

# UMLDiff: An Algorithm for Object-Oriented Design Differencing

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

## Abstract

This paper presents *UMLDiff*, an algorithm for automatically detecting structural changes between the designs of subsequent versions of object-oriented software. It takes as input two class models of a Java software system, reverse engineered from two corresponding code versions. It produces as output a change tree, i.e., a tree of structural changes, that reports the differences between the two design versions in terms of (a) additions, removals, moves, renamings of packages, classes, interfaces, fields and methods, (b) changes to their attributes, and (c) changes of the dependencies among these entities. *UMLDiff* produces an accurate report of the design evolution of the software system, and enables subsequent design-evolution analyses from multiple perspectives in support of various evolution activities. *UMLDiff* and the analyses it enables can assist software engineers in their tasks of understanding the rationale of design evolution of the software system and planning future development and maintenance activities. We evaluate *UMLDiff*'s correctness and robustness through a real-world case study.

**Categories and Subject Descriptors:** D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement –*restructuring, reverse engineering, and reengineering.*

**General Terms:** Design

**Keywords:** Design differencing, structural evolution, design understanding, design mentoring

## 1 Introduction

Change is an essential feature of the evolutionary development of object-oriented software systems and recognizing the changes that a system has gone through its lifecycle is essential to understanding how and why a system has reached its current state. It is, therefore, of critical importance that software engineers are able to understand the various types of design-level structural evolution that an object-oriented system may have gone through, such as refactorings involving moving features among classes, restructuring of data structures or class interfaces, changes to the interactions between classes and so on. These elementary structural-evolution operations are usually intended to improve the quality of the software system, such as its understandability, extensibility and maintainability. Thus, recognizing them is crucial not only for understanding the system design and its evolution but also for obtaining

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'05, November 7–11, 2005, Long Beach, California, USA.  
Copyright 2005 ACM 1-58113-993-4/05/0011...\$5.00.

an accurate picture of the quality requirements of the system so that it can be consistently evolved.

There has been some work [4,5,6] that models the *changes* of a system's components in terms of *CVS-like deltas*, which record lines of code that have been added, deleted, or changed, as reported by *GNU diff-like* tools. These approaches are simple to implement, since it is easy to extract the deltas from a versioning system, such as Concurrent Version System (CVS). Such delta reports are intended to assist software developers in merging different revisions of the system source code. However, *GNU diff-like* tools are essentially lexical-differencing tools and ignore the high-level logical/structural changes of the software system. When the intention is to build an accurate evolutionary history of a software system, *GNU diff* misses a lot of pertinent information. For example, a class renaming or a method movement to another class would most likely be reported as two separate activities: the original entity has been removed and the modified one has been added.

Source-code metrics [2,13] and clone detection [19,22] may help to infer the renamings and moves of design entities. Unfortunately, code-line metrics are at too low level of abstraction and do not necessarily correspond to the developers' intuitions about the system. Consistently maintained change documentation [5,6], if it exists, is a reliable source of information as to what has been changed and what is the rationale behind the change. However, more frequently than not, documentation is vague and incomplete about what has actually been modified. Visualization of low-level data, such as *CVS-like delta* and code metrics, may help to capture the higher-level connections between moved and renamed elements. But, visualization approaches [4,13,19] are inherently limited because they assume a substantial interpretation effort on behalf of their users and become "unreadable" for large systems.

Clearly, there is a need for automatic tools that can assist software engineers to reason at the design level about which structural changes have occurred in long-lived evolving software systems and why. In this paper, we present an automated UML-aware structural-differencing algorithm, *UMLDiff*. It takes as input two class models of an object-oriented software system, reverse engineered from two corresponding code versions. *UMLDiff* traverses the two class models, identifies corresponding entities based on their *name* and *structure similarity*, and produces a *change tree*, i.e. a tree of structural changes, which records the changes between the two system versions in terms of additions, removals, moves, renamings of various types of design entities, such as packages, classes, interfaces, fields and methods in Java, the changes to their attributes, such as visibility and modifiers, and the changes of relations among these entities.

*UMLDiff* is at the core of our design-evolution analysis work, which has been implemented in the JDEvAn tool (Java Design Evolution and Analysis). JDEvAn has been developed as an Eclipse plugin with the objective of investigating the change pat-

terms of software evolution, exploring the underlying motivations behind them, and guiding future development and maintenance activities. *UMLDiff* enables JDevAn to build a detailed and accurate picture of the software system and its components' design evolution without requiring explicit evolution documentation in the form of consistent modification requests and change logs.

The rest of the paper is structured as follows. Section 2 relates this work to previous research. Section 3 describes the meta-model assumed by *UMLDiff* and Section 4 discusses the algorithm in detail and summarizes a taxonomy of structural changes reported by *UMLDiff*. Section 5 discusses the visualization of structural changes. Sections 6 and 7 evaluate *UMLDiff* and argue for its usefulness, respectively. Concluding remarks and plans for future research are outlined in Section 8.

## 2 Related work

*GNU diff*, a text-comparison tool, is commonly used to reconstruct the changes between subsequent versions of a software system. Its output, *CVS-like deltas*, together with modification requests and bug reports, have been substantially used for aiding software evolution and maintenance activities [4,5,6,14]. However, since tools like *GNU diff* consider a software system as a set of files containing lines of text, they report changes at the lexical level and ignore the high-level logical-structure changes of the software system.

There exist other differencing techniques that take into account various types of information in addition to lines of text. Zaremski and Wing [27] investigated signature matching, such as for example the comparison of the types of a function's input and output parameters, for retrieving reusable software components. Ostertag et al. [17] relate the software library components with manually-defined features and terms through domain analysis. They also define manually the weighted subsumer and feature graph over components, based on which component similarity may be computed. Semantic Diff [11] makes use of the local dependency graph and works at the intra-procedure level only not the object-oriented system as a whole. Horwitz developed a technique [10] for detecting statement-level semantic and textual modifications based on augmented control-flow graph; this method is applicable to a simplified C-like programming language and is not suitable for object-oriented software.

However, object-oriented software systems are better understood in terms of structural and behavioral models, such as UML class and sequence diagrams. Unfortunately, syntax-specific structural differencing algorithms report changes in terms of their own primitives. For example, XML-differencing algorithms [29] report changes to XML elements and attributes, ignoring the domain-specific semantics of the concepts represented by these elements: XML-differencing tools of XMI (XML Metadata Interchange) representations of UML models are agnostic of the UML semantics and thus generally do not correspond to the developer's intuition very well.

Within various UML modeling tools, there are UML-differencing methods, such as [16], that detect differences between subsequent versions of UML models, assuming that these models are manipulated through the tool that assigns persistent identifiers to all model elements. This capability is clearly irrelevant when the whole development team does not use the same tool for all their development activities, which is usually the case. In contrast, *UMLDiff*, aware of UML semantics, is able to infer renamings and moves of various design entities based on their name and structure similarity.

There has been some work at investigating the use of comparative analysis of different snapshots of a software system for drawing inferences regarding its evolution. Demeyer et al. [2] define four heuristics based on the comparison of source-code metrics of two subsequent system snapshots to identify refactorings of three general categories. Rysseberghe [19] investigated the use of clone-detection to identify move and renaming refactorings. In contrast, *UMLDiff* enables richer and more accurate analyses based on its structural-change reports.

Ryder's group also work on comparative analysis of structural changes [18]. They define a set of atomic changes derived from the comparison of the abstract syntax trees of corresponding classes in two versions of a project. Apiwatanapong et al. [1] use the enhanced control-flow graph to model methods of object-oriented programs and identify similarities and differences between two methods based-on graph isomorphism. The major objective of their work is to analyze the impact of changes on test cases, while our work is aimed at recovering higher-level design evolution knowledge.

There has been some work at analyzing the changes of software at the design level. For example, Egyed [3] has investigated a suite of rule- and constraint-based and transformational comparative methods for checking the consistency of the evolving UML diagrams of a software system. Spanoudakis and Kim [21] developed a probabilistic message matching algorithm that detects the overlaps between messages that are likely to signify the invocation of operations with the same implementation expressed in UML object interaction diagrams and check whether the overlapping messages violate the consistency rules. 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 software evolution understanding and mentoring.

In terms of research objectives, the project most similar to our work is BEAGLE [22] developed by Tu and Godfrey. Their tool is also aiming at analyzing the structural evolution of software system, which depends on origin analysis to determine the "origin" of "new" files. Their recent work [8] also uses origin analysis to detect the merging and splitting of source-code entities. Origin analysis works at the file-structure level: it detects old functions as the "origin" of new ones based on a combination of clone detection and name and call-relation matching assuming an interactive step for detecting file merging and splitting. In contrast, *UMLDiff* is automated and examines the logical design of the system to recognize the design-level structural evolution of that system.

## 3 The meta-model

*UMLDiff* compares the logical view [12] of object-oriented software systems, which concerns classes, the information they may own, the services they can deliver, and the associations and relative organization among them. Its primary input is the system's source code, residing in a versioning system. JDevAn's fact extractor recovers a data model of the subject system's class design. Essentially, the data meta-model is defined according to the semantics of the UML static-structure model [15], which is formally a graph,  $G_D (V_D, E_D)$ . The nodes set,  $V_D$ , contains program entities of the types supported by the Java programming language<sup>1</sup>, i.e., *package*, (*anonymous*) *class*, *interface*, *array type*, *primitive type*, *field* and

<sup>1</sup> The focus on Java is pragmatic; the JDevAn is not restricted to any specific object-oriented programming language, since its data meta-model is essentially defined according to the UML semantics. Its design and implementation are extendible to other programming languages, assuming the availability of appropriate fact extractors.

block (including method, constructor and initializer). In addition, a virtual root node is included in each model, corresponding to the system-version as a whole.

Each extracted entity is described in terms of its type, name, visibility (public, protected, private, or default if declared without any access modifier) and non-access modifiers (static, final, synchronized, volatile, transient, native, abstract, strictfp), location in the source file, whether it belongs in the system source code or in a library, and the version of the system in which it belongs. The name of array types is in the form of “basetypequalified-name.dimension”. The name of packages, named classes, interfaces and fields is their declared identifier. The name of methods and constructors is in the form of “identifier(paramtype\_list)”. JDevAn’s fact extractor assigns names to anonymous classes and initializers as follows: for anonymous classes, “new super-type\_identifier”; for class initializers, “{class\_identifier.\$initializer\_number}”; for field initializers, “{field\_identifier=...}”. Finally, a fully-qualified prefix is added in front of the names of library entities.

The edge set,  $E_D$ , contains tuples of the form ( $relation, v_1, v_2$ ), where  $v_1$  and  $v_2$  are nodes and  $relation$  is a UML dependency between them. The supported dependency types are *containment*, *declaration*, *inheritance*, *interface implementation*, *field read/write* and *method call* (including *this*, *super*, and *constructor call*), *class creation*, *field data type*, *method return type*, *parameter type*, and *exceptions declared and thrown by a method*. The virtual root node (corresponding to a system-version) is assumed to *contain* all the package nodes declared in that system-version.

JDevAn implements a fact extractor based on the Eclipse Java DOM/AST model [28]. In particular, it implements an abstract syntax tree (AST) visitor to visit the appropriate source code entities and relations as the Java program is being parsed. The extracted ground model facts are stored in a PostgreSQL relational database (*entity* and *relation* table). Based on these ground facts, a variety of derived facts are inferred, such as top-level type or member type, methods declared in a class or an interface, inheritable and inherited fields/methods, and so on. In most cases, derived facts are defined as database views; however, to improve JDevAn’s performance, some frequently used derived relations, such as method implementations and overrides, class/interface usage, etc., are computed and stored into the database tables at the end of fact-extraction process.

To address the fact that PostgreSQL lacks recursive computation capabilities, which is essential to computing the transitive closure of various relations among entities, Simon’s transitive closure algorithm [20] has been implemented as a database server-side extension to compute, at the end of the fact-extraction process, the transitive closure of the containment and inheritance hierarchy, field read/write, method call, and class/interface usage relations, which are populated in the corresponding *transclosure* table.

Furthermore, the number of times that a field is read/written, a method is called, a class is created, and a class/interface is used is recorded. These numbers are subsequently used to compute structural similarity (see algorithms 2 and 3).

## 4 Design Comparison with UMLDiff

Given two versions, A and B, of a software system and their corresponding models - instances of the meta-model discussed in section 3 - UMLDiff recovers the design-level structural changes that occurred as the system evolved from A to B.

UMLDiff assumes a “principled” usage of the versioning system: if a substantial number or a complex set of changes are made

to a particular version before it is stored back in the version-management system, the accuracy of UMLDiff will most likely suffer (see section 6 for a detailed discussion on this subject).

UMLDiff is a domain-specific structural-differencing algorithm, aware of the UML semantics. As per the adopted meta-model, the software system is modeled as a directed graph. Table 1 lists the relations and the types of entities that they relate, which induce a containment-spanning tree on the directed graph.

**Table 1. The children of design entities**

Entity type	Type of children of entity
Virtual root	Packages, array types (assumed to be <i>contained</i> )
Package	Top-level classes and interfaces it <i>contains</i>
Class	Fields, methods, constructors, initializers, inner classes and interfaces it <i>declares</i>
Interface	Fields, methods, inner classes and interfaces it <i>declares</i>
Field	Initializer it <i>contains</i>
Block	Local classes, anonymous classes it <i>contains</i>

Furthermore, based on UML semantics, the containment and declaration relationship defines a logical partial order over entity types:  $virtual\ root > package > (class, interface) > (field, block)$ . Conceptually, UMLDiff traverses the containment-spanning trees of two compared models, moving from one logical level to the next in both trees at the same time. It starts at the virtual root (system-version) level, progressing down to packages, classes and interfaces, and finally, fields and blocks. At each logical level, it identifies corresponding entities of the same type as representing a single conceptual entity in two versions of the system. UMLDiff recognizes that an entity  $e_1$  in version A and an entity  $e_2$  in version B are the “same”, i.e., they correspond to the same conceptual entity, when (a) they have the same or similar name (*name-similarity heuristic*), or (b) they have similar relations to other entities, already established to be the “same” (*structure-similarity heuristic*).

Name similarity is a “safe” indicator that  $e_1$  and  $e_2$  are the same entity: it is indeed a rare phenomenon that an entity is removed and a new entity with the same name but different behaviour is added to the system. UMLDiff recognizes same-name entities first and uses them as “landmarks” to subsequently recognize renamed and moved entities. When an entity is renamed or moved, as is frequently the case with refactorings aimed at improving the extensibility and maintainability of the system, its relationships to other entities, such as the members it contains, the fields it reads/writes, the methods it calls or is called by, etc., tend to remain the same for the most part. Therefore, by comparing the relationships of two same-type entities renamings or moves can be inferred: if they share “enough” relationships to known-to-be-same entities they are the “same”, even though their names (renamed) and/or their parent entities in the containment-spanning tree (moved) are different. Whenever two entities are identified as renamings or moves, this knowledge is added to the current landmarks’ set and is used later on to further match as yet-unmatched entities. This process continues until it reaches the logical-leaf level of the spanning trees and all possible corresponding pairs of entities have been identified.

### 4.1 Similarity metrics

Let us now discuss in detail the two heuristics (name-similarity and structure-similarity) for recognizing the conceptually same entities in the two compared system versions. These two heuristics, based on the semantics of the object-oriented design domain, enable UMLDiff to recognize that two entities are the “same” even after they have been renamed and/or moved. The key to determining

such correspondence between entities is to compare the similarities between them, both at the lexical and at the design-structure level. Note that in the following discussion, the term “matched entities” refers to same-name entities, while “general-matched entities” refers to matched, renamed, and moved entities.

#### 4.1.1 Name similarity

The longest common subsequence (LCS) algorithm is frequently used to compare strings, such as program identifiers. A commonly used metric [8] based on LCS, is shown in eq. 1:

$$\text{length}(\text{LCS}(s_1, s_2)) * 2 / (\text{length}(s_1) + \text{length}(s_2)) \quad (1)$$

LCS reflects the lexical similarity between two strings, but it is not very robust to changes of word order, which is common when renaming an entity, changing the order of method parameters, etc. For example, according to eq. (1), *AddVerticalAction* (score 0.8) is more similar to *VerticalDrawAction* than *DrawVerticalAction* (score 0.77).

```
double nameSimilarity(s1, s2)
1. HashSet pairs1 = pairs(s1.toUpperCase());
2. HashSet pairs2 = pairs(s2.toUpperCase());
3. int union = pairs1.size() + pairs2.size();
4. pairs1.retainAll(pairs2);
5. int intersection = pairs1.size();
6. return intersection*2.0/union;
```

**Algorithm 1. Computing name similarity**

To address this problem, we have defined a new name-similarity metric (see algorithm 1) in terms of how many common adjacent character pairs are contained in two compared strings. The *pairs(x)* function returns the pairs of adjacent characters in a string *x*. By considering adjacent characters, the character ordering information is to some extent taken into account. The similarity between two strings *s<sub>1</sub>* and *s<sub>2</sub>* is twice the number of character pairs that are common to both strings divided by the sum of the number of character pairs in the two strings. It is a value between 0 and 1. We choose to compute name-similarity case-insensitively, since it is common that the name is misspelled with the wrong case or is modified with just case changes. According to algorithm 1, *DrawVerticalAction* (score 0.88) is more similar to *VeritcalDrawAction* than *AddVerticalAction* (score 0.73). This metric is cheap to calculate and, based on our case studies, it seems to result in more intuitive scoring.

#### 4.1.2 Structure similarity

Table 2 lists the entities and relations’ facts that *UMLDiff* uses to compute the structure-similarity between two entities of the same type. These facts are readily available in the extracted data model (see section 3). In principle, other facts could be used as well and the JDEvAn implementation is extendible to include them (see algorithm 2 and 3). For example, interface implementation can be included when determining class renamings, the data type can be included when determining potential field moves, and so on.

**Table 2. Facts for computing structure similarity**

Entity Type	Entity and relationship facts
Package	The top-level classes and interfaces it <i>contains</i>
Named class and interface	The fields, methods, constructors, inner classes and interfaces it <i>contains</i> ; classes/interfaces it <i>uses</i> and is <i>used by</i>
Field	The methods/constructors that <i>read</i> and <i>write</i> it
Method and constructor	Type of parameters it <i>declares</i> ; fields it <i>reads</i> and <i>writes</i> ; methods/constructors it <i>calls</i> and is <i>called by</i>

The algorithm 2 returns a normalized value indicating how similar the two sets, *e<sub>of\_r1</sub>* and *e<sub>of\_r2</sub>*, are. The *e<sub>of\_r1</sub>* and *e<sub>of\_r2</sub>* sets contain entities that are related to the two compared entities, *e<sub>1</sub>* and *e<sub>2</sub>*, according to a given relation type (*relation\_type*) (lines 1,2). It essentially computes the intersection of the two sets based on a given *equals* function (line 7). This *equals* function can examine whether the names of the entities *er<sub>1</sub>* and *er<sub>2</sub>* are the same, or whether the two entities *er<sub>1</sub>* and *er<sub>2</sub>* have already been established as a pair of general-matched entities. The intersection set effectively incorporates knowledge of any “known landmarks” (previously established general-matched pairs of entities).

```
double structureSimilarity(e1, e2, relation_type)
1. Set e_of_r1 = getEntitiesOfRelation(e1, relation_type);
2. Set e_of_r2 = getEntitiesOfRelation(e2, relation_type);
3. if(e_of_r1.size == 0 and e_of_r2.size==0)
4.   pow++; return power(nameSimilarity, pow);
5. int beforecount=0, aftercount=0;
6. for all er1 in e_of_r1 and all er2 in e_of_r2
7.   if(er1.equals(er2)) {
8.     beforecount+=getCount(e1, er1, relation_type);
9.     aftercount+=getCount(e2, er2, relation_type);
10.    e_of_r1.remove(er1);
11.    e_of_r2.remove(er2); }
12. int beforeleftcount=0, afterleftcount=0;
13. for all er1 left in e_of_r1
14.   beforeleftcount+=getCount(e1, er1, relation_type);
15. for all er2 left in e_of_r2
16.   afterleftcount+=getCount(e2, er2, relation_type);
17. int min=min(beforecount, aftercount);
18. int max=max(beforecount, aftercount);
19. return min*1.0/(max+beforeleftcount+afterleftcount);
```

**Algorithm 2. Computing structure similarity**

The *getCount()* function (lines 8,9,14,16) returns how many times the given type of relation appears between two given entities. It returns constant 1 for such relations as *containment*, *declaration*, etc., while for the relations, such as field read/write, method call, class creation, class/interface usage, it retrieves the count aggregated when extracting the corresponding facts.

The challenge is how to determine the similarity when both entity sets *e<sub>of\_r1</sub>* and *e<sub>of\_r2</sub>* are empty (line 3), such as for example, when the methods do not write any fields or make any outgoing calls. It has been our experience that, in such cases, setting the structure-similarity to be simply 0 or 1 is not desirable: lacking any explicit evidence of similarity, assuming that the structure is completely the same or completely different skews the subsequent result. Therefore, *UMLDiff* uses the *nameSimilarity* with an increasing exponent, if there are no entities that have the given type of relation with the two compared entities. The effect is dampened as more empty sets are encountered. For example, suppose we compare the structure-similarity of two methods in the order of types of parameters, field reads, field writes, and so on (see line 4 of algorithm 3). The two compared methods declare no parameters and access no fields. The algorithm 2 returns *nameSimilarity*<sup>1</sup> for comparing types of parameters, *nameSimilarity*<sup>2</sup> for field reads, *nameSimilarity*<sup>3</sup> for field writes, and so on.

#### 4.1.3 Overall similarity metric

Given two entities *e<sub>1</sub>* and *e<sub>2</sub>* of the same type, their similarity is computed by algorithm 3. The algorithm 3 is used in *identifyRename* and *identifyMove* for determining potential renamed and moved entities. *N* is the number of different types of relations when determining renamings or moves for a particular entity type as defined in Table 2. *UMLDiff* uses a user-defined threshold value

(*rename\_threshold* in algorithm 6 and *move\_threshold* in algorithm 7) above which two entities can be considered as the “same” entity renamed or moved. If, for a given entity in the “before” version, there are several potential matches above the user-specified threshold in the “after” version, the one with the highest similarity score is chosen. The higher the threshold is, the stricter the similarity requirement is. The smaller the threshold is, the riskier the renamings and moves are.

```
double computeSimilarityMetric( $e_1, e_2$ )
1.  nameSimilarity=nameSimilarity( $e_1.name, e_2.name$ );
2.  int pow=0;
3.  double metric=0.0;
4.  for all relation_type as defined in Table 2
5.    metric+=structureSimilarity( $e_1, e_2, relation\_type$ );
6.  return (nameSimilarity+metric)/(nameSimilarity+N);
```

**Algorithm 3. Computing similarity metric**

## 4.2 The UMLDiff algorithm in detail

The *UMLDiff* algorithm is described in pseudocode in algorithm 4. The *results* queue contains all processed entities; the *match* queue contains the matching entities discovered at the current logical level so far; the *next* queue contains the matching entities found so far at the logical level below. This set of queues is designed to handle cycles in the meta-model. For example, when a class has nested classes, both the enclosing and the nested classes belong in the same logical level (they are all classes) although the nested classes are at a level below that of the enclosing class in the containment-spanning tree. Nested classes should be processed in the same way as the enclosing class but differently from the fields and methods of the enclosing class. Thus, the matching pairs of nested classes of a class are added to the end of the *match* queue, while the matching pairs of fields and methods of that class are added into the *next* queue (see algorithm 5 and 6).

```
UMLDiff( $vr_1, vr_2$ )
1.  Queue next, match, results;
2.  level=VIRTUAL_ROOT; next.add([ $vr_1, vr_2$ ]);
3.  while (next != null) {
4.    match = next.duplicate(); next.clear();
5.    while(match != null) {
6.      Set s = match.pop(); temp_match.add(s);
7.      HashMap[]  $c_1 = s[1].getChildren()$ ;
8.      HashMap[]  $c_2 = s[2].getChildren()$ ;
9.      identifyMatch( $c_1, c_2, match, next$ ); }
10.   match.addAll(temp_match);
11.   while(match != null) {
12.     Set s = match.pop(); results.add(s);
13.     HashMap[]  $c_1 = s[1].getChildren()$ ;
14.     HashMap[]  $c_2 = s[2].getChildren()$ ;
15.     if(!childrenMatchChecked( $s[1], s[2]$ ))
16.       identifyMatch( $c_1, c_2, match, next$ );
17.     identifyRename( $c_1, c_2, match, next$ ); }
18.   HashMap[]  $m_1 = generateMoveCandidates(vr_1, level)$ ;
19.   HashMap[]  $m_2 = generateMoveCandidates(vr_2, level)$ ;
20.   identifyMove( $m_1, m_2, next$ );
21.   level++; }
22. results.addAll(markAllUnmatchEntities( $vr_1, remove$ ));
23. results.addAll(markAllUnmatchEntities( $vr_2, add$ ));
24. diffAttributesAndDependencies(results);
```

**Algorithm 4. UMLDiff**

The algorithm starts at the virtual roots of the two models under comparison,  $vr_1$  and  $vr_2$ ; these two nodes are placed into the *next* queue as the first pair of matched entities (line 2). First (lines 5-9), *UMLDiff* proceeds to identify all the same-name children of

the pairs of general-matched entities. In the next step (lines 11-17), the pairs of general-matched entities are fetched from the *match* queue and their children, with as yet-undetermined status, are retrieved (through *getChildren()*) and placed into two arrays of HashMaps  $c_1$  and  $c_2$ . The number and entry type of the HashMap depends on the type of the parent entity, as described in Table 1. The renamed entities in these two children sets are identified and inserted as new pairs into the corresponding *next* or *match* queue depending on their logical level. After identifying all matched and renamed entities, all children of the current logical level entities, with yet-undetermined status, are collected (lines 18,19) and placed into two arrays of HashMaps  $m_1$  and  $m_2$ , and they are considered as move candidates.

After all pairs of matched, renamed and moved entities at this logical level of the containment-spanning trees have been identified, the algorithm proceeds to the next logical level (line 21).

This process continues until the *next* queue is empty (line 3). Finally, all unmatched entities contained in the spanning tree rooted at  $vr_1$  are assumed to have been *removed* and all unmatched entities in  $vr_2$  are assumed to have been *added* when system evolves from  $vr_1$  to  $vr_2$  (lines 22,23). The *results* queue contains all *matched*, *renamed*, *moved*, *added* and *removed* entities. The differences between their attributes and dependencies are then computed, which will be discussed in section 4.3.3 and 4.3.4.

In the remainder of this section, we present the details for determining matched, renamed, and moved pairs of entities.

### 4.2.1 Determining matching

*UMLDiff* assumes that enough design entities remain the “same” between two consecutive versions of the system, which serve as the “landmarks” to determine renamed and moved pairs of entities. By “same”, we mean that two corresponding entities of the same UML type have the same names, although their children, attributes, and dependencies with other entities may be different. Of course, a developer can remove a design entity, and then add a new one of the same type with the same name but different functionality. However, this case should rather be a rare exception.

```
identifyMatch( $set_1, set_2, match, next$ )
1.  for all  $e_1$  of a particular entity type in  $set_1$  {
2.     $e_2 = set_2.search\_entity(e_1.name)$ ;
3.    if( $e_2 != null$ )
4.      if( $e_2.level==level$ ) match.add([ $e_1, e_2$ ]);
5.      else next.add([ $e_1, e_2$ ]); }
```

**Algorithm 5. Matching the pair of same-name entities**

Given a pair of general-matched entities  $s[1]$  and  $s[2]$ , *getChildren()* (lines 7,8,13,14 of algorithm 4) returns their corresponding children with yet-undetermined status in the form of an array of HashMaps. Each entry of HashMap is in the form of  $\langle name, entity \rangle$ , where *entity* is the child of the given parent and *name* is its corresponding name as described in section 3. In algorithm 5, for a given entity  $e_1$  of a particular type, *search\_entity()* (hash search) retrieves the same *name* entity  $e_2$  of the same entity type from  $set_2$ . Next, depending on the logical level of the new matched pair of entities, it is added into the *match* or the *next* queue.

### 4.2.2 Determining renamings

*UMLDiff* attempts to identify these entities that have no same-name counterpart in the same parent context as renamed entities. To that end, algorithm 6 calls *computeSimilarityMetric* to examine the similarity of the names of the two candidate entities and the similarity of their relationships with other entities. This similarity-ranking algorithm reports, given  $e_1$ , the  $e_2$  with the highest similar-

ity score (above the user-defined threshold) as the entity renamed from  $e_1$ .

```

identityRename(set1, set2, match, next)
1.  for all e1 of a particular entity type in set1 {
2.    double highestmetric=0;
3.    for all e2 of the same entity type in set2 {
4.      double metric=computeSimilarityMetric(e1, e2);
5.      if (metric > highestmetric)
6.        highestmetric=metric; }
7.    if (highestmetric>rename_threshold)
8.      if (e2.level==level) match.add([e1, e2]);
9.      else next.add([e1, e2]); }

```

**Algorithm 6. Recovering the pair of renamed entities**

*UMLDiff* only checks renamings within the context of two general-matched entities, such as the renaming of a method within a class. Identifying renamings between two arbitrary entities, such as the case of method moved from one class to another and its identifier subsequently renamed, would be computationally expensive. If the developers use the versioning system in a principled manner, such moves and renamings can be identified separately by *UMLDiff*.

#### 4.2.3 Determining moves

Finally, *UMLDiff* proceeds to examine those entities that have not yet been identified as matches or renamings and to consider whether they may have been moved from one part of the system to another. It collects, through *generateMoveCandidate* (lines 18,19 of algorithm 4), all the same-name status-undetermined entities of the same type into an array of HashMaps and considers them as potential move candidates. Each entry of the HashMap is in the form of  $\langle name, Set \rangle$ , where *Set* contains the entities of a particular entity type that have the same *name*. For methods, we use their identifiers instead of their full name (see section 3) to construct the corresponding HashMap, which enables the identification of change of “move a method and modify its parameter list at the same time”. Again, the similarity metric is used to distinguish the pairs of entities that have really been moved.

```

identifyMove(set1, set2, next)
1.  for all entity_name of a particular entity type in set1 {
2.    Set entity1set=set1.search_entity(entity_name);
3.    Set entity2set=set2.search_entity(entity_name);
4.    for all e1 in entity1set {
5.      double highestmetric=0;
6.      for all e2 in entity2set {
7.        if(isGeneralMatch(e1.parent, e2.parent) continue;
8.        double metric=computeSimilarityMetric(e1, e2);
9.        if(metric > highestmetric)
10.         highestmetric=metric; }
11.     if(highestmetric>move_threshold) next.add([e1, e2]); }}

```

**Algorithm 7. Recovering the pair of moved entities**

#### 4.3 A taxonomy of structural changes

*UMLDiff* reports the structural changes between two subsequent versions of a software system in terms of changes to various types of design entities, to their attributes and to the relations among them. The structural changes are stored in the table *status* in the form of  $\langle scategory, stype, prev, next \rangle$ . *scategory* and *stype* represent different categories and types of structural changes as discussed below. *prev* and *next* are integer array containing the id of entity, and if necessary, the id of the related entity and their attributes. For example, the tuple  $\langle read, match\_up, [100, 150, 3], [400, 475, 5] \rangle$  represents the change: field of id 100 is read by method of

id 150 in the original version; this relation also holds in the subsequent version with their counterparts, field of id 400 and method of id 475. However, in the original version there are 3 instances of this relation while in the subsequent version this number increases to 5.

#### 4.3.1 Changes to named entities and their children

All types of named design entities and their children can be *added*, *removed*, *matched*, *renamed* and *moved* except for some special cases discussed below.

*UMLDiff* does not consider the possibility of renamings and moves of design entities from libraries. JDEvAn’s fact extractor extracts “just enough” entities from the referred libraries; extracting all the library entities would be impractical. Thus, *UMLDiff* does not have all the facts to determine structural changes to entities in libraries. The *matched*, *added*, and *removed* status of library entities indicates that they are still referred, newly referred, or no more referred in the source code when the system design evolves.

Array types are compared based on only their names. They can be *matched*, *added*, or *removed*.

The renamings of methods and constructors include changes of their identifiers and/or modifications to their parameter list. *UMLDiff* does not handle changes of the type “move a method plus change its identifier”, but it reports the changes of “move a method plus change its parameter list” (see sections 4.2.2 and 4.2.3). Furthermore, *UMLDiff* does not consider moves of constructors (it makes no sense to do so).

Since abstract-class and interface methods (implicitly abstract) do not have any field-reads/writes and outgoing-calls (although they may have incoming-calls), it is really difficult and error-prone to identify the renamings and moves of abstract methods. Therefore, *UMLDiff* does not consider the moves of abstract methods, while it identifies their renamings by checking if their corresponding implementation methods are identified as renamings.

In the case of regular Java software systems, the moves of packages do not make sense and *UMLDiff* does not consider these changes. However, for the case of systems that consist of a set of subprojects, such as a set of related plugins within the Eclipse platform, it is straightforward to incorporate an additional *subproject* layer between the *virtual root* and *package* layers in algorithm 4 and thus enable the identification of the moves of packages.

Finally, *UMLDiff* does not consider the moves of inner classes and interfaces from their declaring types to packages, and vice versa in its differencing process, which may result from such refactorings as *converting member type to top level*; such refactorings can be easily identified with post-processing of the *UMLDiff* results by querying the pair of newly added top-level type and the removed same-name member type that there exist fields and/or methods moved between them.

#### 4.3.2 Changes to anonymous entities

Initializers are compared based on only their names. They can be *matched*, *added*, or *removed*.

Anonymous classes are a special type of nested classes. From the perspective of design evolution, one single anonymous class does not have too many effects on software design, since, according to the Java specification, anonymous classes are mainly used to avoid creating a bunch of simple subclasses or implementations of interfaces. However, as a whole, they may be indicators of design-style preferences and development habits. Since anonymous classes are specified right along with the *new* (class instantiation), in our meta-model anonymous classes are children of blocks. *UMLDiff* compares the anonymous classes of the corresponding blocks when it computes the differences of class creation between them

(see next subsection). It reports quantitatively the changes of the number of the supertypes of anonymous classes created within a block instead of comparing individual pairs, which simplifies the differencing process when there are a large amount of anonymous classes.

#### 4.3.3 Changes to the relations among entities

At the end of its differencing process (line 24 of algorithm 4), *UMLDiff* examines the changes to the relations among design entities as discussed in this subsection.

**Inheritance.** *UMLDiff* examines the class-inheritance changes between two general-matched classes,  $x$  and  $x'$ , and reports either of the two results below (mutually exclusive):

*match:* extends general-matched superclasses,  $xsup$  and  $xsup'$   
*change:* otherwise

*UMLDiff* also examines interface-inheritance (not necessarily direct) changes of two general-matched classes (interfaces),  $x$  and  $x'$ , which are reported as:

*add:* newly implements (extends) the interface  $xsup'$   
*remove:* implements (extends) no more the interface  $xsup$   
*match:* implements (extends) general-matched interfaces,  $xsup$  and  $xsup'$

**Usage.** Entity-usage relations are field-read/write, method-incoming/outgoing-call, class-creation, and class/interface-usage. *UMLDiff* examines the usage (used-by) changes between two general-matched entities,  $e_1$  and  $e_1'$  and reports them as:

*add:* newly uses (is used by) the entity  $e_2'$   
*remove:* uses (is used by) no more the entity  $e_2$   
*match, match\_up, match\_down:* uses (is used by) general matched entities,  $e_2$  and  $e_2'$  with the same, increasing, decreasing number of times.

Note that for anonymous classes, *UMLDiff* reports the differences of the creation of their supertypes instead of the anonymous classes themselves.

Class and interface usage changes are computed at the end of the *UMLDiff* process. Computing the changes of field accesses, method calls, and class creations for all pairs of general-matched fields or methods in a large software system during differencing process is time-consuming. Therefore, we decided to enable the developer to request this comparison on demand by requesting the computation of the amount of changes of field accesses, method calls, and class creations during examination of the change trees (see section 5).

**Field data type and method return type.** Changes of data type (return type) between two general-matched fields (methods),  $f$  ( $m$ ) and  $f'$  ( $m'$ ), are also reported as either of the two alternatives below:

*match:* has the general-matched data (return) type,  $dt$  and  $dt'$   
*change:* otherwise

#### 4.3.4 Changes to the attributes of entities

Finally, the visibility changes between two general-matched entities are examined. *UMLDiff* reports the visibility changes as either of the following:

*up:* changes access modifier to a less restrictive one  
*down:* changes access modifier to a more restrictive one  
*match:* the visibility stays the same

where the access modifiers can be private, default, protected, and public, in the order of more to less restrictive.

Similarly, the changes of non-access modifiers' between two general-matched entities are examined and reported as follows:

*add:* the entity declares some new non-access modifiers  
*remove:* the entity declare no more some non-access modifiers  
*match:* the entity still declare some same modifiers as before

## 5 Visualization of structural changes

To enable an intuitive means of communicating all the design-change facts produced by *UMLDiff*, we have developed the *change-tree visualization*. There are two types of change trees: inheritance and containment, which are essentially the same but follow the inheritance- and containment- spanning tree of software model respectively. Due to the space limitation, we show only the containment change tree (Figure 1) from our Eclipse case study.

As an Eclipse plugin, JDEvAn reuses and extends the visualization of Eclipse's Java DOM model. Therefore, in change tree, consistent with IDE's convention, the different icons to the left of each node represent the different object-oriented entities: *package*, *class*, *interface*, *field*, and *method/constructor*. Their different colors represent the entity's visibility. The top-right adornment shows the attributes of the entity, for example, *abstract*, *constructor*, *static*, *final*, etc. The bottom-right adornment represents method override or implementation. The only extension is the bottom-left adornment that represents the *UMLDiff* result of a particular entity: it can be the *plus* sign for *add*, *minus* sign for *remove*, *01* for *rename*, *arrow* with a *minus* sign for *move out from source*, *arrow* with a *plus* sign for *move into target*.

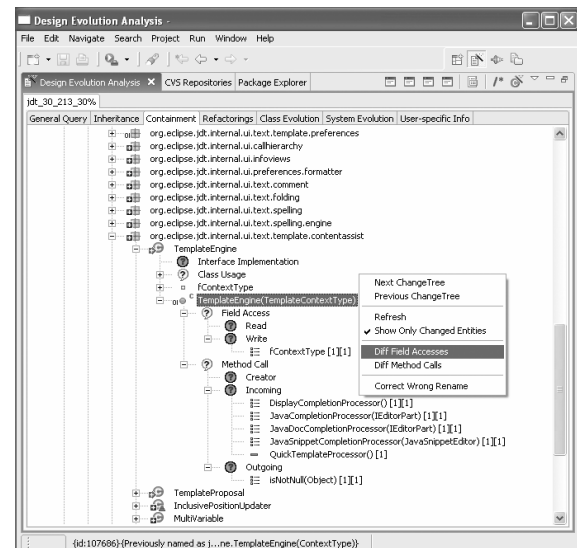


Figure 1. The JDEvAn perspective in Eclipse: a change tree

Figure 1 shows the JDEvAn perspective within the Eclipse platform, which displays the part of containment change tree between version 2.1.3 and 3.0 of JDT-related plugins in our Eclipse case study. As we can see, package `org.eclipse.jdt.internal.ui.text.template.preferences` is renamed; two classes, `TemplateEngine` and `TemplateProposal`, are moved into newly added package `org.eclipse.jdt.internal.ui.text.template.contentassist` from somewhere else (if they are selected, their corresponding source counterparts will be shown in status bar); For `TemplateEngine`, it implements no interface in both version. Field `fContextType` changes its data type from `ContextType` to `TemplateContextType` (if it is selected, the information will be shown in status bar). Its constructor changes the parameter type (see status bar). It reads no field and writes field `fContextType` once in both versions; it does not instantiate any class, while it calls `isNotNull` once in both versions and is called by the constructors of four other classes in both versions and is no more called by `QuickTempalteProcessor()` in version 3.0.

## 6 Evaluation

To date, we have conducted several case studies, analyzing the evolution of Java software systems using *UMLDiff*. They include several small student projects in a third-year software-engineering course, Mathaino – a research project developed by a single developer using a refactoring-driven process, HTMLUnit – a unit testing framework for web applications, JFreeChart – a medium size Java library, and currently, a set of large scale Eclipse plugins (still in process). In this section, we use the JFreeChart case study to evaluate the correctness and robustness of *UMLDiff*, in terms of precision and recall. Interesting readers are referred to our other publications [23,24,25,26] for the detailed evaluation on design evolution analysis and design mentoring (briefly discussed in section 7).

We chose to evaluate the accuracy and robustness of *UMLDiff* with JFreeChart because it is a substantial and realistic software system and, at the same time, it is of a manageable size, possible to inspect “manually” to establish the ground truth for the algorithm’s results. JFreeChart has been developed for more than 4 years; there were 31 major releases between the first version 0.5.6, released on December 1 2000, and the last version 1.0.0, which was released on November 29 2004. It had reached more than 800 classes in some of its versions. We applied *UMLDiff* to pair-wise compare the 31 releases of JFreeChart. A substantial number of changes, with renamings and moves threshold set to 30%, were summarized in Table 3. Its intensive and varied design-evolution history makes JFreeChart an appropriate test-bed for the evaluation of *UMLDiff*.

**Table 3. JFreeChart’s design changes at threshold 30%**

Entity renaming	2181
Entity move	1027
Entity addition	17018
Entity removal	5802
Class inheritance change	185
Interface implementation change	1025
Class usage change	14558
Datatype change	710
Non-access modifier change	303
Visibility change	855

Before we go into the detailed discussion on the *UMLDiff* effectiveness, let us first report on some performance metrics on the *UMLDiff* process. JDevAn’s database (PostgreSQL 7.4.5) runs on a linux workstation (the VMWare guest operating system), and its front-end (an Eclipse plugin) runs on Windows XP Professional. The machine is an Intel Centrino 1.6GHZ with 768M physical memory.

The fact extraction process of JFreeChart’s 31 major releases took about 2.5 hours. Computing the transitive closure on the containment hierarchy, inheritance hierarchy, field access/method call, and class/interface usage took in total about 25 minutes. Table 4 summarizes the time cost of applying *UMLDiff* to subsequent system versions. The average time required for *UMLDiffing* two subsequent versions of JFreeChart system is about 10-12 minutes. For those releases that have major changes, such as version 0.9.5, 0.9.9, 0.9.19, *UMLDiff* requires about 30-50 minutes; most of this time is used to detect moves and renamings. *UMLDiff* deals with a very large information database (227005 entities, 1451499 entity relations, and 6988582 transitive-closure facts). Therefore, if major changes were made between two compared versions, to determine potential moves and renamings, *UMLDiff* has to query the database

for retrieving their corresponding relationships and previously established matched pairs of entities, which is a time-consuming process.

**Table 4. The time cost of *UMLDiff***

Compared versions	Time(mins)
0.5.6 – , 0.7.0 – 0.6.0, 0.7.2 – 0.7.1, 0.7.3 – 0.7.2, 0.7.4 – 0.7.3, 0.8.0 – 0.7.4, 0.9.1 – 0.9.0, 0.9.6 – 0.9.5	<1
0.6.0 – 0.5.6, 0.7.1 – 0.7.0	2
0.9.0 – 0.8.0 , 0.9.2 – 0.9.1, 0.9.3 – 0.9.2, 0.9.8 – 0.9.7, 0.9.11 – 0.9.10,	4 ~ 6
0.9.7 – 0.9.6, 0.9.15 – 0.9.14, 0.9.16 – 0.9.15	8 ~ 10
0.9.4 – 0.9.3, 0.9.12 – 0.9.11, 0.9.13 – 0.9.12, 0.9.18 – 0.9.17, 0.9.20 – 0.9.19	11 ~ 14
0.9.14 – 0.9.13, 0.9.21 – 0.9.20, 1.0.0 – 0.9.21	16 ~ 20
0.9.5 – 0.9.4, 0.9.10 – 0.9.9	23 ~ 25
0.9.9 – 0.9.8	37
0.9.17 – 0.9.16	52
0.9.19 – 0.9.18	58
<b>Total</b>	<b>~370</b>

Once all pairs of successive versions are *UMLDified*, all structural changes discovered are stored in JDevAn’s database. We manually inspected the correctness of each instance against the JFreeChart source code, the accompanying Javadocs and source-code comments, and a textual change log shipped with each major release version. See table 5 and 6 for details on precision.

Recall is harder to assess since it requires knowledge of the total number of modifications of each type that have actually occurred. To develop an intuition about how good the *UMLDiff* recall is, we first run *UMLDiff* with a very low threshold for renamings, i.e., 1%. With such a low threshold, *UMLDiff* is very eager to recognize entities as renamed and thus we expected to collect all instances of renamings to use as the “actually renamed entity set” to assess renaming recalls in other configurations. With the renaming threshold set at 1%, *UMLDiff* reports 2945 instances of renamed entities; after inspecting each one of them, we established that 2154 are correctly identified. At 30% renaming threshold, *UMLDiff* reports 2077 correct renaming instances, i.e. adding up the number of correct instances of renaming package, renaming class and interface, and renaming field and method. This implies that at 30% threshold, the renaming recall is 2077/2154 (96.4%). What is also interesting to note, is that, even at this extremely low renaming threshold, the *UMLDiff* renaming precision is 2154/2945 (73.1%) is not too low. This robustness is due to *UMLDiff*’s similarity-ranking mechanism.

Next, we set the renaming threshold at 100% and the move threshold at 1%. Given this very strict criterion for recognizing renamings, only entities in the “same” parent context that have the exact same relationships with other known general-matched entities are identified as renamed. All other yet-unmatched entities are examined against other yet-unmatched entities in other parent entities: given the very low move threshold, *UMLDiff* is eager to recognize moved entities, based on even the most tenuous structure similarity results. *UMLDiff* reports 1521 instances of moves, of which 964 are correct. Therefore the moves recall at the 30% threshold is 936/964 (97.1%). Since *UMLDiff* does not consider the changes of “move an entity as well as rename its identifier” and “move member type to package, and vice versa”, the real move recall should be slightly lower than 97.1%.

Furthermore, we also evaluated qualitatively the *UMLDiff* recall against the CVS logs and release notes. With respect to the changes that the developers themselves felt interesting enough to document, over 95% of the changes recorded in these documents can be recovered by *UMLDiff* along with a detailed context.

## 6.1 *UMLDiff* correctness

Even though the precision and recall rates are quite good, it is interesting to understand “when *UMLDiff* gets confused”; let us, therefore, review the cases of erroneously reported changes. Table 5 shows the *UMLDiff* results at threshold 30%: the third column reports the number of changes of each type reported by *UMLDiff* for JFreeChart, the number of correctly identified ones in the second column and the precision in the fourth column.

**Renamed class and interface:** Seven out of 128 class/interface renamings are incorrect. Two involve pairs of JUnit test classes and five involve demo classes. All mistakenly recognized pairs of classes are very similar: the JUnit classes share methods such as `suite()`, `testEqual()`, `testCloning()` and `testSerialization()` and the demo classes have methods like `main()`, `createChart()`, and `createDataset()`. Moreover, the efferent relations of these class pairs are also similar. For example, the `suite()` methods of the JUnit test classes create an instance of `TestSuite` with parameter `TestClass.class` and their `testSerialization()` methods use the `ByteArrayOutputStream`, `ObjectOutputStream`, `ObjectOutput` classes. Finally, none of these classes have any afferent relations: the JUnit classes are launched by the JUnit framework, and the demo classes are stand-alone Java applications.

Table 5. *UMLDiff* results at threshold 30%

Type of change	#Correct	#Reported	Precision
Renamed package	29	29	100%
Renamed class/interface	121	128	94.5%
Moved class/interface	306	306	100%
Renamed field/method	1927	2024	95.2%
Moved field/method	630	721	87.3%
Data type and return type	677	710	95.4%
Visibility modifier	845	855	98.8%
Non-visibility modifier	299	303	98.7%
Class inheritance	180	185	97.3%
Interface inheritance	970	1025	94.6%
<b>Total</b>	<b>5894</b>	<b>6286</b>	<b>95.2%</b>

**Moved class and interface:** All reported instances are correct but a few instances are missed. For example, in version 0.9.5, the interface `CategoryItemRenderer` was moved from package `com.jrefinery.chart` to `com.jrefinery.chart.renderer`. At the same time, 11 new methods were added to its original 10 methods, which also changed signatures. Furthermore, out of nine classes that use this interface in both versions, only 3 pairs of them were matched. This dramatic change make the moving of interface `CategoryItemRenderer` not recognizable by *UMLDiff* at threshold 30%.

**Renamed field and method:** 97 out of 2024 field and method renamings are not correct. Most of them involve get and set methods and the fields they access. These methods are simple and short, with few relationships to other entities and present a challenge to *UMLDiff*’s structure-similarity heuristic. It is important to note here that among the 1927 correctly identified field and method renamings, there exist renamings that had no identifier similarity at all, and therefore would not have been intuitively recognized by a developer: for example, the `CategoryPlot.getDataArea()` is cor-

rectly identified as renamed to `CategoryPlot.calculateAxisSpace()` in version 0.9.10.

**Moved field and method:** The precision of recognizing moved fields and methods (87.3%) is worse than that (95.2%) of renamed fields and methods. But its recall is better than that of renamings. For example, in version 1.0.0, a superclass `AbstractPieItemLabelGenerator` was extracted from class `StandardPieItemLabelGenerator`, and 6 (all possible moves) fields and methods were reported as having moved from the subclass to the new superclass. But it does not correspond to the intuition of considering `StandardPieItemLabelGenerator.createItemArray()` as being moved to the newly added class `StandardXYSeriesLabelGenerator`, as reported by *UMLDiff*. The low precision is probably due to the fact that the potential renaming candidates are limited to yet-unmatched children of the given pair of general-matched parent entities, while all pairs of yet-unmatched children of the same type with the same identifier of all the entities at the whole logical level are considered as candidates for potential moves, which increases the difficulty for determining moves. We tried to incorporate parent-name-similarity and parent-usage-similarity when computing structure-similarity for moving fields and methods, which to some extent improves the precision.

**Other errors:** The accuracy of the reported changes to data types and return types, modifiers, visibility, and class inheritance and interface inheritance is relatively higher than that of renamings and moves. The occasional errors are due (a) to erroneously identified renamed and/or moved entities or (b) missed renamings and/or moves or (c) combined moves and identifier-renamings. If two entities are mistakenly identified as renamed or moved, their different data types, modifiers, class and interface inheritance will also be reported as changed. On the other hand, if a renaming or move is missed, the entities referring to the renamed/moved entity will mistakenly be reported as changed. For example, the interface `CategoryItemRenderer` was not identified as having moved to the new package `com.jrefinery.chart.renderer` in version 0.9.5, and, consequently, the type of field `renderer` and method `getRenderer()` of class `CategoryPlot` were identified as changed. In addition, the interface implementation of 15 `renderer` classes that implemented the `CategoryItemRenderer` interface were also identified as changed. Finally, since *UMLDiff* does not attempt to identify cases of combined identifier-renamings and moves, if a class was renamed and then moved, such as for example, `CrosshairInfo` in package `com.jrefinery.chart` in version 0.9.16 and `CrosshairState` in `com.jrefinery.chart.plot` in version 0.9.17, they will be treated as removed and newly added entities (but can be easily recovered through querying the pair of die-hard and legacy classes [23]), which might also result in the wrong data type, and inheritance and implementation changes being identified.

## 6.2 *UMLDiff* robustness

Let us now discuss how the quality of the structural changes reported by *UMLDiff* is impacted by the user-defined renaming and move thresholds, and the style of CVS usage by the development team.

**Renaming and move threshold:** To understand how sensitive *UMLDiff* is to the choice of the “right” renaming and move threshold, we experimented with a few different thresholds. Table 6 presents the *UMLDiff* results at renaming and move threshold 35%. Compared with the results at threshold 30%, six less instances of renamed classes and interfaces are reported: four of them are actual class renamings that are not recognized at threshold 35%, while the other two are incorrect instances reported at 30% but correctly ignored at threshold 35%.

**Table 6. UMLDiff results at threshold 35%**

Type of change	#Correct	#Reported	Precision
Renamed package	29	29	100%
Renamed class/interface	117	122	95.1%
Moved class/interface	303	303	100%
Renamed field/method	1888	1945	97.9%
Moved field/method	608+14	686+14	88.6%
Data type and return type	662	682	97.1%
Visibility modifier	836	841	99.4%
Non-visibility modifier	295	296	99.7%
Class inheritance	178	186	95.7%
Interface inheritance	962	1025	93.7%
<b>Total</b>	<b>5877</b>	<b>6129</b>	<b>95.8%</b>

*UMLDiff* misses the moves of three classes at 35%, CategoryAxis, CategoryPlotConstants, and AbstractRenderer into the corresponding new package in version 0.9.5. For renamed field and method, 79 less instances were reported. 29 of them are incorrect instances being filtered out at threshold 35%, while 40 are actual renamings missed at 35%. The other 10 instances are the results of 4 missed class renamings, also incorrectly ignored at this threshold.

22 correct instances of moved fields and methods are missed at threshold 35%, while 14 fields and methods of missed pairs of renamed and moved classes are identified as moves. The precision of renamings and moves at threshold 35% are slightly better than that of threshold 30%, but as expected, the recalls are slightly lower, 94.5% for renamings and 94.6% for moves. Most of disappeared instances for visibility and modifier changes are incorrect instances, and thus they get relative bigger increases in precision. Because there are 7 actual class and interface renamings and moves are not recognized by *UMLDiff* at 35%, entities that refer to these classes and interfaces are considered to be changed. This directly results in the slight decrease of precision of changes to class and interface inheritance.

**Regularity of CVS usage:** We also examined the changes that *UMLDiff* reported when comparing major releases and the changes it reported when comparing intermediate versions in order to assess the impact of not having regular and frequent CVS updates. To that end, we examined the release versions documented with the major API changes, such as version 0.8.0 and 0.9.0, or those of lower precision, such as 0.9.16 and 0.9.17.

For example, the class CombinedXYPlot in version 0.9.0 was identified by *UMLDiff* as a renaming of class CombinedPlot in version 0.8.0. In fact, the CombinedPlot was renamed MultiXYPlot on April 23, 2002 and subsequently it was renamed again CombinedXYPlot on May 23, 2002 just before release 0.9.0. Clearly, when *UMLDiff* only compares the two major releases - 0.8.0 on March 22, 2002 and 0.9.0 on June 7, 2002 - the intermediate renaming is missed. In general, the smaller the distance between two compared versions, the higher the detail of the report is likely to be.

As another example, the precision of reported renamings between version 0.9.16 and 0.9.17 is about 90.6%, which is worse than the overall precision 95.2%. 15 of 159 reported renamings are incorrect, among which six are related to renaming fields and methods ?ItemLabelGenerator of class AbstractCategoryItemRender and interface CategoryItemRenderer to ?ToolTipGenerator, which did not reflect what changes were really made, since by checking the source code we know that they were actually renamed to ?LabelGenerator. This low precision led us to further investigate the intermediate changes by taking the weekly snapshots between two major releases 0.9.16 and 0.9.17, which resulted in 12 snapshots from January 9, 2004 to March 26, 2004. *UMLDiff* was ap-

plied to these 12 weekly snapshots, which produced more accurate results. Five of six wrong instances were corrected, except for baseItemLabelGenerator of AbstractCategoryItemRenderer being still identified as renamed to baseToolTipGenerator.

Clearly, the quality of *UMLDiff* results is affected by the frequency of saving changes back to versioning system and the time duration between two compared versions. In general, *UMLDiff* will produce better and more accurate results if the changes are properly saved in time and the short time period is used between two compared system versions. In the case of UMLDiffing major releases of JFreeChart, it produces results of both good precision and good recall within threshold 30% and 40%. A threshold higher than 40%, especially 50%, produces results with tenuous precision improvement but at a significant cost of recall, while a threshold below 30% produces results with slightly better recall but much worse precision.

## 7 UMLDiff applications

Having evaluated the quality and robustness of *UMLDiff*, the next question we would like to address is “what is this good for?” In this section we briefly discuss three applications based on *UMLDiff*. These applications are currently under development – to a different degree of maturity – in JDevAn, the tool that also implements *UMLDiff*.

### 7.1 Design-evolution patterns and analyses

Important design decisions are reflected in the source code and in the way the code has changed over time; such decisions can be recognized from their effects, i.e., the design-level structural changes reported by *UMLDiff*. We have developed a suite of automated analyses, based on the structural changes produced by *UMLDiff*, to study the design evolution of object-oriented systems from multiple perspectives [23,24,26].

First, based on the elementary structural changes produced by *UMLDiff*, we have defined queries to elicit more complicated structural change patterns, such as refactorings [7]. For example, an instance of the “Extract Superclass” refactoring can be recognized through a query looking for a set of related changes: (a) the *addition* of a new class, (b) the *modification* of one or more classes from extending other class, such as java.lang.Object to extending the newly added class, and (c) the *moving* of one or more fields and methods from these existing classes to their new superclass.

In addition to refactorings, JDevAn is able to discern any general structural change patterns of interest to the user, such as for example, a large amount of classes start implementing a particular interface. These patterns represent project-specific evolution knowledge. More often than not, they are not recorded in the development log; they usually just exist in the developers’ minds, as part of their development experience. JDevAn is able to recover them and present them to developers as a set of contextual advices [26], which may be valuable to guide future development and maintenance activities.

For an evolving software system with  $N$  successive versions, *UMLDiff* can be applied  $N$  times to generate the differences between the  $(I+1)^{th}$  and  $I^{th}$  versions, where  $0 \leq I < N$  (supposing there is a virtual version 0 with no entities), resulting in a sequence of  $N$  change trees that records the structural modifications of the logical design of the subject system over time. This change-tree sequence provides an audit trail of the design-level structural evolution that the system and its components have suffered throughout their lifecycle. This trail is analyzed to produce a *system-evolution profile* and a *class-evolution profile* for each individual system class and interface, which are then analyzed to discern distinct system and

class evolution phases and styles and the sets of co-evolving classes [23,24].

## 7.2 Design mentoring

Software design is hard to learn because there are few “cut and dried” rules for determining what is correct or what needs to be improved and its application is contextual. Practical experience of designing software is essential in becoming an expert in this activity. Today, software is often developed using an evolutionary process. On one hand, this makes the design task harder since it requires evolving an existing software in a manner consistent with its design history; on the other hand, the software itself embodies examples of high-level object-oriented design principles, design patterns, refactorings, which the designers may study to acquire valuable design experiences.

In [25], we proposed some preliminary work about “mentoring” designers based on the analyses of design evolution patterns. In our recent work [26], we illustrated, through a real world case study, the JDEvAn’s capability to offer, based on capturing and analyzing the instances of recurring design evolution patterns, advice regarding potential modifications that may improve the system design. The advice is grounded in JDEvAn’s knowledge of object-oriented design principles, design and refactoring patterns and programming hints previously adopted by the system under inspection.

## 7.3 Catching-up with evolving APIs

Software reuse, of frameworks and/or libraries, is a common practice today. To some extent, it simplifies the design of new systems but, at the same time, it makes them heavily dependent on the components they reuse. Ideally, the reused components’ APIs would not change. In practice, however, as shown in Table 3, new components’ versions do change their APIs, which imply a need for the systems that use them to adapt. Migrating an application to the new APIs is tedious and disruptive work. CatchUp [9] is an attempt to relieve this burden by recording the refactorings, such as method renamings, made by the library developers within an IDE, such as Eclipse, and then replaying them by the application developers on the library client code to keep it updated. But such a tool is limited to the subset of refactorings supported by a particular IDE, and more important, it requires the library developers explicitly use the tool to record changes and ship them with the new versions of library.

In the absence of such a general tool, the application developers have to resort to studying the available documentation, such as change logs and release notes, which likely contains only a subset of the actual APIs changes (what the library developers considered important). For example, by checking the Eclipse help system, the documents “Incompatibilities between Eclipse 2.1 and 3.0” and “Adopting 3.0 mechanisms and API”, we found about 120 changes regarding the incompatible APIs. However, by UMLDiffing versions 2.1 and 3.0 of one of Eclipse plugins, org.eclipse.jdt.ui (contains about 1/6-1/7 classes of the whole Eclipse), we found 30 class/interface renamings and 12 class/interface moves, about 750 public field and method renamings, moves, and data type changes, 309 inheritance changes, and 302 visibility changes from or to public. Clearly, there are many more changes than the ones documented! UMLDiffing the old and new versions of a framework or library can serve as a good source of information for the application developers migrating an application to the new APIs.

## 8 Conclusions

In this paper, we described *UMLDiff*, an object-oriented design-structure differencing algorithm. This algorithm exhibits several

advantages over the current state of the art. Because it compares software versions at the design level, its results are more directly relevant to the evolutionary-development process than either lexical differencing or metrics differencing. Because it is aware of the domain-specific semantics, its results are more intuitive than general structure differencing algorithms. Because it is automated, it does not rely on subjective interpretation as do visualization approaches and it can provide the basis for a variety of subsequent analyses.

Our evaluation of *UMLDiff* has shown that it is sensitive to irregular CVS usage but has high precision and recall when CVS is used regularly and is fairly robust to the user’s choice of parameters. Finally, *UMLDiff* can provide support for various design-evolution activities in the context of evolutionary software development.

Future work on *UMLDiff* includes investigating the combination of different factors that affect the *UMLDiff* and coming up with the clue on deciding the appropriate set of relations and thresholds for renamings and moves in the context of the subject software project.

## References

1. T. Apiwattanapong, A. Orso and M.J. Harrold. A differencing algorithm for object-oriented programs. *Proceedings of the 19<sup>th</sup> International Conference on Automated Software Engineering*, pp. 2-13, 2004.
2. S. Demeyer, S. Ducasse and O. Nierstrasz. Finding refactorings via change metrics. *ACM SIGPLAN notices*, 2000, 35(10):166-177.
3. A. Egyed. Scalable consistency checking between diagrams - The VIEWINTEGRA Approach. *Proceedings of the 16<sup>th</sup> International Conference on Automated Software Engineering*, 2001.
4. S.G. Eick, J.L. Steffen and E.E. Sumner. SeeSoft—A tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 1992, 18(11):957-968.
5. 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.
6. M. Fischer, M. Pinzger and H. Gall. Populating a release history database from version control and bug tracking systems. *Proceedings of the 19<sup>th</sup> International Conference on Software Maintenance*, pp. 23-32, September 2003.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
8. M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 2005, 31(2):166-181.
9. J. Henkel and A. Diwan. CatchUp! Capturing and replaying refactorings to support API evolution. *Proceedings of the 27<sup>th</sup> International Conference on Software Engineering*, pp. 274-283, 2005.
10. S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation*, pp. 234-246, June 1990.
11. D. Jackson and D.A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. *Proceedings of 9<sup>th</sup> Interna-*

- tional Conference on Software Maintenance*, pp. 243-252, September 1994.
12. P. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, 1995, 12(6):42-50.
  13. M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. *Proceedings of the 4<sup>th</sup> International Workshop on Principles of Software Evolution*, pp. 37-42, 2001.
  14. M.M. Lehman and L.A. Belady. Program evolution-processes of software change. *Academic Press*, London, 1985, 538pps.
  15. *OMG Unified Modeling Language Specification*, formal/03-03-01, Version 1.5, (2003), <http://www.omg.org>.
  16. D. Ohst, M. Welle and U. Kelter. Difference tools for analysis and design documents. *Proceedings of the 19<sup>th</sup> International Conference on Software Maintenance*, pp. 13-22, September 2003.
  17. E. Ostertag, J. Hendler, R. Prieto-Daz and C. Braun. Computing similarity in a reuse library system: An AI-Based Approach. *ACM Transactions of Software Engineering and Methodology*, 1992, 1(3):205--228.
  18. B.G. Ryder and F. Tip. Change impact analysis for object-oriented programs. *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 46-53, 2001.
  19. F.V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. *Proceedings International Workshop on Principles of software Evolution*, pp. 126-130, September 2003.
  20. K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science 58, Automata, Languages and Programming*, 376-386, 1986.
  21. G. Spanoudakis and H. Kim. Reconciliation of object interaction models. *Proceedings of the 7<sup>th</sup> International Conference on Object Oriented Information Systems*, pp. 47-58, August 2001.
  22. Q. Tu and M.W. Godfrey, "An integrated approach for studying architectural evolution", *Proceedings of the 10<sup>th</sup> International Workshop on Program Comprehension*, pp. 127-136, 2002.
  23. Z. Xing and E. Stroulia. Understanding class evolution in object-oriented software. *Proceedings of the 12<sup>th</sup> International Workshop on Program Comprehension*, pp. 34-43, June 2004.
  24. Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. *Proceedings of the 20<sup>th</sup> International Conference on Software Maintenance*, pp. 242-251, 2004.
  25. Z. Xing and E. Stroulia. Design mentoring based on design evolution analysis. *OOPSLA ETX Workshop*, 2004.
  26. Z. Xing and E. Stroulia. Towards Mentoring Object-Oriented Evolutionary Development. *Proceedings of the 21<sup>st</sup> International Conference on Software Maintenance*, 2005.
  27. A.M. Zaremski and J.M. Wing. Signature Matching: A Key to Reuse. *Proceedings of 1<sup>st</sup> ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993, pp. 182-190.
  28. Eclipse, <http://www.eclipse.org>
  29. Mosell EDM Ltd, <http://www.deltaxml.com>.