# Do Software Libraries Evolve Differently than Applications?

Stéphane Vaucher Houari Sahraoui
Département d'Informatique et de Recherche Opérationnelle
Université de Montréal
{vauchers, sahraouh}@iro.umontreal.ca

## Abstract

*More and more, developers use reusable components like libraries to produce high quality software systems. These systems need to satisfy not only the initial demands of their stakeholders, but they need to also offer support for future, changing requirements. While several studies have looked at the cost of modifying systems, there exists no work comparing if libraries evolve differently than applications. This study attempts to verify this assumption quantitatively.*

*In this paper, we define design changes metrics to estimate the amount of high-level change required of individual classes and use metrics to describe their structure. These measures are then used as inputs in models capable of predicting code change. We used machine learning techniques to build these models and tested them on the evolution of industrial open-source systems. Two of the systems were libraries, and two were standalone applications.*

*We found that while design changes are systematically correlated with code changes, structure metrics are better predictors of code change in libraries with well developed class hierarchies. With the two applications without this characteristic, structure alone was a poor predictor.*

## 1. Introduction

If modern software systems have reached their current levels of complexity, it is in many regards due to the use of software libraries. Using good libraries, application developers are able to focus on domain-specific functionalities while relying on libraries to address more general concerns. Although there are clear benefits to reuse, identifying high-quality libraries can be difficult.

When choosing a library there are several qualitative aspects to be taken into account. Several studies [19, 20] have shown that using libraries can improve the reliability of a system as they generally contain fewer bugs, but correctness and reliability are but minor aspects of quality. Since most of the costs of a system come from its evolution [11], it is interesting to assess its *changeability*, its ability to be adapted and extended to meet future, possibly unexpected needs.

Analysing the changeability of software has been addressed in research fields like change-impact analysis [1] and quality prediction [16], but little effort has been put in comparing how libraries differ from end-applications. For one, the stakeholders are different: applications should satisfy the present needs of end-users while libraries implement specific functionalities accessed programmatically by other systems. As such, they need to insure a level of backwards-compatibility to limit the impact of their evolution on applications.

In this paper, we build on existing work with a study that identifies what features affect the changeability of libraries by comparing them to those affecting applications. We have selected two popular open-source Java libraries which are bundled in Java Development Kit distributions. As for applications, we have chosen popular GUI applications that use many libraries to provide key parts of their functionality.

As with previous studies, we consider that there are structural characteristics that enable changes and that these changes might be predictable. If a predictive model identifies structures that enable change, then a system is more changeable. We built models to predict the relative code change of classes in both libraries and applications. In a first step, we looked at how design changes affect implementation changes as this should be a source of change. We then looked at how structure alone predicts changes, thus finding if similarly structured classes are generally changed in the same way. Finally, we combined both approaches to see how system structure affects its changeability. Our results show that the predictive capability of all of the models produced differs significantly depending on the type of system analysed. Changes in well-structured libraries are generally localised, while changes in applications are not.

This paper is organised as follows. In the next section, we explain what is a prediction model and which are the measures used to build one. In section 3, we go over the experimental settings and include a description of the selected systems and the techniques used to conduct the study. We detail the results and relevant inter-

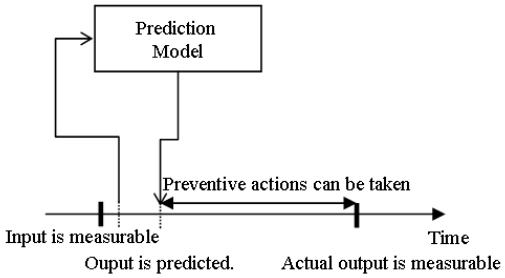



**Figure 1. A time-based representation of a prediction model**

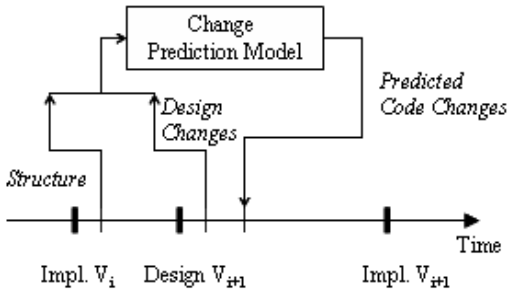



**Figure 2. A time-based representation of our code change prediction model**

pretations in section 4. Related work is presented in section 5. We conclude with a presentation of experimental threats and with our different research perspectives that will guide us in future work.

## 2. Predicting Changes

Predictive models for software quality are used to establish a relationship between two software attributes that can be measured at different moments in the software life cycle. If we consider a timeline, two events characterise a predictive models: the moment when the input attributes can be measured and the moment when the output attribute is actually measured. The purpose of these models is to assist decision making by providing objective estimates as early as possible allowing for preventive actions to be taken if it is necessary. To represent a prediction model, we use a timeline representation as shown in figure 1.

For example, in [6], one of the models predicted the error proneness of C++ classes using as input their export coupling. Class coupling is measured after the design phase (design coupling metrics) and after the implementation phase (implementation coupling measures). The number of post-release faults generated by a class (error proneness) is actually measured after a certain time of operation. A reliable model can predict error proneness as soon as the implementation is finished before the release. This can help a quality assurance team to focus its attention on the high-risk classes.

In the specific case of code changes, the goal is to predict the quantity of changes needed to get from one version to the next. These values can then be used as a basis for effort estimation or to choose amongst alternative designs. As previously mentioned in section 1, we have selected to include two key factors: design changes which can be measured comparing the current design with the design proposed for the next version, and the structure of the system, which can be measured directly from the code. Figure 2 depicts our code change prediction model.

### 2.1 Design Change Quantification

In object-oriented systems, the interface of a class defines the services it offers and it represents an important part of the design of a system. We have consequently decided to quantify design changes using the changes to interfaces. We have used two metrics that have been adapted from the ones described in Grosser et al. in their study of software stability [8].

We measure the proportion of public methods added (PPMA, equation 1) and removed (PPMD, equation 2). We have bounded them in a $[0, 1]$ interval by using the maximum size of the public interface as a normalising factor. We do not attempt to handle a renamed method, so any change to a method's signature counts as a removal followed by an addition.

$$PPMA = \frac{|I(N+1) - I(N)|}{MAX(I(N)|, |I(N+1))} \quad (1)$$

$$PPMD = \frac{|I(N) - I(N+1)|}{MAX(I(N)|, |I(N+1))} \quad (2)$$

where I(N) is the set of public methods in the interface of our class at version N.

For example, let us consider a class *C* with 3 public methods at version i and 2 at version i+1 (see table 1).

| | |
|---|---|
| class C<br>public m1();<br>public m2();<br>public m3(); | class C<br>public m3();<br>public m4(); |

**Table 1. Two consecutive versions of class C**

From the 3 methods in version 1, only one (m3) remains in version i+1 and a new method (m4) is added and the largest interface contains 3 methods. This gives:

$PPMA = 1/3$ and $PPMD = 2/3$

In addition to being easy to compute, our two design change metrics have the advantage that they can be extracted from the design as well as from code.

### 2.2 Code Change Quantification

The measurement of the code changes, also called *code churn*, usually takes the following form:

$$code\ churn = lines_{added} + lines_{deleted} + lines_{modified}$$

This measure was used in many empirical studies on legacy systems with large-size modules [12, 13, 22], but it gives erroneous results when the code has undergone cosmetic changes. For example, adding comments or pretty printing might all be considered as major changes to the class code.

In our case, the tools we use to compute our metrics are adapted to Java and work on the compiled byte code, thus comments and formatting are ignored. To extract code change, we used the same compiler, turning off all possible optimisations. This was feasible since we observed that the size in byte code and the size in the source code are highly correlated (98%).

Between two versions of a class, we match methods using their signatures and calculate an edit distance on the sequence of byte code they contain. Specifically, we have used a Levenshtein edit distance [15], which identifies the minimal number of operations required to transform one sequence into another. It associates a cost for every insertion, deletion and modification of a byte code instruction. In this study, we used its simplest form where every operation has the same cost.

The change to a class is measured as the sum of the changes in its concrete methods. This value is then normalised using maximum size in byte code instructions to bound the value in a $[0, 2]$ interval. When a method is absent in a version, it is considered to contain an empty sequence.

## 2.3 Structure Quantification

Since the beginning of software engineering, there has been an effort to measure different aspects of code. Traditionally, we think of size [28], complexity [18], coupling and cohesion [31]. These concepts have been exported to object-oriented systems with the addition of inheritance [7].

Table 2 presents the metrics used to quantify these structural attributes in this study. These are well-studied and have been used in numerous empirical studies such as [2, 5, 16].

| Metrics | Description |
|---|---|
| CBO [7] | Coupling between objects |
| DCMEC [5] | Descendant class-method export coupling |
| DCAEC [5] | Descendant class-attribute export coupling |
| LCOM5 [7, 10] | Lack of Cohesion of Methods |
| ICH [30] | Information-flow-based Cohesion |
| DIT [7] | Depth of the inheritance tree |
| ABS | The ratio of abstract methods |
| WMC [7] | Weighted method complexity |
| NMD | Number of methods declared |
| NAD | Number of attributes declared |
| NI | Number of byte code instructions |

**Table 2. Structure metrics**

### *2.3.1 Coupling metrics*

Coupling is a measure of links between different modules. We used 3 metrics to quantify different types of couplings. The first, CBO, counts the number of classes that are used by or that use our class either by calling a method or directly accessing an attribute. Our two other measures count how many times the subclasses of a class invoke its methods (DCMEC) or access its attributes (DCAEC).

### *2.3.2 Cohesion metrics*

Cohesion is a measure that a class is working towards one single objective. Two different kinds of cohesion (or lack thereof) are covered by our metrics. LCOM5 calculates a ratio of unused attributes by the methods of a class. ICH measures the internal use of methods.

### *2.3.3 Inheritance metrics*

Our first metric, DIT, measures the distance between a class and its root in its inheritance tree. We defined ABS as the number of abstract methods over the total number of methods provided.

### *2.3.4 Complexity and size metrics*

WMC defines the complexity of a class as the sum of the complexities of its methods. We used McCabe's complexity to calculate the exact complexity of methods [18]. NMD, NAD and NI are size metrics which count respectively the number of methods, attributes and instructions declared in a class.

# 3. Experimental Setting

In this section, we present our experimental settings. The experiment follows the methodology presented in [27] which prescribes starting with the statement of our high-level goals, hypotheses, and variables. This is followed by details of our data and validation techniques.

## 3.1 Objective, Hypotheses, and Variables

### *3.1.1 Objective*

The goal of this experiment is to study factors that may influence the amount of changes in the code in open-source libraries and applications and to compare the results.

### *3.1.2 Hypotheses*

From our goal, we derived three concrete experimental hypotheses.

- H1 : Design changes are sufficient to predict the changes in its code;
- H2 : The structure of a system is sufficient to predict the changes in its code;
- H3 : The combination of both structure and design changes are better predictors of the changes in its code.

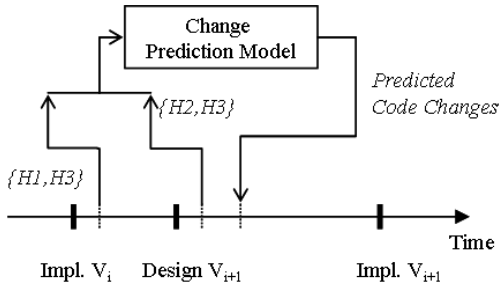



**Figure 3. Experimental setup**

The two first hypotheses should validate, to some extent, intuitive relations that either design changes in a system or its structure affect the maintenance effort. The third verifies if the structure and design changes are complementary. A badly structured system does not indicate a need to change, but rather if it can be changed with little effort. Similarly, design changes identify the need to change, but not the system's ability to accommodate changes.

We have verified our three hypotheses at two levels. To start, we have tested these hypotheses at a general level, combining the data extracted for each type of system. At this level, we limited ourselves to quantitatively validating our hypotheses. In our second phase of investigation, we tested our hypotheses on individual systems and interpreted our results since changes are generally context specific. The terms H1, H2 and H3 will be used in subsequent sections to denote the corresponding experiments.

#### *3.1.3 Variables*

To transform our hypotheses into testable prediction models, we need to define quantifiable input and output variables. In this section, we will present the variables we derived from our hypotheses. Our experimental setup is shown in figure 3. Our input varies depending on the hypothesis tested. For hypotheses 1 and 3, the models use the 11 structure metrics shown in table 2, and for hypotheses 2 and 3, they use the two design change metrics from section 2.1. To extract the metrics, we used PADL and POM [9], tools developed by our research team.

### 3.2 Data collection

To make sure the results of our study are generalisable, we need to make sure our data is representative of real industrial systems [25]. As there are more and more open-source systems produced by large companies available in online repositories, this is now possible. In particular, over the past decade, there has been a move to include quality open-source libraries in commercial products, and this trend is likely to continue.

#### *3.2.1 Selected Systems*

For this study, we selected four systems two of which, Xerces-J and Xalan-J, are libraries and two, ArgoUML and Azureus, are standalone applications. All four systems are reasonably large containing in every version between 200 to 1,200 classes. They have all been developed for a few years, have large communities of users, and implement functionality defined by external stakeholders.

Historical data for the systems was extracted from their code versioning systems. Most projects use 3 different categories of versions: bug fixes (or micro versions), minor versions, and major versions. Bug fixes are very frequent, but are generally limited to corrective maintenance in which design changes should not have much effect. Major versions contain many types of maintenance and affect most if not all aspects of a system, but are very rare; as an example, the popular Apache HTTP Server[1] is only at its second major version after over ten years of development. We have consequently decided to study minor versions. Minor versions include corrective maintenance, but also functional changes, so we can be reasonably assured that they will contain different examples of maintenance tasks. Additionally they are frequent enough for us to build decently sized data sets.

*Xerces-J.* [2] is an XML parser which is used by many commercial products. This library was originally developed by IBM under the name XML4J and was released in 1999 to the Apache Software Foundation to jump start its open-source implementation of XML standards. It conforms to many XML standards like XML schema, DOM, and SAX as defined by the World Wide Web Consortium[3] (W3C). It is distributed with the Sun distribution of the Java Development Kit (JDK)[4]. It is also the reference implementation of the Java XML Parser standard.

*Xalan-J.* [5] is an XSL processor whose origins greatly resemble those of Xerces. Initially developed by IBM as LotusXSL, it was also contributed to the Apache Software Foundation in 1999. In 2001, Sun provided a compiler and the Xalan project took off. Xalan-K currently implements multiple specifications such as XSLT, Xpath and XML QL, all defined by the W3C. It is also distributed with Sun's JDK.

*ArgoUML.* [6] is a UML diagram editor. Since its creation, it only implements a subset of the UML 1.4 specification as defined by the Object Management Group (OMG)[7]. To our knowledge, there are two commercial projects that have adapted its code base for commercial products PoseidonUML[8] and MyEclipse UML[9]. This application relies on a few libraries including Xerces and Xalan.

*Azureus2.* [10] is a popular P2P application that implements the bittorent[11] protocol. At the time of the study, it was the second most downloaded system (nearly 140 million downloads) on

[1] http://httpd.apache.org
[2] http://xml.apache.org/xerces-j
[3] http://www.w3c.org
[4] http://java.sun.com/javase/
[5] http://xml.apache.org/xalan-j/
[6] http://argouml.tigris.org/
[7] http://www.omg/com
[8] http://www.gentleware.com/
[9] http://www.myeclipseide.com
[10] http://azureus.sourceforge.net/
[11] http://www.bittorrent.com/

Sourceforge[12].

### 3.2.2 Descriptive Statistics

Table 3 contains the average metrics values for the classes in each system. Most systems have classes that implement about 10 methods (NMD). Our two libraries have larger classes that generally offer more services. Xerces is the system that makes most use of object-oriented mechanisms like inheritance as shown by an average class depth (DIT) of 3. On the other hand, Azureus barely uses inheritance which indicates that it might be written in more of a procedural style (DIT=1.63).

| Metrics | Xerces | Xalan | Azureus | ArgoUML |
|---|---|---|---|---|
| CBO | 18 | 20.5 | 17.5 | 58.2 |
| DCMEC | 0.03 | 0.02 | 0 | 0.01 |
| DCAEC | 0.02 | 0.04 | 0 | 0.01 |
| LCOM5 | 0.62 | 0.5 | 0.53 | 0.45 |
| ICH | 20.1 | 9.35 | 11.4 | 12.6 |
| DIT | **3** | 2.7 | **1.63** | 2.8 |
| ABS | 0.21 | 0.01 | 0 | 0.01 |
| WMC | 38.6 | 27.1 | 18.9 | 19.9 |
| NMD | 12.7 | 9.35 | 8.45 | 8.2 |
| NAD | 6.7 | 5.7 | 4.9 | 3.7 |
| NI | 413 | 372 | 265 | 212 |

**Table 3. Mean values for structure metrics**

### 3.2.3 System growth

Some major studies of software evolution indicate that most commercial software systems' growth will eventually slow down and hit a plateau [14, 24]. We have consequently decided to describe the growth of our systems as presented in table 4. Detailed views are shown in figures 4(a), 4(b), 5(a), and 5(b). In these graphs, downloaded versions are represented as dot. Some versions were bug-fixes and were not included in our study.

| System | # Version collected | Growth (classes/month) | Growth (NI/month) |
|---|---|---|---|
| Xerces | 22 | 5.5 | 2077 |
| Xalan | 8 | 12.6 | 8671 |
| ArgoUML | 4 | 2 | 1270 |
| Azureus | 15 | 46 | 12808 |

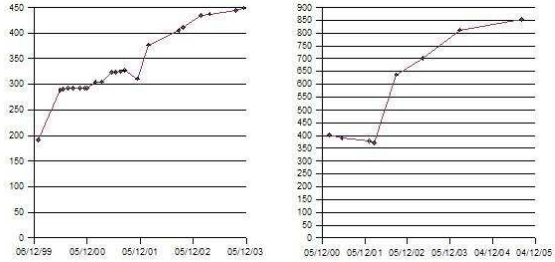
**Table 4. Growth overview**

The growth in Xerces and Xalan is illustrated in figure 4. These libraries have more than doubled in the period studied, and their growths were relatively steady. There are occasional drops in the number of classes due to restructurings.

ArgoUML's growth is shown in figure 5(a). Its growth has stagnated in the last few years as its popularity has waned. Even if it is still marginally used, it seems to resemble abandonware. Azureus (figure 5(b)) is the most active system, growing by a factor of 10.

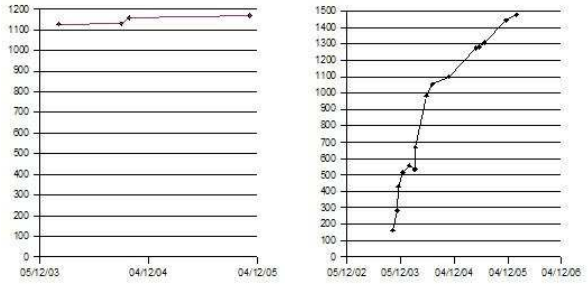
*Code changes.* In table 5, we can see that changes in systems are generally rare occurrences. Generally, about 5% of the code

[12]http://sourcefore.net



(a) Xerces (b) Xalan

**Figure 4. Growth of libraries (# of classes)**



(a) ArgoUML (b) Azureus

**Figure 5. Growth of applications (# of classes)**

base is modified between two versions. When looking at individual systems, Azureus is the system that has seen the quickest growth, but it is also the system where the code has seen the fewest changes. ArgoUML on the other hand is the system where that has stagnated in terms of number of classes, yet it is the one with the highest rate of code change. In order to have systems grow at a healthy pace, we must minimise the work involved in maintain its existing classes.

Figure 6 illustrates that changes are concentrated in a minority of classes. This concurs with Boehm's [3] assessment that rework costs are concentrated in a few key items. This power-law distribution of change is problematic from an experimental standpoint since we would like to have many examples of classes containing a large amount of changes to study.

| System | Change |
|---|---|
| Total | 5% |
| Xerces | 5% |
| Xalan | 6% |
| ArgoUML | 7% |
| Azureus | 4% |

**Table 5. System-level code change**

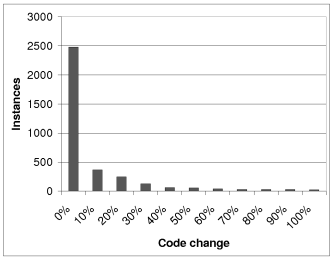



**Figure 6. Change distribution in Azureus**

| System | Cases |
|---|---|
| Xerces | 3,862 |
| Xalan | 3,168 |
| ArgoUML | 4,280 |
| Azureus | 13,084 |
| Total | 24,407 |

**Table 6. Number of data samples per system**

#### *3.2.4 Data Samples*

We decided to validate the three hypotheses at two different levels: at a general level and at a system level. Table 6 presents the number of evolutions analysed. An evolution is a transition between consecutive minor versions of a class. For example, a class that exists for N consecutive versions accounts for N - 1 evolutions.

At a general level, we built models with all of the cases for each type of system. In this setting, there should be decent coverage of the different possible evolutions. When we look at our numbers by system, all systems except for Azureus offer about the same number of evolutions (3,000 to 4,000).

### 3.3 Analysis Techniques

We have used two machine learning techniques to build our prediction models: regression trees and case-based reasoning (CBR). These techniques use descriptions of known evolutions to "learn" how to predict new unknown cases. Regression trees extract rules from our data set, and these rules are then applied to unknown cases. Case-based reasoning finds the cases in its data set that are similar to a new case, and adapts their values. These techniques were chosen because they are nonparametric and do not rely on a specific statistical distribution of variables.

#### *3.3.1 Case Representation*

Every class evolution is represented in the following form:

$Z = (X, Y) \in (\mathbb{R}_d \text{x} \mathbb{R})$ where $X$ is our vector of independent variables, and $Y$ is our dependent variable.

For H1 and H3, $X$ contains our structure metrics. For H2 and H3, $X$ holds our design change metrics. In every case, $Y$ is the relative code change, the value that our models are trying to predict.

Table 7 gives an example of three cases.

| | X | | | | Y |
|---|---|---|---|---|---|
| Class | LCOM5 | DIT | ... | PPMA | Chg. |
| CollectionIndex | 1 | 1 | ... | 0 | 0 |
| HTMLAnchor | 0 | 4 | ... | 0.1 | 0.17 |
| HTMLCollection | 0.12 | 1 | ... | 0 | 0.04 |

**Table 7. Case representation example**

#### *3.3.2 Regression Trees [4]*

This technique uses a divide-and-conquer algorithm to derive prediction rules from the data and organise them in the form of a tree. The tree's nodes represent rules while its leaves hold the values for the prediction. A clear advantage of using the technique is that it is a white-box algorithm that allows us to analyse the rules produced. We have used the regression trees as provided by Weka [26].

#### *3.3.3 Case-based Reasoning*

Case-base reasoning (CBR) [23] is a general problem solving technique that tries to adapt the solutions proposed in known cases to a new case. We have used the k nearest neighbour (kNN) algorithm provided by Weka[26].

To locate similar cases, we find the closest known cases using their Euclidean distance (equation 3) to our new case. To make sure the algorithm is unaffected by different metric scales, independent variables have been normalised in an interval of $[0, 1]$.

$$D(x, y) = \sqrt{\sum_{i=0}^{i<d} (x_i - y_i)^2} \tag{3}$$

where $x$ and $y$ are two cases. $x_i$ and $y_i$ are their independent variables.

In the study, we used the simplest case, a 1NN (k = 1). This configuration assigns the value of the nearest neighbour to a new case.

### 3.4 Validation

To evaluate our prediction models, we need some measures to indicate how accurate they are. Our models return predictions of the value of code change in a class evolution; a good model should return values that are close to the real ones. We split our data to build models with one part and test them with the other. Since we have measured the real values of change in our testing data, we can see how close our predictions are to reality.

In regression models, it is common to use a mean square error, but as the code change distribution is heavily biased towards 0 to such an extent this measure is inadequate. Instead, we have used its statistical correlation presented in equation 4.

$$Corr = \frac{S_{PA}}{\sqrt{S_P S_A}}, where\ S_{PA} = \frac{\sum (p_i - \bar{p}_i)(a_i - \bar{a}_i)}{n-1},$$
$$S_P = \frac{\sum (p_i - \bar{p}_i)^2}{n-1}, S_A = \frac{\sum (a_i - \bar{a})^2}{n-1} \tag{4}$$

where $a$ is the real value, $\bar{a}$ is the average value of $a$, $p$ is the predicted value, $\bar{p}$ is the average predicted value.

A correlation of 0 means that a model is not able to predict code change. The closer the correlation gets to 1, the better it is. Since our data sets contain minimally 3,100 cases and therefore exhibit high degrees of freedom, we accept a correlation of 0.5 as being statistically significant.

In order to maximise our use of our cases, we used ten-fold cross-validation to split our data into training and testing sets. This approach is common in machine learning and consists of splitting our data evenly into 10 groups. We then build 10 models using different combinations of these groups where nine groups are used for learning and one group to test. The total result is the average of all of our 10 models.

# 4. Results and Interpretation

The results of the study will be presented in two parts: a general comparison of libraries and applications, then a specific analysis of the systems involved.

## 4.1 General Models

Our general models were built combining the data from each type of system analysed. Table 8 presents the correlations for our different hypotheses. The high level of correlation (over 0.5) for the models that consider our design change metrics (H1) indicates that they are good predictors of the scope of change. The models measuring the impact structure of structure (H2) show an important difference between both types of systems. For libraries, structure (0.77) conveys more information than design change (0.60) while in applications, structure is not a significant factor (0.25). In both cases, the combination of both types of metrics produces more accurate models as shown by higher correlations. This also shows that structure affects how design changes are implemented. This is especially true in the case of libraries. Additionally, the results also show that the regression tree models generally gives better results.

| Type | Algorithm | H1 | H2 | H3 |
|---|---|---|---|---|
| Libraries | 1NN | 0.59 | 0.69 | 0.81 |
| | Reg. Tree | 0.60 | 0.77 | 0.85 |
| Applications | 1NN | 0.59 | 0.16 | 0.62 |
| | Reg. Tree | 0.64 | 0.25 | 0.70 |

**Table 8. Results on all systems (correlation)**

Results produced for hypothesis 3 using regression trees are presented in figure 7. The graphs show actual and predicted changes of evolutions in order of actual changes. They only contain 5,000 evolutions to minimise visual clutter. When a model returns well correlated predictions, the predicted values increase as actual values increase, as is the case with both models presented.

We can see that, changes values are relatively well predicted since predicted values tend to go up with the corresponding real values while leaving few outliers. There are however clear differences between applications and libraries. All four systems contain between 4 and 7 % code changes, but they don't seem to be distributed in the same manner. In libraries, changes seem to be localised in smaller classes (hence high relative code changes) while in applications they are not. Also, in figure 7(a), there is a small plateau at around 50% of change. This is due to changes to the existence of co-changes. Co-changes occur when a change is systematically propagated to multiple classes and are present in the two libraries but not in the applications.

## 4.2 System Models

In this section the models are built on our individual systems, it should therefore be possible to identify the characteristics of these systems that are involved in our predictions. The results are presented in table 9 which includes the average correlation for every hypothesis. Again, in all but one case, regression tree models give the best results.

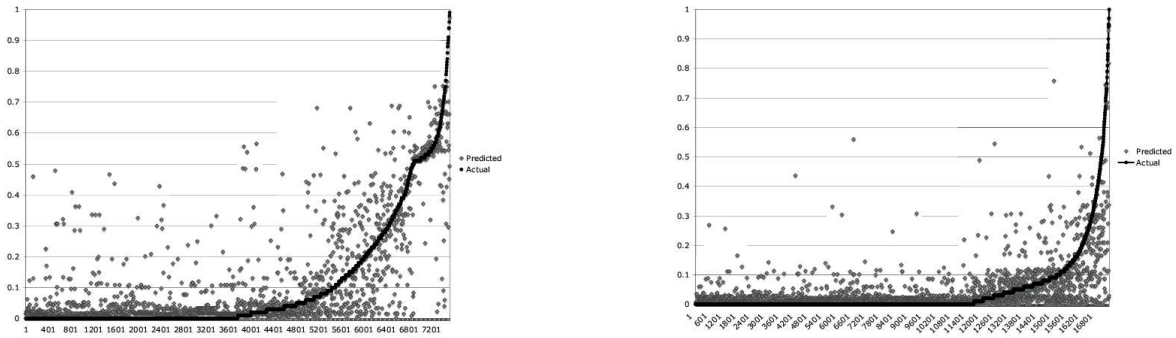
| System | Algorithm | H1 | H2 | H3 |
|---|---|---|---|---|
| Xerces | 1NN | 0.62 | 0.58 | 0.78 |
| | Reg. tree | 0.63 | 0.71 | 0.82 |
| Xalan | 1NN | 0.48 | 0.59 | 0.69 |
| | Reg. tree | 0.48 | 0.63 | 0.76 |
| ArgoUML | 1NN | 0.55 | 0.18 | 0.55 |
| | Reg. tree | 0.53 | 0.21 | 0.56 |
| Azureus | 1NN | 0.71 | 0.12 | 0.68 |
| | Reg. tree | 0.71 | 0.21 | 0.72 |
| Average | | 0.59 | 0.39 | 0.70 |

**Table 9. System-specific results (correlation)**

*Hypothesis 1: Design changes are sufficient to predict code change.* It is without surprise that most models built with our design change metrics show acceptable correlations. All systems (0.59 average correlation) except for Xalan (0.48) have validated our hypotheses. The poor performance is Xalan is due to the strong influence of structure (cohesion) in this system. The large changes occurring in low-cohesion classes in Xalan are not detected by using design changes. This is one of the main reasons for our models' bad performance. Amongst the other systems, Azureus produces the best models since it is the system that the least uses inheritance. Without inheritance, any change in design results in code change; meanwhile, in a well factored system, changes can be spread out throughout the inheritance tree.

*Hypothesis 2: Structure is sufficient to predict system change.* Our results show that code structure alone can be used to predict code changes in our libraries, Xerces and Xalan, but cannot predict them in our applications. We believe that there are two reasons for this. First off, some structures are indications that developers have used a mechanism to accommodate change. Then, there is the case where bad structure hinders maintenance and increases the general amount of code change in a class.

In object-oriented systems, inheritance is a key factor to accommodate change. In our libraries, stable functionality is generally defined in classes near the root of the inheritance trees while more volatile behaviour is defined near the leaves. Both Xerces




(a) Libraries (b) Applications

**Figure 7. Code Change Predictions (H3): Actual and Predicted Values (5000 point sample)**

and Xalan have rich domain models that use deep interface inheritance trees. For example, over 25% of the classes in Xerces are located at a distance of 5 from the root and most correctly predicted changes are located in these "deep" classes. In the regression tree models of the libraries, inheritance metrics are used in the first levels of the trees produced, thus denoting their importance. This supports the findings of Mao et al. [17] who also showed that leaf classes of a class are more likely to contain code changes than other classes. Even though ArgoUML uses inheritance, it does not implement a well-structured domain model which seems to be the key for predicting code change. Azureus barely uses inheritance and there does not seem to be other structural characteristics that predict change.

The cohesiveness of classes in Xalan also seems to be an important factor as it is used in the first level of its regression tree. It is reasonable to assume that classes that only implement a few functions will evolve with less effort. In this library, classes with low cohesion are generally changed a lot. Its classes that are the most changed, have a greater "Lack of cohesion" measure than the average (LCOM5 of 1.5 vs. 0.5). These classes should probably be reviewed by developers to make sure that they are properly factored.

It should be noted that the results of the general models (correlation improvement of 10%) are better than those built on their individual constituents. A reason may be that more examples improve the coverage of evaluated cases and we prevent overfitting our models to only one system. An example is that in the regression model for libraries, the first rule verifies if a class is abstract while this test is not present does not exist in the system-level models.

Finally, we found that a few classes in our libraries seem to be affected by size and coupling. The only exception is Xerces where very large classes with very little coupling frequently contain small changes. We believe that all classes need to collaborate with others. When a class contains too many responsibilities, this affects negatively its maintenance. Since these classes change in almost every releases and there are no signs that these classes were designed to be changed, these classes should also be inspected.

In short, the H2 hypothesis was found valid for libraries, but invalid for applications.

*Hypothesis 3: the combination of both structure and design change metrics is a better predictor of code change.* In general, we have seen that models including both input types produce better models since average correlations improve from 0.59 to 0.70. Specifically, it is safe to say that our libraries have shown that combining structure and design change metrics are promising. Xalan shows the best improvement as it adequately combines the contributions of both previous models. It is hard to imagine better results without including any information about the projects' processes and resources.

For our two applications, the addition of structure metrics to design change barely improves correlations and even drops in one model. This is not necessarily an indication that combining our metrics does not work: The values predicted are generally closer to actual values by about 10%, but this is not taken into account by our correlation measure. This improvement is mostly due to size metrics: when a small class sees its design change, predictions tend to be closer to actual values.

## 5. Related Work

Much research focusing on software libraries is concerned with evaluating the value of reuse, but there is little in terms of quality. Melo et al. [19] showed that using libraries lower fault density in a system and improve productivity. Mohagheghi et al. [20] compared fault-density of reused and non-reused components in an industrial setting. They concluded that reused components are generally less fault-prone and stable than non-reused components. In this study, we didn't focus on correctness, but rather we looked at the capacity of a library to change which includes perfective and

adaptive maintenance.

There exists much work studying how changes affect software quality, but to our knowledge, our work is the first that examines the nature of the systems analysed. A few papers look at the effect of change on fault-proness [21, 22] while others use change as a measure of the size of corrective maintenance [13]. For quality models specifically predicting code changes, Li et al. [16] show that object-oriented structure is a predictor of general maintenance effort (absolute code change), but the results were not generalisable since it concerns small academic systems. Mao et al. [17] show that the effort to adapt a framework to another environment depends on its structure, particularly inheritance. Xing et al. [29] categorise changes in a software library using different pattern identification techniques. Our study differs from these three since we focus on the nature of the evolving systems. Our models are different since they include a measure of the external forces causing code change (design change). Furthermore, by studying a few systems separately and combined, we believe that our results are more generalisable.

## 6. Conclusion and Discussion

In this paper, we have presented a study of the evolution of object-oriented software and the predictability of change in individual classes. Specifically, we looked at the differences between libraries and applications. We used machine learning techniques to build prediction models and our experiments were guided by three hypotheses to verify if code change could be predicted by design changes (H1), code structure (H2), and whether the combination of both factors (H3) improved predictability.

Overall, our models to estimate code churn (and hence maintenance effort) give predictions which are highly correlated to actual values, but the value of structure depends on the nature of the system analysed. Our findings can be summarised as follows:

- H1: As expected, design change is a good predictor of code change especially when the structure of program does not convey any relevant information. However, this is a very coarse measure and it tends assigns the same value (global average) to a large number of classes. This is evident when analysing classes with no changes in interface;
- H2: Structure metrics are good indicators of code change, but only for systems with a high degree of object-orientation as measured by inheritance metrics. This was verified in both libraries studied, Xerces and Xalan. Both when they were analysed individually and together;
- H3: Structure metrics and design change provide complementary information. The combination of both give better results than using them individually;
- The most important structure metrics for prediction purposes were: DIT (depth in the inheritance tree) and LCOM5 (lack of cohesion). Cohesion was especially important when evaluation the Xalan library;
- Models built using the regression tree learning technique were generally better than the nearest neighbour technique (with $K = 1$). This is due to the capacity of trees to ignore unimportant attributes;
- In most cases, some models built by combining data of different systems produced better results than system-specific models. A reason for this might be that some useful prediction rules could not be learned because there were not sufficient examples in individual systems.

As for any empirical study, several factors threaten the general validity of the results [27]. The main issue that must be considered is the representativeness of the chosen systems. All systems have been developed for many years and all four have been included in commercial products. The 24,000 cases (class evolutions) considered offer sufficient variations in structure and design/code changes. A significant threat is the application domain. Indeed, three of the four systems are development tools. Furthermore, a future study should be conducted to include additional systems of both types.

We use correlation as a measure of the performance of the prediction models. Although, it does not give a precise idea on the prediction errors, it is a good indicator if the model predicts correctly the magnitude of change. Moreover, an alternative measure, the absolute relative error supports the results shown.

For our future work, we have already started to look at our problem using classification techniques to discriminate between different classes of change and see how it applies to different types of software systems. We are also considering change as a multidimensional phenomenon. As a system grows, its different classes will change in many possible ways. We have started measuring different types of change and using clustering techniques to automatically discover evolution patterns on systems.

## 7. Acknowledgements

The study presented in this paper was partially funded by the Fonds québécois de recherche sur la nature et les technologies (FQRNT) and the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors want to thank Jean Vaucher and Naouel Moha for their valuable comments on early versions of this paper.

## 8. References

[1] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering*, 22(10):751–761, 1996.

[3] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10):1462–1477, 1988.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

[5] L. C. Briand, P. T. Devanbu, and W. L. Melo. An investigation into coupling measures for C++. In

*International Conference on Software Engineering*, pages 412–421, 1997.

[6] L. C. Briand, J. Wüst, and H. Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001.

[7] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM Press.

[8] D. Grosser, H. A. Sahraoui, and P. Valtchev. Predicting software stability using case-based reasoning. *Proceedings of Automated software engineering*, 00:295, 2002.

[9] Y.-G. Guéhéneuc. A reverse engineering tool for precise class diagrams. In J. Singer and H. Lutfiyya, editors, *Proceedings of the* $14^{th}$ *IBM Centers for Advanced Studies Conference*, pages 28–41. ACM Press, October 2004.

[10] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[11] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Softw. Eng.*, 25(4):493–509, 1999.

[12] T. Khoshgoftaar, E. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. volume 00, page 364, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[13] T. M. Khoshgoftaar and R. M. Szabo. Improving code churn predictions during the system test and maintenance phases. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 58–67, Washington, DC, USA, 1994. IEEE Computer Society.

[14] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[15] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Journal of Soviet Physics - Doklady*, 10(8):707–710, feb 1966.

[16] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Software Systems*, 23(2):111–122, 1993.

[17] Y. Mao, H. Sahraoui, and H. Lounis. Reusability hypothesis verification using machine learning techniques: A case study. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 84, Washington, DC, USA, 1998. IEEE Computer Society.

[18] McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2:308–320, 1976.

[19] W. L. Melo, L. Briand, and V. R. Basili. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report CS-TR-3395, University of Maryland, 1995.

[20] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[21] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 24, Washington, DC, USA, 1998. IEEE Computer Society.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.

[23] R. C. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, New York, NY, USA, 1983.

[24] W. M. Turski. The reference model for smooth growth of software systems revisited. *IEEE Trans. Softw. Eng.*, 28(8):814–815, 2002.

[25] L. G. Votta and A. Porter. Experimental software engineering: a report on the state of the art. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 277–279, New York, NY, USA, 1995. ACM Press.

[26] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, June 2005.

[27] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[28] R. Wolverton. The cost of developing large-scale software. *IEEE Trans. on Computers*, C-23(0018-9450), 1974.

[29] Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Trans. Software Eng.*, 31(10):850–868, 2005.

[30] B. S. L. Y. S. Lee. Measuring the coupling and cohesion of an object-oriented program based on information flow. pages 81–90, Maribor, Slovenia, 1995.

[31] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.