Understanding Class Evolution in Object-Oriented Software

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

Abstract

In the context of object-oriented design, software systems model real-world entities abstractly represented in the system classes. As the system evolves through its lifecycle, its class design also evolves. Thus, understanding class evolution is essential in understanding the current design of the system and the rationale behind its evolution. In this paper, we describe a taxonomy of class-evolution profiles, a method for automatically categorizing a system’s classes in one (or more) of eight types in the taxonomy, and a data-mining method for eliciting co-evolution relations among them. These methods rely on our UMLDiff algorithm that, given a sequence of UML class models of a system, surfaces the design-level changes over its lifecycle. The recovered knowledge about class evolution facilitates the overall understanding of the system class-design evolution and the identification of the specific classes that should be investigated in more detail towards improving the system-design qualities. We report on two case studies evaluating our approach.

1 Motivation and Background

The objective of reverse engineering is most often to enable software understanding in support of maintenance, feature enhancement and adaptation activities [5]. In object-oriented systems, classes model abstractions of real-world entities around which these systems are designed. Therefore, understanding the system classes, i.e., their internal structure and their role in the context of the system functionality and behavior, constitutes a crucial step towards understanding the overall system design for both maintenance and new development.

There have been several research efforts to date aiming at understanding systems at the class level. For example, Lanza et al. [17] introduced the “class blueprint”, a visualization of the internal structure of system classes at a particular point in their lifecycle. The class blueprint distinguishes among different types of classes, such as classes with wide interfaces that offer many entry points to their functionalities, definers that reside at the top of a hierarchy or specializers that are leaves of an hierarchy, etc. However, such visualizations require a substantial interpretation effort on behalf of their users and become fairly “unreadable” for large systems with numerous classes.

Furthermore, given that most software development nowadays adopts an evolutionary lifecycle model, analyzing a single snapshot of a system’s classes enables only limited insight; a comparative analysis of a sequence of snapshots should be more valuable in understanding the system’s design rationale. For example, consider a software maintainer who wants to identify “hotspots”, i.e., areas of substantial evolutionary activity, over the lifespan of a 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 evidence of highly coupled design may focus his examination into the source code of these classes to determine the cause of problem and to propose modifications to remedy it.

Such evolutionary analysis was the objective of Demeyer et al. [7], which investigated the use of comparative analysis of software metrics for drawing inferences regarding the evolution of a system. However, the result of their analysis refers to the system as a whole and does not provide any insight regarding the evolution of individual or groups of classes.

Another, potentially more precise, source of evolutionary information could be documentation, either at the source-code level or at the change-log level of the version-management system used for the development of the software system. Unfortunately, more frequently than not, such documentation is sparse and inconsistent [5,13].

In our work on understanding class evolution in object-oriented systems, we have adopted class models of subsequent system snapshots (which may be released versions or simply snapshots checked-out in regular time intervals) as the primary input of our method. These class models are easily obtainable, given the source code that resides in a version-management system and any of a variety of existing round-trip software-development tools [29,30], and they are, by their very nature, fairly accurate representations of the source [19]. The fundamental intuition underlying our method is that by
comparing a sequence of snapshots of a system’s class
cmodels, one can extract a history of the evolution of the
system, in terms of the “additions”, “removals”, “moves”,
“renamings” and “signature changes” of classes, interfaces,
and their fields and methods, between subsequent snapshots as they are reverse engineered
from its source code.

In addition to its lightweight and simpler knowledge
assumptions, the other important methodological
constraint of our method is that it has to be highly
automated and not to require much input information or
output interpretation from the user.

Given the structural changes reported by the class-
model comparison step, our method produces two types
of automatic analyses. First, it categorizes each
individual system class in terms of a class-evolution
taxonomy that consists of eight distinct evolution types.
Second, it employs a data-mining method to elicit co-
evolution relations between sets of classes that have
common change behaviors. These two types of class
evolution analyses enable software maintainers to obtain
plausible answers to the following questions:
- Which classes were changed and which methods
  and/or fields were added, deleted, moved, renamed,
signature-changed and in which version?
- Were there unusual incidents over the lifecycle of
  the classes?
- Were there classes that have the same lifespan as the
  whole system and never changed at all?
- Which classes either change frequently or do not
  change over several versions?
- Which classes exist for only a very short time?
- Were there classes that grow or shrink rapidly from
  one version to the next?
- What other classes should be examined when
  modifying a particular class?

The remainder of the paper is structured as follows. In
section 2, we relate this work to previous research. In
section 3, we present the overall methodology and
rationale of our approach. Two case studies evaluating
our approach are discussed in section 4. Finally, we
conclude our discussion with a summary of the lessons
we have learned to date and our plans for future work.

2 Related work

There already exists a substantial body of literature on
the general “software-evolution understanding” topic. In
general this work can be grouped according to the
primary source of input data.

A substantially analyzed source of data has been the
“history” recorded by version-management systems. 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.

Gall et al. [12] use information in the product release
history of a system to uncover coupling among
modules based on sequence matching. Zimmermann et al.
[25] identify (heavily dependent on visualization of
historical data stored in CVS archive) the fine-grained
coupling between program entities like methods and
fields.

The recent work of Shirabad [21] is the most similar
to our second type of automatic analysis. Based on past
maintenance experience, recorded in the form of change
requests and code-update records, he explored
supervised inductive-learning method for recognizing
co-updated modules to predict whether updating one
source file may require a change in another file. In
contrast, our data mining method for recovering class
cou-evolution is unsupervised and does not require labeled
training examples.

The major shortcoming of this line of research is that
the interpretation of the version-management activities
within the lifetime of a software project highly depends
on the process model adopted by the developer team.
Therefore, when this model is not known, as is
frequently the case, inferences based on this data are
quite unreliable.

A second line of research has focused on the
visualization of software-process statistics, source code
metrics, static dependence graphs, etc. Eick et al.
developed tools [10] for visualizing the evolution of
software statistics at the source-code line level and
change data such as developer, size, effort, etc. Collberg
et al. [6] focus on using a temporal graph model to
visualize entire evolution history of a given system, such
as control-flow graphs, inheritance graphs, etc., which
are collected for a fixed time slice.

Lanza [16] describes how to use a simple two-
dimensional graph to convey the implicit information of
software metrics. Based on the visualization of the
evolution of class metrics (number of methods and
number of instance variables), they also define a
categorization of classes, similar but more coarse-
grained than our taxonomy, thus failing to recognize
products of restructuring activities.

These visualization methods require a substantial
interpretation effort on behalf of their users and tend not
to scale gracefully to handle large systems. They are
limited by the size of the visible area of a screen. In
contrast, the method we propose is quite automated with
only a little human effort needed. Furthermore, it is
relatively more scalable because it focuses on only
changing system classes instead of all its modules.
Finally, there has been some work at analyzing the changes of software at the design level. Egyed [8] has investigated a suite of rule-based, constraint-based and transformational comparative methods for checking the consistency of the evolving UML diagrams of a software system. Selonen et al. [20] have also developed a method for UML transformations, including differencing. However, they cannot surface the specific types of changes as reported by UMLDiff and these projects have not explored the product of their analyses in service of evolution understanding.

3 Methodology

In this section, we present our structural modification detection algorithm, UMLDiff. We then discuss class evolution profiles and consider two types of automatic analyses: (a) the categorization of system classes into one (or more) of eight distinct evolution types and the detection of co-evolving classes given a set of such profiles, and (b) how the derived class evolution knowledge may assist software engineers in their task of understanding software evolution and planning future maintenance activities.

3.1 UMLDiff: Class-modification detection

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. The general tree-to-tree correction problem has been studied extensively [2], and has been applied to show differences between XML data [28]. However, such general tree-differencing algorithms report changes as “XML element modifications” ignoring the domain-specific semantics of the nodes. Let us consider XMI, XML Metadata Interchange for UML models, as an example. When a class implements a new interface, a general XML-differencing tool would only report that a set of xml-nodes was inserted but would not recognize the implementation of a new interface, since it does not understand the XMI semantics. For the same reason, if an attribute or method was moved from one class to another, the change would most likely be reported as two separate activities: the addition of some xml nodes and the deletion of some others.

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 model, 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”.

In the context of software evolution, where local transformations, such as refactoring, frequently involve moving features 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 are able to recover the field/method movement when analyzing software evolution, in order to recognize “preventive maintenance” phases in the software life span. Figure 1 discussed below, shows an example of such “move” operations.

In general, the problem of detecting the class-model changes between two versions of an object-oriented system can be viewed as a graph-difference problem, since class models can be viewed as specific types of directed graphs. This problem is NP-complete which makes an automatic approach impractical. Therefore, we have limited our initial exploration to considering only the inheritance hierarchies of the class model. The algorithm underlying our class-evolution analysis work, UMLDiff, was designed to address the shortcomings of the domain-agnostic XML differencing algorithms that we discussed above.

UMLDiff is essentially a domain-specific tree-differencing algorithm, aware of the UML semantics captured by the XMI syntax. It takes as input two UML class models, corresponding to two versions of the system under analysis, represented in XMI. Such class models can be either produced in the software-design phase by the system developers, or they can be reverse engineered from the system code, using any of the currently available roundtrip software engineering tools [29,30]. The first step of the algorithm is to parse the input forests of class models into two labeled tree structures. The target representation contains the application classes and interfaces, their fields, their methods and their inheritance, implementation, and nested class relations. Nested classes of a particular class are enclosed in a special element in the context of that class. Multiple-inheritance is handled by duplicating the class node (not including its children) under each of its super classes. The next step of the algorithm is to identify the after-before changes between them, in terms of the “additions”, “removals”, “moves”, “renamings”
and “signature changes” of super- and sub- classes, interfaces, and their fields and methods. This UML differencing process brings to the surface structural modifications to the software design. The results are represented as change trees, i.e., trees of delta operations, i.e., structural modifications, which, if applied to the before version would result in the after version. Change trees are represented in an XML-based syntax. Figure 1 diagrammatically depicts an example change tree.

![Change Tree Diagram](image)

Figure 1 An example of change tree

The different icons represent the different object-oriented entities: “class”, “interface”, “method”, and “field”. The top-right adornments show the modifiers of the object, for example, “abstract”, “static”, etc. The bottom-right adornments represent the status of a particular object. It can be plus sign for “add”, minus sign for “delete”, filled triangle for “rename”, empty triangle for “change signature”, arrow with minus sign for “move out from source”, arrow with plug sign for “move into target”. The change tree presents the developers the detailed structural modifications to the class model when software system evolves from one version to another.

The Figure 1 shows 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 corresponds to the differences between versions 27 and 28 of the extended refactoring sample from M. Fowler’s book [11] as found in [26]. It represents the modifications to the class model 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 [11]”

### 3.2 Class-evolution profile analysis

UMLDiff reports the changes between two snapshots of a system’s class hierarchies. There are N such hierarchies in an evolving software system with N successive snapshots, and consequently UMLDiff can be applied N-1 times to generate the differences between the \((I+1)^{th}\) and \(I^{th}\) snapshots, where \(1 \leq I < N\). Thus, \(N-1\) change trees can be obtained that record the structural modifications, in terms of the “additions”, “removals”, “moves”, “renamings” and “signature changes” of classes, interfaces, and their fields and methods, when software evolves from one snapshot to another.

A sequence of change trees between successive snapshots of a system provides an audit trail of the design changes that the system’s class models have suffered throughout the evolutionary lifecycle of the software system. This trail can be analyzed to produce a set of class-evolution profiles, corresponding to the individual system classes. The class-evolution profile is represented in an XML-based syntax and reports the changes made to an individual class in each subsequent system snapshot, i.e., in which snapshot it was created, with how many fields and methods, which of its elements were added, deleted, moved, renamed, or signature-changed in a particular snapshot, and in which snapshot it was deleted. The transformation of a sequence of change trees into a set of class-evolution profiles is implemented as an XSLT program.

The evolution profiles of the system classes are automatically analyzed in two ways: First, they are categorized into one (or more) of eight types in the taxonomy of class evolution profiles; Second, the co-evolution relations between sets of classes are detected using a data-mining method.

### 3.3 A taxonomy of class-evolution types

The class-evolution profile records the change trail of each individual system class. By examining a large number of systems and their classes, we observed that there are a few interesting patterns that can be recognized in the class-evolution profiles. Therefore, we have developed a taxonomy of class-evolution types. Based on its evolution profile, a class can thus be automatically categorized into one (or more) of the eight distinct evolution types in this taxonomy.

The evolution history of each class is broken into user-specified equal length periods of snapshots from its starting point. To improve the continuity of class evolution type analysis, we employ a moving window technique when we create the snapshot periods. For classes of an evolving software system with \(N\) successive snapshots, let the user-specified length of snapshot period be \(n\), thus, instead of generating \(\lfloor N/n \rfloor + 1\) discrete snapshot periods, we generate \(N-n+1\) overlapped snapshot periods of length \(n\) for each individual class, which makes the analysis smoother and more accurate. For a class snapshot period with \(n\) snapshots, we check its evolution characteristic and categorize it into one (or more) of eight evolution types as discussed in the rest of this section. The categorization of class evolution types is implemented as an XSLT program.
Note that the class-evolution types currently in the taxonomy are only the most distinct ones in our case studies. More categories may be possible to identify with further analysis. Furthermore, these categories introduced here are not mutually exclusive. The same class may exhibit several different type patterns in its evolution profile.

In the remainder of this section, each type in the taxonomy is discussed. Some of the types simply indicate different development activities during a particular period of the system evolution, while others constitute evidence for structural-design shortcomings that need to be improved in the future. Classification of the system classes in the taxonomy enables a quick and intuitive view of the whole system and the potential rationale behind its evolution.

3.3.1 Active classes

An active class keeps being modified over several snapshots. For a particular snapshot period of class \( C \), if the number of snapshots in which class \( C \) has been modified is greater than the user-specified threshold, then class \( C \) is considered as active during this snapshot period. Active classes are hotspots from the view of software evolution.

Active classes are the result of several frequent software-development phenomena. For example, “entity” classes representing data structures are often found to be incorrect or insufficient abstractions of the objects they represent. Thus, they suffer a sequence of modifications until they are deemed “correct”.

Alternatively, behavioral interfaces may be ill-designed. The concept of interfaces is critical in designing and developing extendible object-oriented systems, and restructuring of classes’ interfaces is the result of several development activities. Design patterns are essentially conceptualizations of a set of collaborating interfaces designed to address a set of contradicting problem forces. Late adoption of a design pattern will likely result in class-interface changes. Similarly, many refactorings [11], such as move features between classes, tease apart inheritance, etc., aim at improving the modularity and the understandability of a class’ public interfaces.

Finally, important classes, implementing complex functionalities, become sometimes incohesive and need to be restructured. When a class is responsible for a complex functionality, sometimes it ends up with a multitude of members, many of them used in slightly different contexts. As a result, requirements changes in any of these contexts impact the same class, causing continuous changes to it.

In general, active classes are frequently evidence of poor design and should be examined closely.

3.3.2 Idle classes

An idle class rarely changes after it is introduced into the system. For a particular snapshot period of class \( C \), if the number of snapshots in which this class has been modified is less than the user-specific threshold, then class \( C \) is considered as idle during this snapshot period.

There are several reasons for the existence of idle classes. They may be core classes of an adopted framework or external components. Alternatively, they may be root classes of the system’s class hierarchies without many implemented features, or classes implementing stand-alone functionalities, or well-designed classes that need little changes.

On the other hand, an idle class could also be dead code. For example, an originally useful class may end up with few responsibilities as a result of refactoring, or a class stub, intended for a particular purpose, may be abandoned due to requirements’ or design changes.

Particular attention should be given to idle classes whose lifespan is similar to that of the software system. They may have already been disused but no one “dares” to remove them. According to Fowler, [11] “Each class you create costs money to maintain and understand. A class that isn’t doing enough to pay for itself should be eliminated. So let them die with dignity.”

3.3.3 Rocket and shrinking classes

A rocket class exhibits a sharp increase in its size at a certain point of its evolution. For a particular snapshot period of class \( C \), if, at some snapshot(s), the total number of newly added fields and methods is greater than the user-specific threshold, then class \( C \) is considered as rocket during this snapshot period. A shrinking class is exactly the opposite; its size sharply decreases at some point of its evolution.

Rocket classes may be the result of well-planed feature extensions. However, more often than not, developers tend to write code before they have decided where is the best place to put it, relying on subsequent refactorings to distribute the newly added fields and methods. Even worse, when different developers work on the same code base developing related features over time, they may replicate similar entities. Soon, the strategies for solving a certain kind of problem are spread through the code, which makes subsequent development “like swimming in spaghetti without a paddle” [11].

Rocket classes should be closely examined since they are often prime refactoring candidates. Similarly, shrinking classes, usually resulting from refactoring activities redistributing their functionalities, are candidates for Collapse Hierarchy and Inline Class refactorings.

3.3.4 Die-hard and legacy classes

A die-hard class is a class that is removed from the system but most of its functionalities are moved to other
classes. At the ending point of a particular class $C$, if the number of “move out from source” fields and/or methods is greater than user-specific threshold, then class $C$ is thought of as an instance of die-hard class. A legacy class is just the opposite of die-hard class: it is introduced into the system as a placeholder for fields and methods moved in from other classes. Both these types of classes represent evidence of redistribution of functionality or the reorganization of class hierarchy.

According to Lehman [18] “As a program is evolved its complexity increases and it will be perceived as of declining quality unless rigorously maintained.” For a successful and long-lived software system, we expect periods in its lifetime when proactive structural maintenance is done to reduce its complexity. In similar spirit, XP development methods [11] advocate the intermingling of new development and refactoring phases. Such proactive-maintenance and refactoring activities often result in feature movements resulting in die-hard and legacy classes.

Upon examination, these classes may provide hints regarding the system’s design rationale. For example, an idle subclass that isn’t substantial enough to “pay for itself” is subjected to a Collapse Hierarchy and thus disappears as a die-hard class. An Extract Class refactoring, applied to fan out fields and methods from an active large class, may result in a legacy class. Finally, software developers may recognize that they have just replicated some existing functionality and may create a legacy (super)class to move into it the common features of the replicas.

### 3.3.5 Volatile classes

Given a particular snapshot period, if the number of fields and/or methods of class $C$ that have changed their visibility, modifier, data type, and parameter list is greater than a user-specified threshold, then class $C$ is considered as volatile during this snapshot period.

The volatile classes represent evidence of data structure or class interface restructuring. They may be the result of refactorings such as Encapsulate Field, Rename method, Hide Method, Introduce Parameter Object, etc. Signature changes may aim at simplifying long parameter lists, changing the visibility of fields and methods, downcasting, or extracting data clumps into new objects. By carefully examining signature changes and hypothesizing the motivating rationale behind them, a developer can gain some intuition into the design style and qualities adopted for the system under development.

### 3.3.6 Short-lived classes

A short-lived class exists only in a few versions of the system and then disappears. A class can be defined as short-lived, if the interval between the class creation and deletion is less than a user-defined threshold.

Such classes may have been used to prototype a feature or to test another class. A single short-lived class may not be especially interesting. But typically a larger set of short-lived classes that show up over a limited time span may indicate that the system is going through rapid feature extension and its developers are trying different solutions for this purpose. A software maintainer should closely monitor the development process and the product produced during such periods, since they may introduce side effects into the system, for example, a class added because of changes that were planned but never implemented.

### 3.4 Class co-evolution

In section 3.3, we discuss a taxonomy of class evolution types based on the analysis of evolution characteristics of individual classes. In this section, we present a method for recovering the implicit evolutionary interdependence between sets of classes. We also discuss potential applications of recovered class co-evolution knowledge in the context of software maintenance.

#### Table 1: Transaction database for class evolution

<table>
<thead>
<tr>
<th>SID</th>
<th>Sets of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S01</td>
<td>C1, C2, C5</td>
</tr>
<tr>
<td>S02</td>
<td>C2, C4</td>
</tr>
<tr>
<td>S03</td>
<td>C2, C3</td>
</tr>
<tr>
<td>S04</td>
<td>C1, C2, C4</td>
</tr>
<tr>
<td>S05</td>
<td>C1, C3</td>
</tr>
<tr>
<td>S06</td>
<td>C2, C3</td>
</tr>
<tr>
<td>S07</td>
<td>C1, C3</td>
</tr>
<tr>
<td>S08</td>
<td>C1, C2, C3, C5</td>
</tr>
<tr>
<td>S09</td>
<td>C1, C2, C3</td>
</tr>
</tbody>
</table>

We think of a change tree, plus the first snapshot, as a transaction that records the system classes that have been modified (including creation and deletion) in the corresponding snapshot. For a software system with $N$ snapshots a database with $N$ transactions is generated that describes the class level evolution of software system. Each transaction $T$ contains a set of classes with a unique identifier, the snapshot ID (SID). Table 1 shows a hypothetic transactional database. We assume that the system, in its final version, has five classes C1, C2, C3, C4, C5. Its first snapshot contains only C1, C2, C5 and in the next snapshot, class C2 is modified and class C4 is added, and so on.

We apply the Apriori association-rule mining algorithm (we used its implementation in the Weka [31] toolkit) to discover co-evolving classes, in most cases previously undocumented or unknown, that have common change behaviors. We describe briefly the Apriori algorithm here. Interested readers are referred to [1] for more details.

Given a set of transactions, the original Apriori algorithm generates all association rules with at least some user-specified minimum support and confidence. The algorithm involves two subproblems. First, all sets of items (itemsets) that have transaction support above
minimum support. The support for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called large itemsets and all others are small itemsets. Next, the large itemsets are used to generate the desired rules. The general idea is that, if ABCD and AB are large itemsets, then the rule AB → CD holds if its confidence, i.e., the ratio support(ABCD)/support(AB) is greater than the user-specified minimum confidence. Note that the rule will surely have minimum support because ABCD is large.

However, the so-called strong rules generated by the support-confidence framework may not be interesting to the user, since the premise and the consequent may be negatively associated, which means that the occurrences of the former may decrease the likelihood of the occurrence of the latter. The alternative measure lift can be used to measure the statistical dependence (correlation) between the occurrences of itemsets. The Weka toolkit also supports the support-lift framework, which we are using to generate correlation rules between sets of classes.

For the transactional data shown in Table 1, if the minimum support is set to 20% and the minimum confidence to 40%, then we can generate the rules C1 ∧ C2 ⇒ C5 or C5 ⇒ C1 ∧ C2, which indicate a co-evolution relation among these three classes. Note that the rule allows a consequent to have more than one item, which is the advantage by applying an association rule method over the classification method proposed in [21].

3.4.1 Intentional co-evolution

Class co-evolution may be the consequence of the original design and implementation decisions. For example, an approved design may require certain classes to be modified for every new feature addition. Unfortunately, such design intent is not always documented; the software developers just “know” that they have to modify a certain set of classes when making a certain kind of change. However, such knowledge is easily lost with staff turnover.

In this sense, detecting co-evolving classes can be thought of as a design recovery process that provides a way to identify high-level relations among classes, which increases the overall understanding of a long-lived evolving system. This co-evolution relation can be used as the basis for advice on maintenance activities. For example, if three classes were often changed together, when a developer modifies two of them, it would be recommended to examine if a change is also necessary to the third one.

3.4.2 “Maintenance smells”

Fowler [11] describes a set of “code smells” that constitute evidence for the need for particular refactorings. Some of them can be characterized as “evolution smells”, which are not obvious in a single snapshot of system but can be identified by analyzing changes made to a system over time. Two examples of such evolution smells are the following:

- Shotgun Surgery: whenever a kind of change is made necessitating many little changes to many different classes.
- Parallel Inheritance Hierarchies: whenever subclassing one class results in having to subclass other classes (a special case of shotgun surgery).

The most observable evolution characteristic of these smells is the classes that have been changed together over time. Therefore, the identification of co-evolving classes may help software engineers discover whether the system suffers from these smells. For example, the original design of a software system may have followed the Model/View/Controller (MVC) model. However, due to side effects of changes that the system has gone through its life span, a cluster of classes, belonging in the presentation layer and the data-model layer, are discovered to evolve in parallel. That may reveal the high coupling between presentation and data model layer, which means that the current system implementation deviates from the original design intent. A “Separate presentation from data model” refactoring could be applied to improve the cohesion and reduce the coupling.

3.4.3 System instability

In addition to enabling maintenance advice and providing evidence of “smells” necessitating particular refactorings, class co-evolution may also used as an indicator of general “system instability”.

Bianchi et al. [4] and Hassan et al. [14] claim that the entropy of a software system is a good indicator of the degree of disorder of its structure. The term “entropy” refers to the amount of uncertainty related to information in a distribution. Intuitively, in the context of software evolution, if a software system is being modified across all its modules, it will have highest entropy, and the software maintainers will have a hard time keeping track of all the changes. Both researches rely on maintenance documentation to determine the relations among system components.

In a similar vein, Bevan [3] defines software instability as a set of related artifact elements that have often changed together. She uses a static dependence graph to visually identify related software artifacts. We believe that the co-evolving classes detected by applying association rule mining could provide a good primary input for system instability analysis. We plan to evaluate the overall development process by analyzing the knowledge revealed by co-evolving classes. We expect to be able to identify abnormal phases of software evolution due to class co-evolution.

4 Case studies

The objective of our evolution-analysis work is to support software practitioners to understand software
evolution at the higher level of abstraction by automated identifying and analyzing the evolution characteristics of system and its components from multiple perspectives at varying degree of resolution.

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 first author, who was not involved in the development of these software systems, performed the analysis of the case-study data. All his intuitions are in-sync with the second author’s— who happens to be the supervisor of these software projects-post-mortem understanding of the development progress.

4.1 Evolution of a long-term XP project

Mathaino [15] is a research prototype tool that can be used to migrate text-based legacy interfaces to modern web-based platforms. It was developed by following a strict refactoring-based development process, inspired by eXtreme Programming methodology. It underwent 91 builds from July 2000 till February 2001. The first version has 29 classes, 284 methods, and 256 attributes. The last version has 144 classes and about 1800 methods and 1800 fields. We reverse-engineered the source code of the 91 system versions to generate 91 class models and run UMLDiff to surface design changes when software evolves from one version to another. We set the length of the snapshot period to 5.

Class MathainoXMLGUIGeneratorPlugin has been continuously modified since it was introduced into system in the 20th version. In the 24th and 29th versions, a large number of fields/methods were added to this class. This is a very typical example of development practice for new feature extensions that a set of field and/or method additions followed by rapid change over a fairly limited time span. Similar cases were observed with the classes “MathainoMainFrame”, “ScreenFechter”, etc.

A set of short-lived classes, such as dlgUser, dlgMsg, GUIView, FrameGrpah, were introduced into the system in the 13th version, and were then removed from system in the 18th version. It seems the developer of Mathaino had been testing user-interface related features during this period of time.

Class MathainoCreatorPluginRegisty is a shrink class, which lost most of its fields and methods in the 19th version. This is actually the result of the refactoring activity discussed below.

We identified instances of idle classes, such as ConstantFieldNode, SplitPaneNode, TableData, etc. Class SplitPaneNode has the same life span as the Mathaino system. An interesting idle class is MathainoPlugin, which is the ancestor abstract class of all creator and parser plugins. It was introduced into system in the 19th version as the result of a refactoring activity, and then was modified a little bit in the 20th version. After that, it has never been touched.

By applying Apriori mining, we discovered a set of co-evolving class clusters, including (a) AbstractForm, AbstractInputField and AbstractOutputField, (b) OutputFieldProperties, InputFieldProperties and FormProperties, (c) MathainoInputFieldRepository and MathainoInputFieldRepositoryNode, and (d) FormNavigator, MathainoXHTMLGUITranslator and TaskDataExtractor. Note that the first three class clusters share the same prefix or suffix; the developer of Mathaino followed a principled notation convention, which might enable the discovery of the co-evolution relation through code inspection. At the same time, given the naming convention, the prefix similarity may indicate that the co-evolution is intentional. The fourth cluster, however, does not belong in the same category. It might also be intentional co-evolution. But it cannot be recovered by simply checking class names.

By analyzing the evolution of the first 18 versions, we also discovered that a set of classes, MathainoCreatePlugin, MathainoParserPlugin, and their subclasses MathainoTemplateCreatorPlugin and MathainoTemplateParserPlugin, were co-evolving, but not after the 20th version. The names of these classes suggest that there may exist a parallel inheritance hierarchy in the system. In the 19th version, we identified instances of die-hard, legacy and volatile classes, such as MathainoPluginHandler, MathainoPluginRegistry, MathainoPlanNavigatorPlugin, MathainoPluginContext, MathainoTemplateCreatorPlugin, etc., which suggest that the developer of Mathaino probably made some change to remove this co-evolving in the 19th version. His report [22] validated our intuition based on the analysis of evolution of these classes. Until its 18th version, Mathaino had two separate plugin hierarchies, one for the “creator” and the other for the “parser”, and two separate plugin loaders and registries respectively. This design was not extensible to handle new types of interactions and would easily result in “parallel inheritance”. At the code level, a lot of code was duplicated. It would seem then that the Mathaino developer noticed this evolution smell and made a design decision in the 19th version to “Extract Superclass” from these two separate plugin class hierarchies, and their corresponding plugin loaders and registries, which reduced the code duplication and made the system architecture much more maintainable.

4.2 Collaborative development process of small undergraduate teams

We also conducted a comparative case study on term projects of five undergraduate student teams, which took place during a single-term (about four months) software engineering course, to explore the potential impact of our
class evolution analyses on their collaborative development process. This course is organized in a three-phase lifecycle, and the deliverables of three phases are requirement analysis and object-oriented design, user-interface prototype implementation and complete implementation. 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. Although the “official” project length was 12 weeks, only 5 to 7 different snapshots were actually obtained for each project. Therefore, we set the length of snapshot period to 3.

In these five projects, we were able to find instances of all the class-evolution types except from die-hard and legacy. For example, the class “CalenderModel” of team (E) is an instance of rocket class, which changed dramatically at the snapshot of week 9. It is also a typical example of active class. This class is actually a large class that was continuously changed during the project. Team (E) used the MVC model for architecting their project, but they designed a single model class, “CalenderModel”. Each different view uses only part of the information in this class, and as their visualization of the calendar changed, their model kept on evolving too. A similar problem is observed with the classes “Entry” of team (B) and “ApplicationModel” of team (D).

Team (E) created two classes, “MORCal” and “MORCalModel!” at the snapshot of week 10, which were not touched afterwards. Upon closer inspection of the code, these classes were found to be dead code. In such a case, the instructor could have suggested to the team that they remove these two classes to improve the understandability and maintainability of their project.

At the snapshot of week 8, team (C) created a class “ViewDriver” to test the “YearView” and “MonthView” classes, the class “ViewDriver” was deleted next week. Another such example is the “UserModel” class of team (E). It was created in the snapshot of week 10 with methods “addToDo” and “ToDoModel”, and was removed in the following snapshot. It seems that the class “UserModel” was initially designed to handle some “ToDo” related features. Team (E) may have later found out that it was not a good place to implement such functionality. Thus, they removed it shortly after it was added into the system.

We believe the reason for lacking instances of die-hard and legacy class 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. Furthermore, because of the short time period and limited number of snapshots, most classes of student projects are co-evolving together. Our examination of the co-evolving classes indicated that they were more likely accidental and did not reveal any real class dependencies.

5 Conclusion and future work

In this paper, we discussed our method for analyzing class evolution in support of understanding the design drift in an evolving object-oriented software system. The method relies on readily available data, as opposed to consistently documented software project change requests. It takes as input a sequence of class models of the system represented in XMI, reverse engineered from a corresponding set of code versions. These models are compared using the UMLDiff algorithm to detect various types of changes to the system’s classes, interfaces, and their fields and methods. Finally, based on the class-evolution profiles, we produce two types of automatic analyses: the categorization of individual system classes according to the taxonomy of class evolution types and the detection of co-evolution of sets of classes.

We have discussed potential applications of recovered class evolution knowledge in the context of software maintenance: advice regarding the scope of future maintenance activities, guidance for particular refactorings and, potentially, recognition of system instabilities. In our Mathaino case study, we discovered several class co-evolution instances and we also found evidence that the project developer acted according to the advice that our theory would have generated, had it been in place during the system’s development.

The approach of analyzing class evolution presented in this paper has been implemented as a part of one of two analysis plugins in Eclipse [27], in the context of the JRefleX project [24]. In addition to the UMLDiff algorithm and two types of automatic analyses we have already discussed, the plugin has also implemented several visualization instruments to present analysis results, such as change tree view (see Figure 1) and class evolution histogram.

For the future work, we are investigating whether a more specific notion of co-evolution, in terms of the specific modifications identified by UMLDiff, would enable more precise maintenance and refactoring advice. The more specific co-evolution notion would enable us to answer such question as “Are there any classes in some part of the hierarchy that are often restructured when some classes are added into another part of the hierarchy?”

We are now developing a fact extractor plugin in Eclipse to remove the dependence on external tools [29,30], which enable us to handle more types of relations other than class hierarchy, such as field access, method incoming and outgoing calls, etc. This will not
only enrich the accuracy and types of output of UMLDiff algorithm, but also enable us to investigate the evolution of general static system models instead of class models.

We also plan to conduct a similar case study on a much more complex software system than Mathaino, Eclipse, an open source Java IDE, which is built on an extensible plugin framework. The core of Eclipse has more than 60 plugins, most of which have several dozens of revisions. Its core plugins have been divided into several subgroups, such as compare support, team support, search, user interface, etc., which have been developed in separate IBM research branches.

We believe Eclipse provides a good test bed to evaluate the scalability of our method in the real industrial world. Furthermore, we are expecting to be able to enrich the taxonomy of class evolution types, and study further the influence of various factors on class evolution, through which we hope we could summarize a systematic way to optimize various thresholds in order to provide the best possible advice from system evolution.

Acknowledgment

This work was supported by an Eclipse Innovation Grant and by CSER, the Consortium for Software Engineering Research, and NSERC the National Sciences and Engineering Research Council of Canada.

References