Understanding Object-Oriented Architecture Evolution via Change Detection

Zhenchang Xing and Eleni Stroulia
Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
{xing, stroulia}@cs.ualberta.ca

Abstract

Understanding the software architecture of a system and the process by which it has evolved to its current state is an important task that software developers are often faced with. It becomes relevant when one needs to assess a system for the purpose of adopting it in a new context, or to further develop it to meet new requirements and change requests. In this paper, we describe our work on analyzing and understanding the evolution of an object-oriented application at the class-design level. We introduce a structure-matching algorithm for comparing two or more versions of an architecture represented in UML (XMI). The algorithm produces a "change tree" that reports the differences of the compared versions in terms of class/field/method additions, deletions, moves, and renamings. Analysis of a series of change trees corresponding to a series of versions can reveal interesting and useful information about the evolution history of the application architecture, such as evolution styles, class evolution types, change patterns, etc. In this paper, we discuss the algorithm, the change-tree data structure, and the architecture-evolution analysis, and we report on two case studies evaluating our approach.

1 Introduction

Change is ubiquitous during the lifecycle of a software system: software is often developed using an evolutionary process, such as extreme programming for example, and after its release and deployment it is evolved to fix defects, meet customer-driven functionality enhancements, and adapt to changes in the deployment environment. As a result, the actual design of a system is due, to a great extent, to its evolution history.

Understanding the design rationale and its evolution is an important aspect of the overall software-understanding problem [9,14,7,18] that developers, maintainers, and managers face. They often need a high-level understanding of the complete system and in-depth information on selected components and how they have evolved to their current state. Consider, for example, the case of a new developer, who has just joined a project and has been assigned the task of modifying and extending the behavior of a system class. She inspects the design history of the subsystem she is working on and she discovers that this subsystem has been regularly refactored to "inline" methods and classes; therefore she hypothesizes that improving run-time performance may have been a design objective and, thus, she may decide to modify the existing class without subclassing it, to keep in synch with the spirit of the previous modifications of the system.

As another example, consider a software inspector who wants to identify hot spots over the lifespan of software system. By comparing a set of subsequent versions, he may find out that a few classes have been substantially changed in every new version, irrespective of what features were modified in this version. This information can help him focus his examination into the source code of those classes, to determine what reason(s) may have caused this problem, such as the fact that these are giant classes that implement too many functionalities, and to suggest potential remedies, such as design patterns that attack the problem under investigation.

Finally, consider an instructor, supervising a set of team projects for her software-engineering capstone-project course. She needs to closely monitor the development process of all the teams to make sure that they are on the right track, and to intervene in a timely and informative fashion if not. She needs to know what parts of the system are being changed and how, and if there are any "interesting" events or trends, such as, for example, slow or explosive code growth or particular refactorings.

The objective of this work is an automated tool for supporting design-level evolutionary analysis. Research in understanding software evolution has traditionally focused on analysis and visualization of information extracted from version-management systems [9,11,14], or software metrics of size and complexity [6,7,17]. The innovation of our methodology lies in the fact that it does not examine the development documentation and release history of the application or the evolution of its code metrics, but instead, it relies on recovering and analyzing the modifications on the software design from one version to the other through comparison of class hierarchies. Further analysis of the recovered design-level modifications results in interesting insights regarding the evolution history of the software system under analysis.

Our tool compares designs to recover their differences. Comparison of original designs against designs reverse-engineered from code could reveal discrepancies between the designers' intent and the actual implementation. On the other hand, comparison of a sequence of reverse-engineered designs, corresponding to a sequence of software-system versions, could recover the evolution profile of individual application classes, identify transformations brought about by refactoring, and characterize the nature of evolution of the application design.

The remainder of the paper is structured as follows. In section 2, we place this work in the context of previous related research. In section 3, we present the overall methodology of our approach. In section 4, we discuss the core algorithm of the change-detection process. In section 5, we discuss the rationale of the analysis of the recovered changes. Section 6 presents the implementation of our method including its visualization instruments. Two case studies illustrating our approach are discussed in section 7. Finally, we conclude our discussion with a summary of the lessons we have learned to date and our plans for future work.

2 Related research

There already exists a substantial body of literature on the subject of "software-evolution understanding." Eick et al. [9] analyze the change history of the code, which is assumed to reside in a version management system. Several derived attributes, i.e., "Code-Decay Indices", are calculated and the corresponding fault potential and change effort can be predicted as a function of these indices through regression analysis. The objective of this research is mainly to support project management so that code decay is delayed. The same team also developed tools [10,11] for visualizing software statistics at source code line level, and change data contained in version management system, such as developer, size, effort, etc. However, they do not visualize any "structural" differences of UML models of software system.

Gall et al. [14] use information in the release history of a system to uncover logical coupling among modules. Their method aims mainly at understanding co-evolution, i.e., whether there exist modules that are always changed together. In the similar vein, Zimmermann et al. [25] detect the fine-grained coupling between program entities like methods and fields. However, unfortunately, such documentation is not always readily available, since, more often than not, the rationale for a new version is not documented. Even worse, even when such documentation exists, it is seldom kept in synch with the code modifications, and therefore it is an unreliable source of system changes.

Demeyer et al. [7] define four heuristics based on code-size and inheritance metrics to identify refactorings that might have occurred on a piece of code. The refactorings identified by this work are grouped into three general categories and no concrete ones, i.e., the ones described in [13], are identified.

Demeyer et al. [6] and Lanza [17] describe how to use a simple two-dimensional graph to convey the implicit information of software metrics of object-oriented entities. Their work does not directly visualize the changes of the object-oriented entities themselves; instead the analysis is focused on the evolution of code metrics.

Collberg et al. [5] focus on the visualization of the evolution of software using a temporal graph model. They do not compare designs to surface the structural differences through a structure-matching algorithm.

Emden et al. [12] present a tool for detecting and visualizing code smells based on the analysis of extracted facts of program structure, while our work focuses on the "evolution smells" at the design level.

Egyed [8] has investigated rule, constraint based transformation and comparison approach for consistency checking between UML diagrams when developers add new information to system model or modify existing ones.

Selonen et al. [21] have also developed a method for UML transformations, including differencing. However, none of these projects have explored the product of their UML-diagram transformations in service of design analysis and evolution understanding.

Finally, the general tree-to-tree correction problem has been studied extensively [3], and has been applied to show differences between XML data [28]. The major difference between these general algorithms and *UMLDiff* is, *UMLDiff* takes into account the structural syntactic information contained in the class model of application, and it can identify the "move" of object-oriented entities, which enable us to identify perfective changes that cannot be identified from documentation like revision archives.

3 The design-evolution understanding methodology

The primary **data input** of our design-evolution understanding method is the design of the application under analysis, as captured in its UML class diagrams. Such class diagrams can be either produced in the software-design phase or they can be reverse engineered by the application code, using any of the currently available roundtrip-engineering tools [29, 30]. Such tools export the reverse engineered diagrams in XMI (UML1.3), which is the assumed input-data representation in our process for design-evolution analysis. Given the XML-based syntax of the input data, it is parsed into a class-hierarchy tree. Multiple-inheritance is handled by duplicating the class node (not including its children) under each of its super classes.

The basis of the **change-detection process** is a heuristic tree-comparison algorithm that recognizes, element "additions", "deletions", "moves" and "renamings". This tree-comparison algorithm essentially implements a UML differencing operation that can surface structural modifications to the application classes and interfaces, their attributes, their methods and their specialization-generalization relations.

The result of the comparison between two application versions is represented in a **change tree. Aggregate information** can then be extracted by examining a sequence of such change trees, to characterize the evolution history between two non-consecutive versions of the application. By analyzing sequences of changes trees, we can recover information about software evolution at three different levels:

- At the system level, we can identify different evolution phases, such as functionality extensions vs. refactorings, and "interesting" events through the application evolution history.
- At the class level, we can recognize different types of classes according to their evolution profiles, such as continuously modified classes vs. legacy classes, for example.
- At the "change-tree level" we can identify various change patterns, such as co-evolution and refactorings, for example.

Finally, the collected design-evolution information can be **visualized** to present different views of the application evolution to the interested developers.

By examining and analyzing the above information, one is able to get a quick overview of the whole application history, to assess if the application has been evolving in a consistent way, to identify interesting parts of the design or parts that may suffer from "bad smells", such as code duplication and parallel inheritance hierarchies. Furthermore, one can obtain plausible answers to a set of interesting questions:

- What classes were changed and which methods and/or fields are added, deleted, and moved in which version?
- Are there unusual incidents over the life cycle of the system?
- Which version(s) mainly involve additive or perfective changes?
- What functionality is the focus of several consecutive versions?
- Are there classes that have the same lifespan as the whole system but never change at all?
- What classes change frequently or do not change over several versions?
- What classes exist for only a very short time?
- Does class rapidly grow or shrink from one version to another?
- Are there sets of classes that change together?
- What refactorings were made?

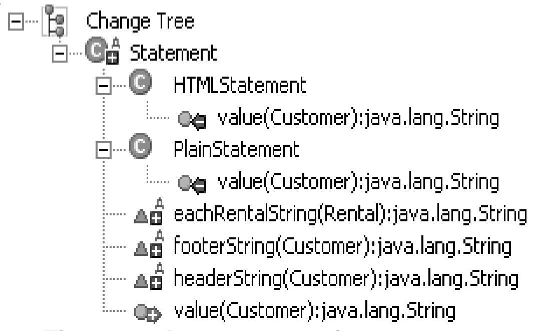
4 Class-hierarchy change detection

In this section, we discuss in detail the change-tree data structure used for representing the differences between the class models of two versions of a software application and the tree-differencing algorithm for generating such trees, given the XMI representations of two such models.

4.1 The change tree

In our approach, we have cast the problem of detecting the class-model changes between two versions of an object-oriented software application as a graph-difference problem, since class models can be viewed as specific types of directed graphs. To make the problem more tractable – the general problem of comparing two directed graphs is NP-complete – we have limited our initial exploration to considering only the inheritance trees of the class model, ignoring all other relations, such as association and composition. This decision was also motivated by the fact that most UML reverse engineering tools do not do a good job at inferring such relations [16].

Given two class models, corresponding to two different software versions, represented in XMI, the first step of the process is to parse the class hierarchies they contain into two labeled tree structures. The target representation contains the application classes and interfaces, their attributes, their methods and their specialization and implementation relations. Next, the tree-differencing algorithm, described in the next subsection, is applied to the forests and identifies the *after-before* changes between them. The result is represented as a change tree, i.e., a tree of delta operations, which if applied to the earlier version (before) would result in the later version (after). Figure 1 diagrammatically depicts an example change tree, which will be further discussed in section 5. Its visualization will be explained in section 6.1.



```
Change Tree
  Statement
    HTMLStatement
      value(Customer):java.lang.String
    PlainStatement
      value(Customer):java.lang.String
    eachRentalString(Rental):java.lang.String
    footerString(Customer):java.lang.String
    headerString(Customer):java.lang.String
    value(Customer):java.lang.String
```

Figure 1: An example of change tree

4.2 The UMLDiff algorithm

The overall problem of detecting and representing changes to data is important for version and configuration management. It is an active research area on its own in the area of data management. Probably the most well known algorithm for textual comparisons, *GNU diff*, was discussed as the string-to-string correction problem using dynamic programming in [23]. Used in the context of code differencing, it reports changes at the code line level.

As more data and documents are stored in XML format, some sophisticated version control systems include XML-aware features to handle XML documents. However, these report changes as "XML element modifications" ignoring the domain-specific semantics of these nodes. Take the XMI as an example. When a class implements a new interface, the general XML-differencing tools would only report that a set of xml-nodes was inserted, but not the implementation of a new interface, since it does not understand the XMI semantics.

Furthermore, such change representations do not necessarily correspond naturally with the developers' intuition about change. For example, when an attribute or method is moved from one class to another, the reported changes are often reported as two separate activities, one or more lines of code (xml-nodes) are removed from one file and an equal number of lines of code (xml-nodes) are added to the other one. It is difficult to detect and represent the "move" of these lines (xml-nodes), which is exactly what would be preferable to the developer.

Recognizing changes at this higher level of abstraction and taking into account the UML-specific semantics of XMI documents is exactly the motivation of this work. If we rely on the structural syntactic information captured in the class hierarchy, we could identify such activities as "moves"; this is because the granularity of change operations is larger and correspondences between additions and deletions could be explored to uncover higher-order operations such as "moves".

The *UMLDiff* algorithm is tailored from tree-to-tree correction algorithm [4] and is applied in the context of class hierarchy-tree differencing. The algorithm can identify many interesting structural changes of the class model, such as the deletion, insertion, and change of signature of classes, interfaces, methods, and fields, the move of methods and fields, and the renaming of methods. However, not all possible changes can be automatically detected. To determine a "class renaming", for example, it would be necessary to check if most of the subclasses, methods, and fields of two differently named classes are the same, and that would be both time consuming and also an "unsafe" inference. Furthermore, "field renamings" are not considered. That is because *UMLDiff is* based on name/attribute schema, and it is difficult to determine a "field renaming" without information about the field's semantics. An interactive step can be added after the analysis process to identify such changes.

Attention should also be given to "move". In general, tree-differencing algorithms, like [3], do not consider "move" as a primitive operation. But in the context of software evolution, where local transformations, such as refactoring, frequently involve moving elements from one class to another, recognizing such "moves" is essential. A "move" operation often represents the redistribution of information or the reorganization of the class hierarchy, modifications that are usually part of perfective changes that are intended to improve the developer's ability to maintain the software without altering functionality or fixing faults. Thus, it is important that we could recover the method/attribute movement when analyzing software evolution, in order to recognize "perfective maintenance" phases in the software life cycle.

The outline of *UMLDiff*, our class-hierarchy comparison algorithm, in shown in Figure 2. The algorithm takes as input two class hierarchy trees, T_1 and T_2. The algorithm exploits the XMI node semantics: it

```
UMLDiff(T1, T2) {
    /*T1 and T2 are the roots of class hierarchy
    trees*/
    MatchTwoTrees(T1, T2, Not_Inner_Class);
    MatchTwoTrees(T1, T2, Not_Inner_Interface);
    ChangeTree = GenerateChangeTree(T1, T2);
}

MatchTwoTrees(R1, R2, Label) {
    /*R1 and R2 are the roots of two (sub)trees. This
    function matches two trees rooted at R1 and R2
    respectively*/
    M = match(R1, R2, Label, equal_func);
    M = M + match(R1, R2, Label_Unmatched,
    equa1l_func);
    For each pair (n1, n2) in M {
        match(n1, n2, Field, equal_func);
        match(n1, n2, Field_Unmatched, equal1_func);
        match(n1, n2, Method, equal_func)
        match(n1, n2, Method_Unmatched, equal1_func);
        match(n1, n2, Method_Unmatched, equal2_func)
        match(n1, n2, Interface_Impl, equal_func);
        MatchTwoTrees(n1, n2, Inner_Class);
        MatchTwoTrees(n1, n2, Inner_Interface);
    }
    match(R1, R2, Field_Unmatched, equal_func);
    match(R1, R2, Method_Unmatched, equal_func);
}

match(r1, r2, Label, equal_func) {
    /*r1 and r2 are the roots of two (sub)trees. This
    function matches nodes with Label in two trees
    rooted at r1 and r2 respectively using equal_func*/
    M = null;
    Add (r1, r2) to M; //match two roots
    L1 = chain(r1, Label); L2 = chain(r2, Label);
    M = M + lcs_func(L1, L2, equal_func);
    For each unmatched node x∈ r1 with Label
        For each unmatched node y∈ r2 with Label
            If equal_func(x ,y) Then Add(x, y) to M
    Return M;
}

GenerateChangeTree(T1, T2) {
    Duplicate T2 into changeTree;
    Mark all unmatched nodes of T1 "delete";
    Visit the nodes of T1 in breadth-first order
        Let x be the current node, y be the parent of x;
        Let z be the partner of y in changeTree
        If x is "delete" Then Attach x under z;
        If x is "matched" Then
            Let w be the partner of x in changeTree;
            Let v be the parent of w;
            If z ≠ v Then
                Mark x "movesrc"; Mark w "movetrg";
                Attach x under z;
    Mark all unmatched nodes of changeTree "insert";
    Remove the total matched nodes of changeTree;
    Return changeTree;
}
```

Figure 2: The change-detection algorithm

does not attempt to match nodes of different types, for example, Method and Field, Interface and Method, etc. For each matched pair of Class or Interface nodes, the algorithm attempts to match their methods, fields, implemented interfaces, and inner classes. That is, the algorithm avoids unnecessary matches between arbitrary pair of methods contained in class hierarchy trees.

The basic idea of the function *match* is to use the longest common subsequence (LCS) routine [20] to match nodes of specific type of object-oriented entities, such as field, for example, in two (sub)trees. It first obtains two node sequences by chaining together all nodes with a given type in two trees rooted at r_1 and r_2 respectively, and then calls the LCS routine, *lcs_func*. The function *lcs_func* computes the LCS of two sequences in time $O(ND)$, where $N=|S_1|+|S2|$ and $D=N-2|LCS(S1,S2)|$. It takes three parameters: two sequences L_1 and L_2 to be compared, and an equality function *equal_func(x,y)* used to compare $x \in L_1$ and $y \in L_2$ for equality. Three equality functions are defined as follows:

- *equal_func:* returns TRUE if the value of all attributes of two nodes are the same;
- *equal1_func:* returns TRUE if the attribute "name" of two nodes are the same, but the value of some other attributes are different. It is used to identify "change signature of entity".
- *equal2_func:* returns TRUE if the value of all attributes but "name" are the same. It is used to identify "rename method".

During the matching phase, the algorithm annotates each node in T_1 and T_2 with the "memory (status)" of the type of changes that should be applied to it, to transform T_1 to T_2. Finally, the function *GenerateChangeTree* is used to generate the change tree, T_2-T_1, after matching two class hierarchy trees. It first duplicates T_2 into the initial change tree, then traverses T_1 in breadth-first order to copy "delete" nodes and "movesrc" nodes into the change tree, and finally traverses the change tree in post-order fashion to delete completely matched nodes, i.e., the nodes that are marked as "matched" and so are all their descendents. The resulting change tree represents a set of structural modifications, which if applied to T_1 would produce T_2.

5 The design-evolution analysis

A sequence of change trees between subsequent versions provides an audit trail of the design changes of the class model when software system evolves. They are conveniently saved as XML files. Consequently the powerful tools, like XSLT and XPath, can be used to process them for further information extraction and design-level evolution analysis.

Evolution phases and styles. These change trees are processed further to gain an insight into the overall evolution process of software system or its individual classes.

Based on this analysis, one can easily identify different phases, such as *growth*, *maintenance*, and *steady going*. In growth phases, the change activities are mainly the addition of new class, interface, method, and field. The maintenance phases contain much method and attribute movement. That is, the major activities of maintenance phases often involve the reorganization of class hierarchy, the redistribution of information, etc. Such activities are usually aimed at improving the system's clarity and maintainability. The *steady-going* phases are the phases that have relatively few change activities, when most of the software system or classes are stable.

Furthermore, a method similar to the one described in [2] is used to reveal evolution styles, such as constant small modifications, occasional large modifications, etc. [2] classify the software evolution into different styles based on the result of phasic analysis.

Growth spurts and unusual incidents. We can identify individual interesting versions of software system or classes, such as versions with aggressive growth spurts, unusual sharp increase or decrease in size, and so on.

By *growth spurts*, we mean that the amount of changes made to the system or class is relatively high in these versions. Particular attention should be given to versions with such aggressive growth spurts. They usually imply big expansion in system features and/or functionalities. Sometimes the overall "quality" of the system design may deteriorate due to such expansions. For example, the development team may not realize the commonalities that are being created between classes as similar features are implemented in multiple classes; in such cases, an inheritance-based structure is usually imposed later to extract the commonalities in the ancestor classes and encapsulate the variability in their descendants [13].

The *unusual incidents* represent anomalies in the software development process, for example, an unusual increase in size of system followed by a sharp decrease.

Class-evolution types. We can classify classes according to their change profiles. For example, a class can be defined as *short-lived*, if the interval between the class creation and deletion is less than a user-defined constant. We have identified several interesting types of class-evolution profiles, described below, and we have also implemented a set of corresponding XSLT programs to identify classes of these types in a system.

An *active* class keeps being modified over several versions. Many changes occur, which may be the additions of functionality, the removal of obsolete features, and refactorings. Active classes are hotspots from the view of software evolution. They may be giant classes, whose functionality should be fanned out, or incohesive classes, resulting from bad design choices, that need to be improved.

An *idle* class rarely changes after it is added into the system. They could be root classes, well-designed classes, or stand-alone features, but they could also be dead code.

Particular attention should be given to these classes whose lifespan is the same as that of the software system. They may have already been disused but no one "dares" to remove them.

A *short-lived* class exists only in a few versions of the system and then disappears. They may have been used to prototype a feature or to test another class, etc.

A *rocket* class sharply increases its size at certain point of evolution. Sometimes, developers tend to write code before they have figured out where is the best place to put it. Thus, we should examine those classes since they may introduce bad smells, such as code duplication, into the system. The s*hrink* class is the opposite of *rocket* class. Its size sharply decreases at certain point. If a class lost most of its functionality and is not doing very much, an "Inline class" refactoring can be applied to move all its features into another class and delete it [13].

A *die-hard* class is a class that is removed from the system but most of its functionalities are moved to other classes. A *legacy* class is just the opposite of *die-hard* class. Such classes are added into system at certain point of evolution, but most of their attributes and methods are moved in from other classes. The *die-hard* and *legacy* classes represent evidence of redistribution of functionality or the reorganization of class hierarchy.

Note that these class-evolution profiles are only the most distinct ones. More categories may be possible to identify with further analysis. Finally these categories introduced here are not mutually exclusive, i.e. a class can be active over several versions and then become a die-hard class eventually.

Co-evolving classes. The change trees record what classes has been modified (including creation and deletion) in which versions. A simple XSL stylesheet can collect information like Table 1, from change trees. The black dot represents that class has be changed in particular version.

Table 1: Class change history

	Class A	Class B	Class C
Version 1		•	•
Version 2	•		•
Version 3	•		

Using this information as a dataset, we can apply data-mining analysis techniques [1] to mine co-evolving classes. In our implementation, we programmatically use Weka [31], a Java library of machine learning algorithms that implements [1], to discover co-evolving classes that have common change behaviors. Co-evolution represents potential dependencies among classes, which are not evident in source code. They may point out potential structural shortcomings where the refactoring opportunities exist. For example, the original design of a software system follows the MVC model. But after several versions of implementations, we find out several classes in presentation layer often change together with a few classes in data model layer. That may reveal the high coupling between presentation and data model layer, which means that the implementation deviates from the original design. A "Separate presentation from data model" refactoring could be applied to improve the cohesion and reduce the coupling. We can also utilize the co-evolution relationship to advice preventive-maintenance activities. For example, if three classes are often changed together, when the developers modified two of them, we could recommend him to check out if they also need to make some changes to the third one.

Change patterns. The class-hierarchy trees represent inheritance relations among the classes of a software system. The change trees reveal the modifications to the class hierarchy trees. The change tree in Figure 1 corresponds to the differences between version 27 and 28 of the extended refactoring sample from M. Fowler's book [13] as found in [26]. In version 28, a new abstract class, "Statement", was created with three newly created abstract methods, "eachRentalString", "footerString", and "headerString". The "value" methods of its two subclasses, "HTMLStatement" and "PlainStatement", were pulled up into the new class "Statement". . This change tree represents the modifications to class hierarchy after an "Extract Superclass" refactoring, which is described as follows: "if you have two classes with similar features, then create a superclass and move the common features to the superclass [13]". We have developed XSLT programs that implement such heuristics to identify the following types of refactorings contained in change trees:

- Collapse hierarchy, Inline class;
- Extract class, Duplicate observed data;
- Extract superclass/subclass;
- Extract interface;
- Form template method;
- Replace type code with subclass;
- Pull up/down method/field;
- Move method/field;
- Add/remove parameter, Rename/Hide method.

6 UMLDiff in Eclipse

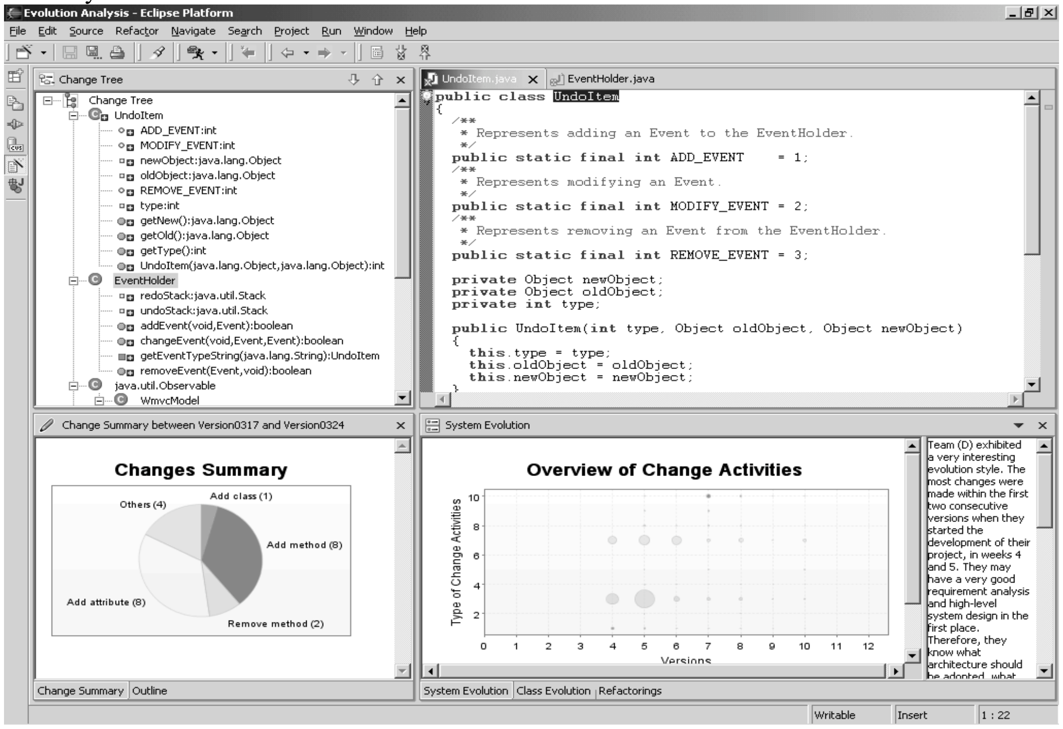
The design evolution analysis methodology presented in this paper is implemented as an Eclipse [27] plugin, an integral part of JRefleX project [24], whose goal is to develop a set of tools to monitor the collaboration process of software teams [18] and to aid the understanding of changes in software design.

6.1 Visualization[1]

In addition to the *UMLDiff* algorithm and the heuristics analysis XSLT programs we have already discussed, the plugin includes several visualization instruments to present the change trees, the aggregate data, and the

[1] The full color figures in this section can be downloaded from www.cs.ualberta.ca/~xing/screenshots.zip.

design evolution analysis results.



```
Change Tree
  UndoItem
    ADD_EVENT:int
    MODIFY_EVENT:int
    newObject:java.lang.Object
    oldObject:java.lang.Object
    REMOVE_EVENT:int
    type:int
    getNew():java.lang.Object
    getOld():java.lang.Object
    getType():int
    UndoItem(java.lang.Object,java.lang.Object):int
  EventHolder
    redoStack:java.util.Stack
    undoStack:java.util.Stack
    addEvent(void,Event):boolean
    changeEvent(void,Event,Event):boolean
    getEventTypeString(java.lang.String):UndoItem
    removeEvent(Event,void):boolean
  java.util.Observable
    WmvcModel
```

```
public class UndoItem
{
  /**
   * Represents adding an Event to the EventHolder.
   */
  public static final int ADD_EVENT    = 1;
  /**
   * Represents modifying an Event.
   */
  public static final int MODIFY_EVENT = 2;
  /**
   * Represents removing an Event from the EventHolder.
   */
  public static final int REMOVE_EVENT = 3;

  private Object newObject;
  private Object oldObject;
  private int type;

  public UndoItem(int type, Object oldObject, Object newObject)
  {
    this.type = type;
    this.oldObject = oldObject;
    this.newObject = newObject;
  }
```

Team (D) exhibited a very interesting evolution style. The most changes were made within the first two consecutive versions when they started the development of their project, in weeks 4 and 5. They may have a very good requirement analysis and high-level system design in the first place. Therefore, they know what architecture should be adopted, what

Figure 3: Evolution analysis perspective

Tree view. A change tree, its subtrees, and the identified refactorings can be represented naturally in a tree view, like the one shown in Figure 1, the top-left corner of Figure 3, and the right-hand side of **Error! Reference source not found..** The different icons represent the different object-oriented entities, "class", "interface", "method", and "field" respectively. The top-right adornments show the modifiers of the object, for example, "abstract", "static", etc. The bottom-right adornments represent the status of particular object. It can be plus sign for "insert", minus sign for "delete", filled triangle for "rename", empty triangle for "change signature", arrow with minus sign for "move source", arrow with plug sign for "move target". The tree view presents the developers the detailed structural modifications to the class model of software system.

Evolution matrix and histogram. The evolution matrix, like the one in the bottom-right corner of Figure 3, provides a quick overview to the overall history of evolution of the system or individual classes through its development lifecycle. Each column represents a version of the software, while each row represents the different types of changes. Rows 1 to 10 show the following: ??a newly created class (1), a deleted class (2), a newly created method (3), a deleted method (4), a moved method (5), a renamed method (6), a newly defined attribute (7), a deleted attribute (8), a moved attribute (9), and a signature change (10). The area of the bubble represents the amount of such types of changes. Thus, a bubble of size *s* at the *(x,y)* point in the matrix indicates that *s* number of changes of type *y* happened between version *x-1* and *x*.

The histogram, like the one reported Figure 4, depicts the change profile of the system or individual classes. It is a color stacking bar chart. The horizontal axis of the histogram represents the versions of the software system, while the vertical axis represents the amount of change. The different colors represent the different type of changes. The changes of type except those of type "Delete" have positive values, and those of type "Delete" have negative values.

Both the matrix and the histogram provide a good way to visualize the high-level evolution information of a software system. The matrix is good at presenting evolution phases and styles, growth spurts, and unusual incidents, while the histogram is effective for displaying the amount of change and class evolution types.

Pie chart. The pie chart, like the one reported in the bottom-left corner of Figure 3, is a primary tool to summarize data. We use it to show the amount of different types of changes and their ratios when system or a particular class evolves from one version to the next. It complements the matrix and histogram with actual number and ratio of changes.

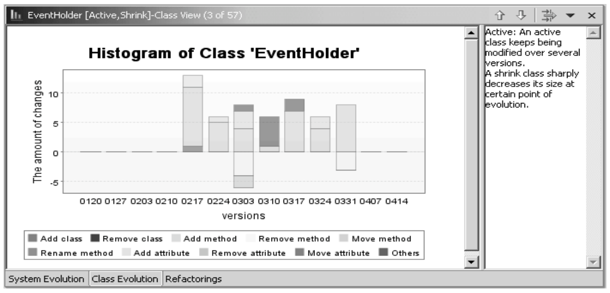
6.2 Usage scenario

Eclipse developers using the evolution-analysis plugin start with a set of XMI models of a software system. They can analyze any two versions, or run the plugin incrementally. The plugin reads in the XMI models, parses their class-hierarchy trees, applies the UMLDiff algorithm against these trees and saves the deltas into change trees, and finally extracts and analyzes the aggregate information, which are visualized in the various views discussed above in an Eclipse perspective shown in Figure 3.

The change trees are shown in a tree view in the top-left corner, *Change Tree view*. This view works similarly to the navigator pane of many IDEs. The user can expand or collapse tree to see more information. To look into the source codes of a specific element, one can double click on the element to bring out the java-source editor, shown in the top-left corner. To inspect previous or next change trees, one has to click the arrow button to move backward or forward.

The bottom-left corner, *Change Summary view*, shows a pie chart that summarizes the amount of different types of changes and their ratios from one version to the next.

The bottom-right corner stacks three views, *System Evolution view, Class Evolution view, and Refactoring view*. The System Evolution view in Figure 3 shows the evolution matrix of the system. The users can toggle between the matrix and the histogram view. The detailed information about what happened in a particular version can be obtained from the Change Tree and the Change Summary views. An editable comment view can be toggled to let the users input any information they may want to note about the system evolution.

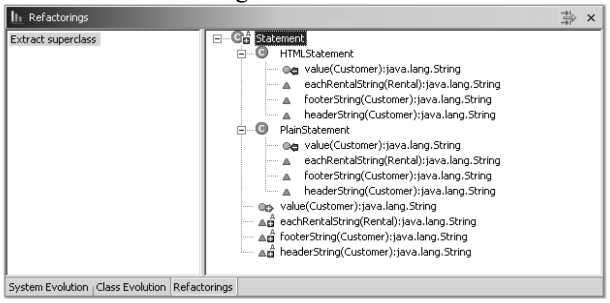


Active: An active class keeps being modified over several versions.
A shrink class sharply decreases its size at certain point of evolution.

Figure 4: Class-evolution view

The *Class Evolution view* of Figure 4 shows the histogram of a class. It also lets the users toggle between the matrix and histogram views. The plugin analyzes the change profiles of individual classes, classifies them into one or more of the evolution types that are shown in the view title, and adds the default comments on evolution types as shown in the right-hand side editable comment window. A query mechanism is implemented to enable the users to select and show the classes of evolution types they are interested in. The plugin also identifies co-evolving classes of individual classes. The users can select one of them from the drop down menu of the class evolution view to show the evolution information of that class, if any.

The *Refactoring view* shown in **Error! Reference source not found.** shows at the left-hand side a list of all the identified refactorings that have been made in a particular version. The right-hand side is a tree view that displays the snippet of the change trees corresponding to the selected refactoring.



Extract superclass

```
Statement
  HTMLStatement
    value(Customer):java.lang.String
    eachRentalString(Rental):java.lang.String
    footerString(Customer):java.lang.String
    headerString(Customer):java.lang.String
  PlainStatement
    value(Customer):java.lang.String
    eachRentalString(Rental):java.lang.String
    footerString(Customer):java.lang.String
    headerString(Customer):java.lang.String
  value(Customer):java.lang.String
  eachRentalString(Rental):java.lang.String
  footerString(Customer):java.lang.String
  headerString(Customer):java.lang.String
```

Figure 5: Class-evolution view

7 The case studies

The objective of our evolution-analysis work is to support developers and inspectors to understand software evolution at the design level by identifying, analyzing, and visualizing class hierarchy changes. In this section, we discuss two case studies that we conducted to evaluate the effectiveness of our method.

It is important to note here that the analysis of the case-study data was performed by the first author alone, who was not involved in the development of these software systems. All his intuitions are in synch with the second author's-who happens to be the supervisor of these software projects-post-mortem understanding of the development progress.

7.1 Longitudinal architecture-evolution analysis

Mathaino [15] is a research prototype tool that can be used to migrate text-based legacy interfaces to modern web-based platforms. It underwent 90 builds from July 2000 till February 2001. The first version has 64 classes, 284 methods, and 256 attributes. The last version has 143 classes, 1770 methods, and 1886 attributes.

The versions between 0-8, 20-41, and 43-80 were growth phases, since the change activities in these intervals were mainly the increase of class, interface, method, and field, in which versions like 14, 23, 35, 40, 64, and 77 were growth spurts. The versions 7-9, 18-19, and 40-42 were identified as maintenance phases, since they contain much method and field movement. Several refactorings, like Extract superclass, Extract class, Move method, etc., were identified in these phases using analysis method described in section 5, which was also validated by the developer's report in [22]. The versions between 81-90 are a steady-going phase. Very few change activities occurred in this phase. The software system was stable. There was an unusual increase in size of system in version 13 and a sharp decrease in version 18. The inspection of change trees of version 13-12 and 18-17 revealed that there are many user interface related classes added in version 13 but most of them are deleted in version 18. All the evolution types of classes listed in section 5 were identified.

7.2 Collaborative software development of small undergraduate teams

In this subsection, we discuss a case study of the term projects of five undergraduate teams that took place during a single-term (about four months) software engineering course. This course is organized in a three-phase lifecycle, and the deliverables of three phases are paper design, user-interface prototype implementation and complete implementation. Currently, these three deliverables are the points where the instructor can identify design problems. Since the student-provided documentation is not consistent throughout the various documents, comparative analysis is difficult and problems may go undetected. We chose to evaluate our design-evolution method on this data, in order to explore its potential impact in project-based software-engineering education.

The objective of the particular term project was to develop a daily event Calendar that can be used to plan future appointments and to place reminders such as holidays and events. Five student teams authorized us to use their software products for this case study. We took weekly snapshots of their projects from their CVS repositories, from January 20th, 2003 through April 14th, 2003, resulting in 13 versions for each project.

Based on evolution-phase analysis, we discovered that teams (A) and (E) defined a few classes in the first place, and proceeded to develop them one step at a time. Their change activities involved *continuous small modifications*. Another characteristic of these two teams is that major changes were made in the middle of their project development, mainly between weeks 7 and 10. They did not try to implement their project at the last minute, like teams (B) and (C).

The evolution processes of teams (B) and (C) contain two versions with aggressive growth spurts. Their projects started with a few classes, and did not change a lot until week 7. However, there is a sharp increase in the size of their projects at week 8, which is followed by small changes until week 10, in which another growth spurt is observed. These *occasional large modifications* coincide with the deadlines for project part 2 and part 3. This means that most features and/or functionalities of their projects were implemented just before the deadline - a bad but not untypical practice. Teams (B) and (C) exhibited similar evolutionary development styles. But, since team (C) adopted the MVC model as the application architecture, their work is more organized than that of team (B), and their project quality as evaluated by the course TA[2] was better. This result validates our intuition that good architecture enables software quality.

[2] For each deliverable, all team projects were marked by the same TA.

Team (D) exhibited a very interesting evolution style. The most changes were made within the first two consecutive versions when they started the development of their project, in weeks 4 and 5. They may have a very good requirement analysis and high-level system design in the first place. Therefore, they seem to know what architecture should be adopted, what functionalities should be supported, and further how to implement them. In that way, they were able to put almost everything in place when they started implementation. Actually, they obtained the best mark for the first deliverable which is essentially a requirements-and-design document, Their change activities at the class level are well-planed and that is just the opposite to those of most other teams. Most other teams added many new classes when the project deadline was approaching in week 10. Team (D) just added a few things, but the most remarkable thing for team (D) at week 10 is that they moved some methods among classes, which means they were trying to improve the system structure, when most other teams were struggling to meet their requirements.

In these five projects, we were able to find in stances of all the class-evolution types but *die-hard* and *legacy*. We believe that the reason is the nature of the undergraduate term projects. They are relatively small and must be completed within about 3 months. The structure of system is simple, and thus it does not need such maintenance activities that bring about *die-hard* and *legacy* classes. On the other hand, due to time constraints students aim at completing a working system and are usually unwilling to perform such maintenance activities. However, we found evidence of refactoring. For example, at the snapshot taken on week 11, team (E) created a utility class named "DateWorker" and date-related functionality was moved from the pre-existing "Appointment" class to the new "DateWorker" class. This is an example of the "class extraction" refactoring.

8 Conclusions

In this paper, we discussed our recent work on understanding the evolution history of object-oriented applications by analyzing the changes of their class-inheritance hierarchies.

At the crux of this work is the change-tree data structure that summarizes the structural differences between two versions of the application's class hierarchy. UMLDiff, the change-tree construction algorithm that produces the change tree, unlike earlier string- and XML-differencing algorithms, is able to recognize element moves in addition to element additions, deletions, and renamings; thus, modifications that were reported as unrelated deletions and additions become noticeable. Furthermore, it reports the detected changes in terms of UML semantics, thus better matching the developers' intuition about their system. In addition to UMLDiff, we have also developed a suite of analysis tools and visualization instruments used to make design-evolution information intuitive to software developers and inspectors. These tools capture evolution information at several levels of granularity: the system level, the individual-class level and the change-pattern level. To date, we have evaluated this work in the context of two different case studies, both of which revealed interesting information about the design of the subject software systems to an analyst with no prior knowledge of their development.

In the future, we plan to extend the process with an interactive step for heuristically discovering further types of changes, such as field renamings for example. We plan to investigate the comparison of original designs against designs reverse-engineered from code to identify discrepancies between the designers' intent and the actual implementation. We also plan to evaluate the impact of our methodology when available to developers in the process of development. We expect that this analysis should enable them to better monitor and control the progress of their work.

References

1. R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile, September 1994.
2. E. J. Barry, C.F. Kemerer, and S.A. Slaughter, "On the Uniformity of Software Evolution Patterns", *Proceedings of the 25th International Conference on Software Engineering,* Portland, Oregon, May 2003, pp. 106-113.
3. D. Barnard, G. Clarke and N. Duncan, "Tree-to-tree Correction for Document Trees", Technical Report 95-375, Queen's University, January 1995.
4. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information", *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Montréal, Québec, June 1996, pp. 493-504.
5. C. Collberg, S. Kobourov, J. Nagra, J. Pitts and K. Wampler, "A system for graph-based visualization of the evolution of software", *Proceedings of the 2003 ACM symposium on Software visualization,* San Diego, California.
6. S. Demeyer, S. Ducasse, and M. Lanza, "A hybrid reverse engineering platform combining metrics and program visualization", *Proceedings of 6th Working Conference on Reverse Engineering,* IEEE, Oct. 1999.
7. S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics", *ACM SIGPLAN notices*, 2000, 35(10):166-177.
8. A. Egyed, "Scalable Consistency Checking between Diagrams - The VIEWINTEGRA Approach," *Proceedings of the 16th IEEE International Conference on Automated Software Engineering,* San Diego, USA, 2001.
9. S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data", *IEEE Transactions on Software Engineering*, 2001, 27(1):1–12.
10. S. G. Eick, J.L. Steffen, and E.E. Sumner, "SeeSoft—A tool for visualizing line-oriented software statistics", *IEEE Trans. Software Engineering,* 1992, 18(11):957–968.
11. S. G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes", *Software Engineering,* 2002, 28(4):396-412.
12. E. V. Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells", *Proceedings of 9th Working Conference on Reverse Engineering"*, Oct, 2002.
13. M. Fowler, "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley, 1999.
14. H. Gall, K. Hajek and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History", *Proceedings of the International Conference on Software Maintenance*, Bethesda, Washington D.C., November 1998.
15. R. Kapoor and E. Stroulia, "Mathaino: simultaneous legacy interface migration to multiple platforms", *Proceedings of 9th International Conference on Human Computer Interaction,* 2001.
16. R. Kollmann, P. Selonen, E. Stroulia, T. Systa, A. Zundorf, "A study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering", *Proceedings 9th Working Conference on Reverse Engineering, IEEE.*
17. M. Lanza, "The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques", *Proceedings of International Workshop on Principles of Software Evolution,* 2001.
18. M. M. Lehman and L. A. Belady, "Program Evolution-Processes of Software Change", Academic Press, London, 1985, 538pps.
19. Y. Liu and E. Stroulia, "A Lightweight Project-Management Environment for Small Novice Teams", *Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering*, 2003, pp. 42-48.
20. E. Myers, "An O(ND) difference algorithm and its variations", *Algorithmica,* 1986, 1(2):251-266.
21. P. Selonen, K. Koskimies, M. Sakkinen, "Transformations between UML Diagrams", *Journal of Database Management*, Vol. 14, No. 3, 2003.
22. E. Stroulia and R. Kapoor, "Metrics of Refactoring-based Development: An Experience Report", *Proceedings of the 7th International Conference on Object-Oriented Information Systems*, Calgary, AB, Canada, 27-29 August 2001, pp. 113-122, Springer Verlag.
23. R. A. Wagner and M.J. Fischer, "The string-to-string correction problem", *Journal of the ACM,* January 1974, 21(1):168-173.
24. K. Wong, W. Blanchet, Y. Liu, C. Schofield, E. Stroulia, and Z. Xing, "JRefleX: Towards Supporting Small Student Software Teams", IBM Eclipse Workshop at OOPSLA 2003 (to appear).
25. T. Zimmermann, S. Diehl, and A. Zeller. "How History Justifies System Architecture (or not)". *Proceedings of International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003.
26. http://www.cs.unc.edu/~stotts/COMP204/refactor.
27. Eclipse, http://www.eclipse.org.
28. Mosell EDM Ltd, http://www.deltaxml.com.
29. Rational Rose, http://www.rational.com.
30. Together, http://www.togethersoft.com.
31. Weka, http://www.cs.waikato.ac.nz/~ml/weka