (either in FC or IC) divided by the total number of versions where the classes exist, indicates its volatility. The maturity of a cluster measured as the average lifespan of its classes (in versions) is a sign of its cohesiveness since a cluster with short-lived classes might be more sensitive to random co-occurrences. Finally, the final column indicates the pattern identified which are presented in Section 5.3.

Information concerning the general change in the systems is also provided in the table. Both systems have a similar change rate of about 25%. JFreeChart has more short-lived classes due to the fact it grew by a factor of 10 and most classes were not present in the first versions.

5.2. Cluster Analysis

Our study of the clusters will focus on those that are significantly different from the others with interesting attributes identified in bold. We analyse, in order, the two largest clusters (X7 and J9), the most volatile clusters (J7, X8) and finally the mature classes (X4, X5). For each particular cluster, we find similar clusters using DTW. Very small clusters were ignored for they do not hold a sufficient number of classes to identify a genuine pattern.

Large Clusters. There are very large clusters that contain the majority of classes in both systems. These clusters (X7, J9) contain groups of unrelated classes which change rarely; but when they change, it is because of an important system-wide modification. The evolution blueprint of J9 is presented in Figure 1. This figure as well as the others, show the evolution of change rates (discretised, Y axis) over versions (X axis). It shows contains a first spike of change corresponds to a restructuring which is as important as the second leading up to a major release. Yet the version numbering does not reflect this fact.

Frequently Changed Clusters. The most active group of classes is found in JFreeChart (J7). Study of the code and documentation shows that most of the classes are designed to be extension points. For example, two classes named

### Table 1. System Descriptions

<table>
<thead>
<tr>
<th>System</th>
<th>Num</th>
<th>Version Initial</th>
<th>Last</th>
<th>Size (classes)</th>
<th># of blueprints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td>36</td>
<td>1.0.1 (1999)</td>
<td>2.9.0 (2006)</td>
<td>210</td>
<td>518</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>36</td>
<td>0.5.6 (2000)</td>
<td>1.0.6 (2003)</td>
<td>48</td>
<td>790</td>
</tr>
</tbody>
</table>

### Table 2. Cluster Descriptions

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Size (# of classes)</th>
<th>Probability of change</th>
<th>Lifespan (versions)</th>
<th>Patterns found</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>9 (15%)</td>
<td>22%</td>
<td>17</td>
<td>CS</td>
</tr>
<tr>
<td>X2</td>
<td>16 (4%)</td>
<td>25%</td>
<td>19</td>
<td>CS</td>
</tr>
<tr>
<td>X3</td>
<td>25 (3%)</td>
<td>35%</td>
<td>19</td>
<td>CS</td>
</tr>
<tr>
<td>X4</td>
<td>26 (4%)</td>
<td>37%</td>
<td>17</td>
<td>CC</td>
</tr>
<tr>
<td>X5</td>
<td>96 (14%)</td>
<td>25%</td>
<td>38</td>
<td>CC</td>
</tr>
<tr>
<td>X6</td>
<td>38 (5%)</td>
<td>48%</td>
<td>18</td>
<td>CS</td>
</tr>
<tr>
<td>X7</td>
<td>377 (55%)</td>
<td>22%</td>
<td>13</td>
<td>PC</td>
</tr>
<tr>
<td>X8</td>
<td>13 (2%)</td>
<td>48%</td>
<td>16</td>
<td>USA</td>
</tr>
<tr>
<td>X9</td>
<td>59 (8%)</td>
<td>23%</td>
<td>11</td>
<td>CS</td>
</tr>
<tr>
<td>X10</td>
<td>10 (1%)</td>
<td>38%</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>X11</td>
<td>30 (4%)</td>
<td>23%</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>690 (100%)</td>
<td>26%</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\)http://xerces.apache.org/xerces-j/

\(^2\)http://www.jfree.org/jfreechart/
JFreeChart and ChartFactory, are used by users of the library to create charts. Whenever a new chart type is implemented in the library, a new method is added. A co-changing test class JFreeChartTest is also present in the cluster. The high level of change is presented in Figure 2 which shows that only a few versions exhibit no change.

The most active cluster in Xerces, X8, shows high level of change throughout the first half of the life of the system; then none for the second half. Examination of the cluster shows that it contains all the classes from a particular package. The available documentation indicates that this package was a constant source of problems and if there is no change in the second half, it is because the developers removed those classes, presumably due to the high maintenance cost.

**Maturity Analysis.** When many versions of classes are available, the groupings returned by the clustering should be more significant. We have a few clusters noticeably more mature than the others like X4, X5 in Xerces. These classes define the document object model (DOM) for XML, WML and HTML documents. As the system evolves, developers incrementally implement a series of specifications defined by the W3C. The initial model is stable, but the implementation changes. JFreechart on the other hand has no long-lived clusters.

**Matching Clusters.** When trying to locate similar patterns using DTW, we opted for a simple approach of identifying for each cluster, which of the others was the most similar and then verify a deeper relationship. Three pairs were deemed interesting: \((J1, J2)\), \((X4, X5)\) and \((X1, X8)\). The first two pairs were grouped together because of similar function change while the last matched because of problematic evolution of complex functionality.

Figure 3(b) shows the evolution changes for \((J1, J2)\) and \((X4, X5)\). Classes show similar changes in their interfaces, yet seem to evolve differently in implementation. The clusters contain important classes in their respective systems (renderers for charting library and XML DOM documents). Since the code needed to be flexible, the developers used inheritance to organise the code; but inheritance is used differently in each system. In JFreeChart, the inheritance tree is separated in two major types of classes (XY and Stack chart renderers) and changes affect either one of the two types. In Xerces, changes are divided vertically: X4 holds the superclasses where most of the functionality is implemented while X5 has the small subclasses that delegate most calls to superclasses.

The \((X1, X8)\) pairing is also interesting. We saw earlier that the classes of X8 were badly designed and disappeared after the first major version. In version 2, the functionality mostly got moved to new classes which we find in X1. X1 and X2 were matched by DTW because of a high level of functional change occurring in early minor versions followed by lower change (bug fixes or minor adjustment stabilisation).

### 5.3. Patterns Found

From the results presented in the previous section, we consider that four different patterns were located:

- The first is a pattern of frequent and substantial change. We name classes in this group *usual suspects* because
they are likely to come under review whenever change is considered. Clusters X8 and J7 are examples. This pattern can be due to bad design: X8 had to be rewritten. It is however not necessarily sign of poor quality: in JFreeChart, it corresponds to classes that are meant to change regularly;

- **Concern stabilisation** (X1,X8) occurs when newly introduced code (high levels of functional change) requires a few versions before stabilising. Changes in the first major version of Xerces even required a rewrite before becoming completely idle;

- **Punctual changes** (X7,J9) are one-time changes that affect a large number of unrelated classes. These can indicate the severity of a version change even when the numbering system doesn’t indicate it;

- **Common concerns** (X4,X5,J1,J2) are long-lived classes that co-evolve functionally since they all implement the same concern.

In Table 2, these patterns are indicated by the following abbreviations: US, CS, PC and CC respectively.

### 6 Conclusion

In this paper, we show how clustering can be used to identify change patterns in evolving systems. This technique was tested on two mature industrial open-source systems. From the resulting clusters found, we could identify four distinctive and useful change patterns.

In the near future, we plan on extending the work to include more change metrics and more systems to see if some distinctive and useful change patterns.

To this effect, we are currently working with a quality assurance team in a large company which needs tools to judge the quality of outsourced software systems.

### References


