Autom Softw Eng (2007) 14: 215–259
DOI 10.1007/s10515-007-0007-3

Differencing logical UML models

Zhenchang Xing · Eleni Stroulia

Published online: 26 May 2007

Abstract *UMLDiff* is a heuristic algorithm for automatically detecting the changes that the logical design of an object-oriented software system has gone through, as the subject system evolved from one version to the next. *UMLDiff* requires as input two models of the logical design of the system, corresponding to two of its versions. It produces as output a set of change facts, reporting the differences between the two logical-design versions in terms of (a) additions, removals, moves, renamings of model elements, i.e., subsystems, packages, classes, interfaces, attributes and operations, (b) changes to their attributes, and (c) changes to the relations among these model elements. In this paper, we detail the underlying metamodel, the *UMLDiff* algorithm and its heuristics for establishing lexical and structural similarity. We report on our experimental evaluation of the correctness and robustness of *UMLDiff* through a real-world case study.

Keywords Design differencing · Structural evolution · Design understanding · Design mentoring

1 Introduction

Recent research on mining software repositories has shown that interesting insights into the lifecycle of a project and the design rationale underlying its evolution history can be obtained by comparatively analyzing different aspects of the series of the software versions stored in a repository. To software engineers, it is especially important to be able to understand the design changes that their software system has

Z. Xing (✉) · E. Stroulia
Computing Science Department, University of Alberta, Edmonton, AB T6G 2H1, Canada
e-mail: xing@cs.ualberta.ca

E. Stroulia
e-mail: stroulia@cs.ualberta.ca

suffered through its evolution. Such changes include extensions with new classes and behaviors, modifications of design entities to meet new requirements or fix bugs, and refactorings, i.e., behavior-preserving restructurings involving redistribution of features among classes, restructuring of data structures or class interfaces, and changes to the interactions between classes. Knowing what changes have happened in the past is critical for understanding how the various functional requirements of the system are met and what qualities have been considered important during its development. An accurate picture of the system's design-evolution history can substantially support its further development in a manner consistent with the qualities embodied in its current design.

The research problem then becomes to accurately and efficiently recognize the changes in the system's design structure and behavior as it evolves from one version to the next. There has been some work (Eick et al. 1992, 2001; Fischer et al. 2003) modeling the *changes* of a system in terms of *code-line deltas*, reporting lines of code that have been added, deleted, or changed, as reported by *GNU diff*. These approaches are simple to implement, since it is easy to extract the deltas from a versioning system, such as Concurrent Version System (CVS). However, lexical differences do not provide much intuition regarding the design changes that the reported source-code modifications were meant to implement. For example, a small change to a single line of code may correspond to the renaming of a class to conform to a naming convention, or to the reorganization of the inheritance hierarchy to better model the underlying application domain. Source-code metrics (Demeyer et al. 2000; Lanza 2001) can recognize whether or not method extraction or inlining has occurred in general, but they do not pinpoint what specifically has been changed. Clone-detection methods can pinpoint (Rysselberghe and Demeyer 2003; Tu and Godfrey 2002) specific refactoring instances, but only for a limited set of refactoring types. Consistently maintained change documentation (Eick et al. 2001; Fischer et al. 2003) is a reliable source of information as to what has been changed and why; however, more frequently than not, documentation is vague and incomplete. Visualization of data, such as code-line deltas and source-code metrics, may communicate some intuition regarding element moves and renamings; however, visualization approaches (Eick et al. 1992; Lanza 2001; Rysselberghe and Demeyer 2003) 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 automated methods for assisting software engineers to reason, at the design level, about what changes have occurred in long-lived evolving software systems and why. In this paper, we describe *UMLDiff*, an algorithm for differencing UML logical-design models. This algorithm takes as input two logical-design models corresponding to two versions of an object-oriented software system. It traverses the two models in parallel, moving from one type of design entities to its children types; as it does so, it identifies corresponding entities, i.e., model entities that correspond to the same conceptual design entity, based on their *lexical* and *structural similarity. UMLDiff* produces as output a set of *change facts* reporting the various design changes it has discovered when comparing the two models, i.e., additions, removals, moves, and renamings of subsystems, packages, classes, interfaces, attributes and operations, and changes to the attributes and relations of these model elements.

UMLDiff is at the core of our design-evolution analysis work (Schofield et al. 2006; Xing and Stroulia 2004a, 2004b, 2005a, 2005c, 2006a, 2006b, 2006c), which has been implemented in the JDEvAn (Java Design Evolution and Analysis) tool (http://www.cs.ualberta.ca/~xing/jdevan.html). In addition to *UMLDiff*, JDEvAn also includes: a fact extractor that crawls the software versions to extract the models required as input by *UMLDiff*, a database back-end to store the extracted models and the *UMLDiff* change facts, several special-purpose analyses to infer more complex phenomena based on the *UMLDiff* change facts, and visualization modules for intuitively communicating, exploring and analyzing the discovered information.

The rest of the paper is structured as follows. Section 2 relates this work to previous research. Section 3 describes the metamodel assumed by *UMLDiff* as the underlying representation of its input logical-design models and the process by which these models are extracted from Java software. Section 4 discusses in detail the algorithm and its similarity heuristics. Section 5 reports on our experimental evaluation of *UMLDiff*. Finally, concluding remarks and some open questions for future research are outlined.

2 Related work

Lexical differencing tools, like *GNU diff*, are frequently used by developers, in concert with modification requests and bug reports, to reconstruct the changes between subsequent versions of a software module (Eick et al. 1992, 2001; Fischer et al. 2003; Lehman and Belady 1985). Figure 1 shows the Eclipse text-comparison results between two versions of a program.[1] In the after version, the duplicated method value(), which used to be implemented by classes PlainStatement and HTMLStatement, was pulled up into their new superclass, Statement. Unfortunately, the changes reported by the text-comparison tool are unintuitive: the first line was changed; five lines of code were added; a block of code was replaced by a single line. Since lexical differencing tools view software programs as text documents, they report changes at the lexical level, ignoring the high-level logical-design changes to which they correspond.

The Abstract Syntax Tree (AST) is one view of the structure of a software program. Figure 2 depicts a partial AST of the class PlainStatement in the before version. Yang (1991) developed a dynamic-programming tree-matching algorithm, for computing the similarity between ASTs. However, ASTs of realistic programs are big, which makes general tree-differencing algorithms impractical. Furthermore, they are often redundant. For example, Fig. 3 shows the VarDeclarationFragment subtree, corresponding to a variant of the second local variable declaration—String results = (this.headerString(aCustomer)). Although there is no actual semantic difference between the two variants, a tree-differencing algorithm, comparing it against the original VarDeclarationFragment subtree (the bottom-right corner of Fig. 2), would

[1] Excerpted from the version 27 and 28 of the extended refactoring example at http://www.cs.unc.edu/~stotts/COMP204/refactor. We adapt its version 23, 27 and 28 as the running example to illustrate *UMLDiff* algorithm in Sect. 4.

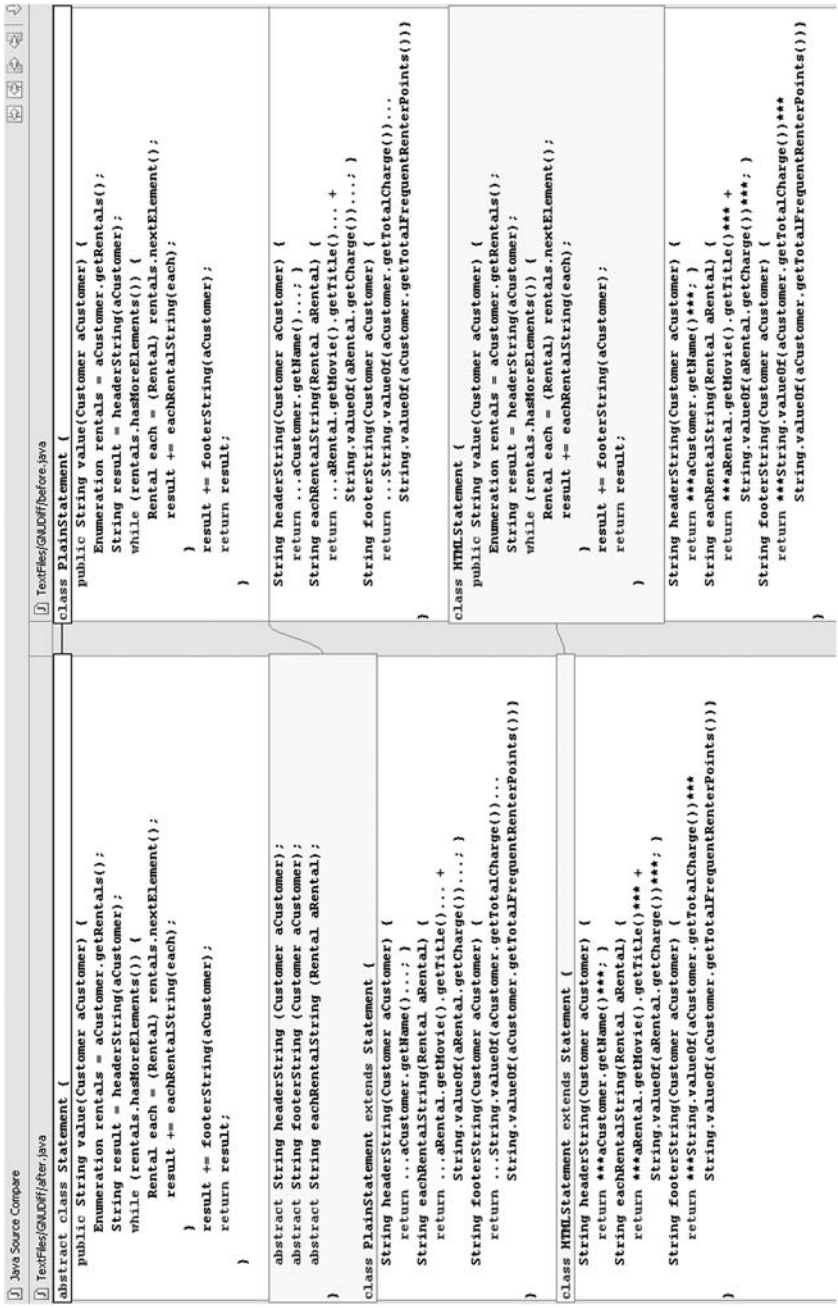

Fig. 1 Eclipse text compare

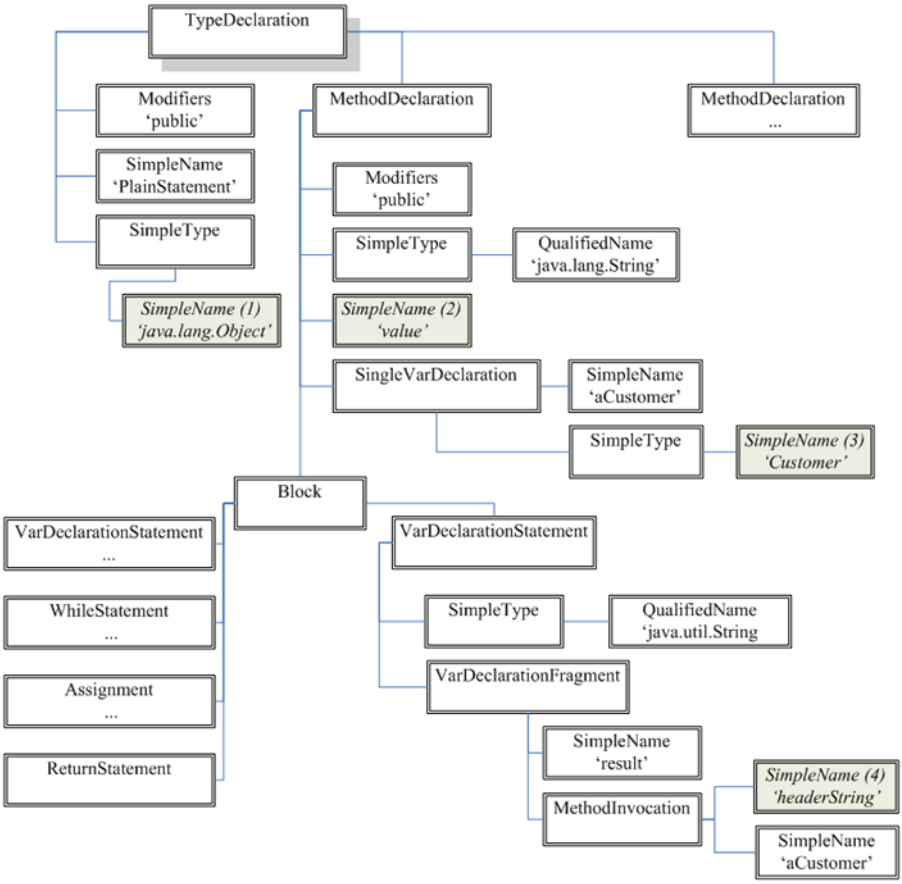

Fig. 2 The partial AST of class PlainStatement

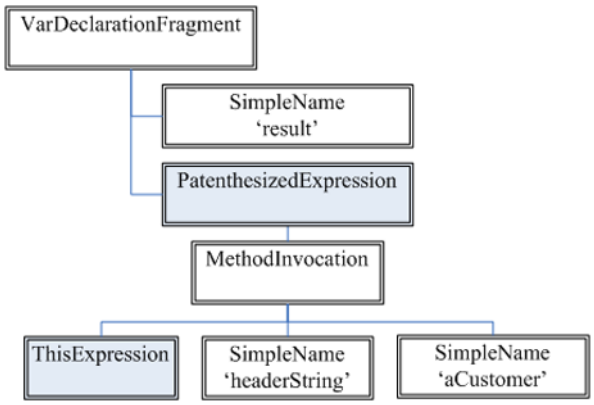

Fig. 3 The partial AST of changed local variable declaration

report the addition of node ParenthesizedExpression (which results in the MethodInvocation subtree being pushed one-level deeper) and the addition of node ThisExpression. AST is a low-level representation, designed for code compilation, optimization and transformation; interpreting AST changes into the higher-level logical changes requires substantial effort. For example, the value changes of four tree nodes of type SimpleName (gray highlight in Fig. 2) represent completely different logical changes: (1) the change of PlainStatement's superclass; (2) the renaming of the value() method; (3) the change of the parameter type of aCustomer; (4) the change in the outgoing usage of the method value().

There exist other differencing techniques that make use of other types of program representations. Semantic Diff (Jackson and Ladd 1994) operates on a representation of the local dependency graph and works at the intra-procedural level only, as opposed to the system as a whole. Horwitz developed a technique (Horwitz 1990) for detecting statement-level semantic and textual modifications, based on augmented control-flow graphs; this method is applicable to a simplified C-like programming language and is not suitable for complex object-oriented software systems.

Large-scale object-oriented software systems are better understood in terms of structural and behavioral models, such as UML class and sequence models. The UML modeling tools often store UML models in XMI (XML Metadata Interchange) format for data-interchange purposes. XML-differencing algorithms, applied to such easily available XMI representations, report changes of XML elements and attributes, ignoring the domain-specific semantics of the concepts represented by these elements. Figure 4 shows the partial XML comparison results (by Delta XML http://www.deltaxml.com) between the XMI representations (exported by ArgoUML http://argouml.tigris.org/) of two versions of the UML class model of the program listed in Fig. 1. The tool reports that two *UML:Operation* nodes (annotation [1]) were modified—their *name* attributes were changed—and that the *UML:Operation* with *name* attribute "eachRentalString" (annotation [2]) was removed, instead of recognizing that the value() method was moved to the new superclass Statement. Furthermore, a single logical change in the UML model may cause several XMI changes. For example, a generalization—the superclass of PlainStatement changed from Object to Statement—results in three changed XMI nodes (annotation [3]). Finally, similar XML element changes may represent completely different logical changes. For example, the attribute changes of the two *UML:Class* elements (annotation [4]) represent generalization and usage-dependency change respectively. Similar to AST comparison, an interpretation step is required to aggregate and abstract the change reports of XML comparison tools in terms of higher-level logical changes.

Several UML modeling tools come with their own UML-differencing methods (Ohst et al. 2003; http://www-128.ibm.com/developerworks/rational/library/05/712_comp/). They detect differences between subsequent versions of UML models, as long as these models are constructed and manipulated exclusively 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. Furthermore, the persistent identifiers imply only one-to-one mapping between model elements, even when many-to-one mappings are preferable. For example, reporting that both PlainStatement.value()

Fig. 4 XML-differencing XMI representation of UML models

and HTMLStatement.value() have been moved to Statement better reflects the intention of the change, which is to pull up commonalities from several subclasses into the superclass, than reporting that one has been moved and the other has been removed.

There has also been some work on comparative analysis of different snapshots of a software system for drawing inferences regarding its evolution. Demeyer et al. (2000) defined four heuristics based on the comparison of source-code metrics of two subsequent system snapshots, to identify refactoring activities of three general categories. Rysselberghe and Demeyer (2003) investigated the use of clone detection to identify moves and renamings. However, the source-code metrics do not report the details of what has or has not been changed. For example, the PlainStatement.value() and Statement.value() methods have the same NOM (Number of Messages sent in method body, see Lorenz and Kidd 1994) metrics, but PlainStatement.value() calls

directly the header/footer/eachRentalString() methods of PlainStatement while Statement.value() calls the abstract methods of Statement that are implemented by the corresponding PlainStatement methods. Ryder's group has also worked on comparative analysis of structural changes (Ryder and Tip 2001). 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. Apiwattanapong et al. (2004) 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.

All the above differencing techniques rely on various program representations that are designed for purposes other than understanding higher-level logical changes of software system. However, there has also been some research on analyzing the changes of software at the design level. Egyed (2001) 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 (2001) developed a probabilistic message-matching algorithm that detects the overlaps between messages that are likely to signify the invocation of operations and check whether the overlapping messages are inconsistent. 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 future decision making. Godfrey et al., in their BEAGLE system (Godfrey and Zou 2005; Tu and Godfrey 2002), use origin analysis to determine the "origin" of "new" files and to detect the merging and splitting of source-code entities. Origin analysis works at the file-structure level, corresponding to the physical model of the software rather than its logical model: it detects old functions as the "origin" of new ones based on a combination of clone detection and call-relation matching and requires an interactive step for detecting file merging and splitting.

In our own earlier work (Xing and Stroulia 2005b), we discussed an earlier version of *UMLDiff* that relied on name-similarity assessment using a particular metric, and structure-similarity assessment based on the immediate relations among model elements to recognize changes in the logical-design model of object-oriented software. In this paper, we discuss an enhanced version of the algorithm.

The new *UMLDiff* algorithm, which we discuss in this paper, uses any of three alternative lexical-similarity metrics, inspects comments (e.g., Javadoc) as an additional source of information for assessing elements' lexical-similarity in addition to name-similarity, examines transitive usage dependencies to assess structural similarity between two operations, performs multiple rounds of the renaming/move recognition steps and propagates the knowledge of the identified operation renamings along the inheritance hierarchy. These capabilities have improved the accuracy of the algorithm, whose run-time performance is still efficient and acceptable from a pragmatic point of view.

3 The metamodel

UMLDiff compares logical-design models of object-oriented software systems. The underlying metamodel is defined according to the semantics of the UML metamodel

(OMG 2003). We summarize the UML profile in terms of metaclasses and metarelations, of concern to *UMLDiff*, in Appendix 1.

Figure 5 diagrammatically depicts the UML model of the version 28 of our running example (see Sect. 4.1), in terms of instances of model-element metaclasses, relation metaclasses, and meta-compositions and meta-associations. The instances of model elements are denoted with the "name:metaclass" syntax: for example, "Statement:Class" represents an instance of the *class* metaclass, whose *name* attribute is Statement. The model elements may have other attributes, such as *visibility, isLeaf*, *isRoot*, *isAbstract*, *deprecated*, etc. For example, the *visibility* attribute of operation Customer.getAllCharge() is *public*. The *isAbstract* attribute of operation Statement.printFooter(Customer) is *true*. The model elements are linked to each other by instances of relation metaclasses, meta-compositions and meta-associations. For example, the model Version28 contains a default package (an instance of *ElementOwnership* meta-composition), which contains a class Customer (the other instance of *[namespace–ownedElement]* meta-composition), which declares four operations (four instances of *[owner–feature]* meta-composition). The operation HTMLStatement.printFooter(Customer) is associated with the class String as its return type (an instance of *[typedParameter–type]* meta-association). The operation Customer.htmlStatement() instantiates (an instance of $Usage_{«instantiate»}$) the object HTMLStatement. The class PlainStatement is a subclass of the class Statement (an instance of *Generalization*). The operation HTMLStatement.printEachRental(Rental) implements (an instance of $Abstraction_{«realize»}$) the abstract operation Statement.printEachRental(Rental).

UMLDiff requires as input representations of the system's logical design in terms of UML models, as shown in Fig. 5. These representations may be obtained through reverse-engineering the system source code. *UMLDiff* has so far been applied to comparing reverse-engineered UML logical-design models, given the source code of software systems implemented in Java. However, by adopting the semantics of the UML model as the metamodel underlying its input representations, it can readily be used to compare reverse-engineered models of software systems developed in (a mix of) other object-oriented programming languages, or to compare subsequent upfront design models to study their evolution, or to compare the upfront design model with the implemented model reverse-engineered from source code to validate code-to-design conformance.

3.1 UML model reverse engineering in JDEvAn

The *UMLDiff* algorithm has been implemented in the JDEvAn tool (http://www.cs.ualberta.ca/~xing/jdevan.html), which also implements a Java fact extractor based on the Eclipse Java DOM/AST model (http://www.eclipse.org). JDEvAn's Java fact extractor reverse engineers UML models in the form expected by *UMLDiff*, from source code. The mapping of the Java language constructs to UML metaclasses and metarelations is described in Appendix 2. Our current focus on Java is pragmatic; *UMLDiff* is not restricted to any specific object-oriented programming language, since its metamodel is essentially defined according to the UML semantics. Its design and implementation are extendible to software systems developed in other object-oriented programming languages, assuming appropriate fact extractors that are able to map pro-

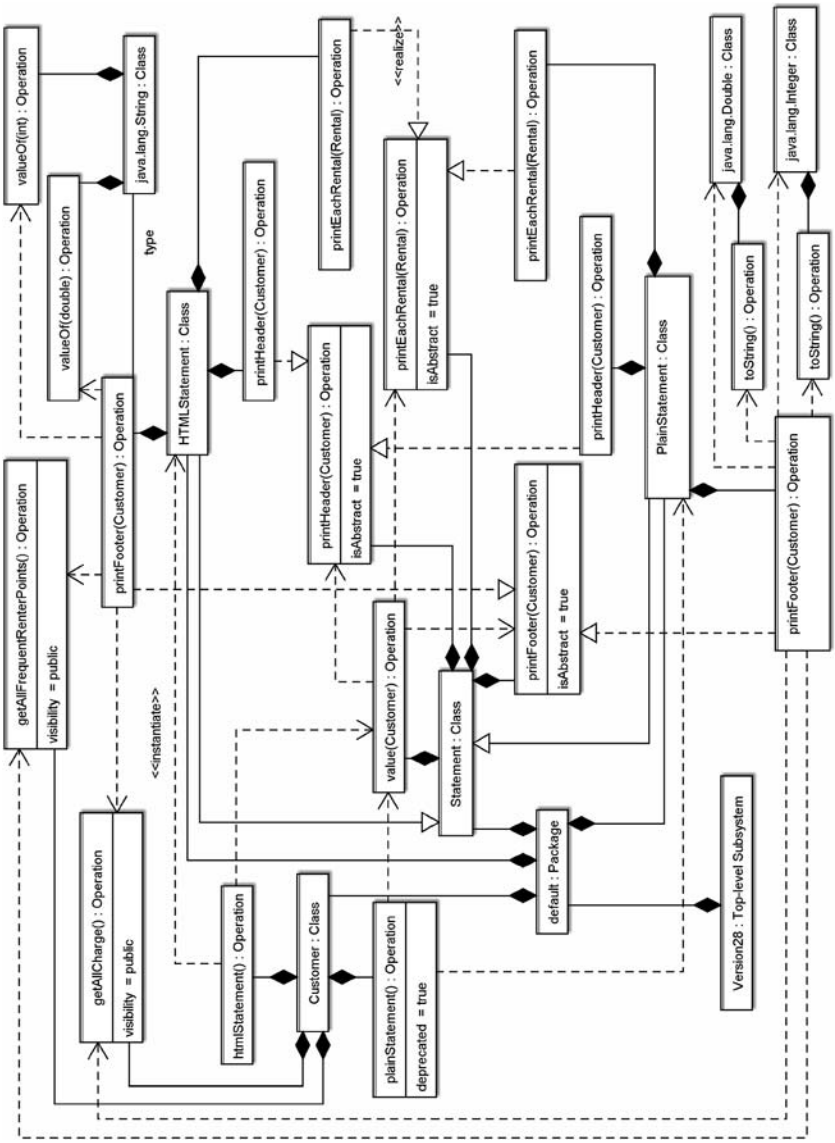

Fig. 5 An example of UML model that *UMLDiff* compares

gramming language constructs into the UML model elements and relations expected by *UMLDiff*.

Java software subsystem is not really a Java construct; it is a conceptual element. The top-level subsystem corresponds to the model of the analyzed system as a whole. Each *Operation* is associated with a *Method* element, which contains the body of the corresponding Java method, constructor, or class initializer. A *Method* element is not contained in the declaring class of its specification operation. The return type of a Java method is treated as a special *Parameter*, whose *name* and *kind* attributes are *return*. A field's initializer is modeled as the *initValue* attribute associated with its corresponding *Attribute*. Although Java requires exceptions to be subclasses of java.lang.Throwable, other programming languages, such as C++, allow exceptions to extend arbitrary classes. Therefore, to avoid restricting *UMLDiff* to the Java particulars, the fact extractor does not explicitly model exceptions; instead, at the end of fact extraction process, it marks as exceptions the classes that appear in $Usage_{«send»}$, *[context–raisedSignal]*, and *[reception–signal]* relations. The fact extractor does not model *Receptions* either, since in most modern object-oriented programming languages, operations are normally receptions that handle the signals, such as exceptions. Instead, at the end of the fact extraction process, it marks the operations that appear in *[reception–signal]* relations as receptions. Finally, the fact extractor ignores three Java specific modifiers, *volatile*, *native*, and *strictfp*, and assumes that the classes and interfaces that belong in Java libraries are contained in the top-level subsystem.

Each extracted model element is described in terms of its name, the type of its corresponding UML metaclass (as described in Tables 10 and 14), its corresponding visibility and attribute(s) (as described in Table 16), and its attached *UMLDiff*-specific tagged values (as described in Tables 13 and 17). The relations between model elements are described in tuples of the form (*relation*, e_1, e_2), where e_1 and e_2 are model elements and *relation* is a type of UML metarelation as described in Tables 11, 12 and 15 that applies between e_1 and e_2. 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 as the *count* tag, attached to the corresponding usage dependency.

The name of array types is in the form of "BasetypeQualifiedname.Dimension". The name of packages, 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 also assigns names to anonymous classes, "new supertype_identifier$number"; class initializers, "{class_identifier.$number}"; and field initializers, "{field_identifier = …}". The "number" is the ordinal number of the anonymous class or the class initializer within the enclosing Java class. Finally, a fully qualified prefix is added in front of the names of model elements that belong in Java libraries.

Anonymous classes are a special type of nested classes, with no explicitly declared identifiers. They are specified along with class creation expression within blocks and are then generated by the compiler when parsing expression as the nested classes of the class that declares the corresponding block. Thus, an anonymous class is modeled as a class with *name* "new supertype_identifier$number" whose *visibility* = *private*, *isAbstract* = *false*, and *isLeaf* = *true*, which is contained by the corresponding enclosing class. It is also associated with the corresponding method element of its

declaring operation. The $Usage_{«instantiate»}$ dependency between the anonymous class and its declaring operation is not modeled. Instead, it is mapped to the direct super type of the anonymous class. The fact extractor does not model Java local classes/interfaces, which are declared within methods, constructors, or class initializers, because "local types" is a Java specific feature, rarely used in practice.

The extracted UML logical-design models are stored in a PostgreSQL relational database, extended with Simon's transitive closure algorithm (Simon 1986) for computing transitive containment and inheritance relations, field read/write, method call, and class/interface usage relations. The relational database enables the *UMLDiff* implementation to work on large-scale software projects, such as Eclipse (http://www.eclipse.org). It also provides the flexibility to infer derivable information about model elements and their relations.

4 Comparing logical models of object-oriented software with *UMLDiff*

4.1 The running example

We will demonstrate how *UMLDiff* works with a small running example, adapted from the versions 23, 27 and 28 of the extended refactoring example at http://www.cs.unc.edu/~stotts/COMP204/refactor, which is available to download at http://www.cs.ualberta.ca/~xing/asej2007/runningexample.zip. When the system evolves from version 23 to 27, the nested class PlainStatement is extracted from the class Customer. The main responsibility of this class is to print out the customer's movie rental information in plain text format, which is originally performed by the operation Customer.statement() in version 23. In version 27, the operation Customer.statement() instantiates a PlainStatement object, to which it delegates this task. Similar changes are also made to $\text{Customer.htmlStatement()}_{23}$[2] and the newly introduced top-level class $\text{HTMLStatement}_{27}$ contained in default_{27} package. Furthermore, the $\text{Customer.statement()}_{23}$ is renamed to $\text{plainStatement()}_{27}$ in order to be consistent with htmlStatement() and to more clearly convey the intention of the method.

The main change between versions 27 and 28 is to pull up the operation value() from PlainStatement and HTMLStatement to their superclass, Statement. However, to demonstrate several *UMLDiff* key features, we intentionally complicated versions 27 and 28 by including the following changes:

- We renamed the operations getTotalChange()/getTotalFrequentRenterPoints() of the class Customer to getAllCharge()/getAllFrequentRenterPoints() respectively and changed their visibilities from package to public.
- We renamed the operations headerString()/eachRentalString()/footerString() of the classes Statement/PlainStatement/HTMLStatement to printHeader()/printEachRental()/printFooter() respectively.
- In version 27, the class PlainStatement is a nested class of the class Customer, while, in version 28, it is moved out from the class Customer and becomes a top-level class contained in the default package.

[2] Denotes the model element contained in a particular version.

- In version 27, the operation PlainStatement.footerString() uses String.value (double)/value(int) to convert the double and int values to String, while, in version 28, it changes to use Double.toString() and Integer.toString().
- In version 28, the operation Customer.plainStatement() is deprecated.

4.2 *UMLDiff* overview

UMLDiff is an UML-semantics-aware differencing algorithm. As per the adopted metamodel, the software system is modeled as a directed graph $G(V, E)$, where V, the vertex set, contains model elements and E, the edge set, contains relations among them. Given two versions, "before" and "after", of a UML logical-design model and their corresponding graphs $G_{\text{before}}(V_{\text{before}}, E_{\text{before}})$ and $G_{\text{after}}(V_{\text{after}}, E_{\text{after}})$, *UMLDiff* essentially maps the two model graphs by computing the intersection and margin sets between $(V_{\text{before}}, V_{\text{after}})$ and $(E_{\text{before}}, E_{\text{after}})$, in terms of $(V_{\text{before}} - V_{\text{after}})$ for the removed model elements, $(V_{\text{before}} \cap V_{\text{after}})$ for the mapped elements, $(V_{\text{after}} - V_{\text{before}})$ for the added model elements, $(E_{\text{before}} - E_{\text{after}})$ for the removed relations, $(E_{\text{before}} \cap E_{\text{after}})$ for the mapped relations, and $(E_{\text{after}} - E_{\text{before}})$ for the added relations.

UMLDiff first attempts to map the model element sets V_{before} and V_{after}. It relies on the composition relations to traverse in a breadth-first fashion[3] the vertices (model elements) of the directed graph of the UML model. The composition relations (instances of three meta-compositions—see Table 12) induce a strict containment-spanning tree on the directed graph. The UML semantics guarantees that all model elements can be visited by traversing the containment hierarchy starting from the top-level subsystem corresponding to the system version, and that the children of their containing parent are unique in terms of their names. The meta-composition defines four logical levels over types of model elements: subsystem (including the top-level subsystem) > package > (class, interface) > (attribute, operation). The model elements of type subsystem, package, class and interface may contain the nested same-type elements. Table 1 summarizes the containment hierarchy of the UML model elements. Table 5 shows the partial containment hierarchy of versions 23 and 27 of the model of our running example.

UMLDiff traverses the containment-spanning trees of the two compared models, descending from one logical level to the next, in both trees at the same time. It starts at the top-level subsystems that correspond to the two system versions and progresses down to subsystems, packages, classes and interfaces, and finally, attributes and operations. *UMLDiff* recognizes that a model element e_1 in the "before" version and an element e_2 of the same type in the "after" version are the "same", i.e., they correspond to the same conceptual model element, when (a) they have the same or similar name and comment (*lexical-similarity heuristic*), and (b) they have similar relations to other model elements, that have the same name and type or have already been established to be mapped (*structure-similarity heuristic*).

Name similarity is a "safe" indicator that e_1 and e_2 are the same entity: in our experience with several case studies (Schofield et al. 2006, Xing and Stroulia 2004a, 2005b, 2006b), very rarely is a model element removed and a new element with the

[3] In the rest of this paper, all references to "traversals" are implied to be "breadth-first traversals".

Table 1 The containment hierarchy of UML model elements

Type of model element	Type of the children
Top-level Subsystem	Subsystem and Package ProgrammingLanguageDataType Class and Interface whose isFromModel = false
Subsystem	Subsystem and Package
Package	Package, Class and Interface
Class	Class and Interface Attribute, Operation, Operation$_{«create»}$, and Operation$_{«initialize»}$
Interface	Class and Interface Operation
Attribute	N/A
Operation	Parameter

same name but different element type and different behavior is added to the system. *UMLDiff* recognizes same-name model elements of the same type first and uses them as initial "landmarks" to subsequently recognize renamed and moved elements. When a model element is renamed or moved, as is frequently the case with refactorings, its relations to other elements, such as the children elements it contains, the attributes it reads/writes, the operations it calls or is called by, etc., tend to remain much the same. Therefore, by comparing the relations of two same-type model elements renamings and moves can be inferred: if they share "enough" relations to known-to-be-same or same-name elements of the same type they are the "same", even though their names and/or their parent (containing) model elements are different. Whenever two model elements are identified as renamings or moves, this knowledge is added to the current landmarks' set and is used later on to further match not-yet-mapped elements. This process continues until it reaches the logical-leaf level of the spanning trees and all possible corresponding pairs of model elements have been identified.

Given two renaming or move candidates, *UMLDiff* computes their structural similarity as the cardinality of the intersection of their corresponding related-element sets: given the sets of elements that are related to the two compared candidates with a given type of relation, *UMLDiff* identifies the common subset of same-type same-name elements and elements that have already been established as mapped. Therefore, if all or most the model elements related to two candidates were also renamed and/or moved and cannot be established as "same", the *UMLDiff* structure-similarity heuristic would fail. If, on the other hand, a set of related elements were renamed or moved but enough model elements related to the affected set remained unchanged, it would be possible to recognize this systematic change.

UMLDiff applies two techniques, i.e., multiple-rounds-of-renaming-and-move-identification and propagating-operation-renamings-along-inheritance-hierarchy, to propagate the knowledge of established renamings and moves along their usage and inheritance relations (see Sects. 4.4.4 and 4.4.5). Finally, global renamings, such as

renamings to meet a new naming convention, for example, may be recovered, by enabling the user to specify a string transformation—introducing a prefix or appending a suffix, or replacing a certain substring—that should be applied to the names of the model elements of one of the compared versions, before the differencing process.

Once *UMLDiff* has completed mapping the sets of model elements, V_{before} and V_{after}, it proceeds to map the relation sets, E_{before} and E_{after}, by comparing the relations of all pairs of model elements $(v_{\text{before}}, v_{\text{after}})$, where $v_{\text{after}} = \text{null}$ if v_{before} is removed and $v_{\text{before}} = \text{null}$ if v_{after} is added. The relations from (to) a removed model element are all removed and the relations from (to) an added model element are all added. For a pair of mapped elements $(v_{\text{before}}, v_{\text{after}})$, they may have matched, newly added, and/or removed relations. Note that a removed (added) relation between two model elements does not indicate any of the elements it relates being removed (added).

Next, *UMLDiff* detects the redistribution of behavior among operations, in terms of usage dependency changes, and finally computes the changes to the attributes of all pairs of mapped model elements.

The *UMLDiff* differencing process is configured through a set of parameters.

1. The *LexicalSimilarityMetric* specifies which one of three lexical-similarity metrics (Char-LCS, Char-Pair, and Word-LCS) will be used by *UMLDiff*.
2. The *RenameThreshold* and *MoveThreshold* specify the minimum similarity values between two model elements in the two compared versions in order for them to be considered as the same conceptual element renamed or moved. *UMLDiff* allows multiple rounds (*MaxRenameRound* and *MaxMoveRound*) of renaming and/or move identification in order to recover as many renamed and moved entities as possible.
3. The similarity of the comments of the model elements (*ConsiderCommentSimilarity*) may also be taken into account when comparing two elements, if the compared elements have an initial overall similarity value above the *MinThreshold*; this prevents model elements with very low name- and structure-similarity from qualifying as renamings or moves just because of their similar comments.
4. The similarity of transitive usage dependencies (*ConsiderTransclosureUsageSimilarity*) between two compared operations may also be used to assess their structural similarity.
5. At the end of the *UMLDiff* differencing process, it can be instructed whether or not to compute the usage dependency changes for all model elements and analyze the redistribution of operation behavior.

The remainder of this section elaborates on the key features of *UMLDiff* algorithm.

4.3 Similarity metrics

UMLDiff relies on two heuristics—lexical and structure similarity—for recognizing the conceptually same model elements in the two compared versions of the system model, in spite of the fact that they may have been renamed and/or moved. In the following discussion, the term "matched elements" refers to same-name model elements of the same type, while "mapped elements" refers to matched, renamed, and moved elements.

4.3.1 Lexical similarity

The term "lexical similarity" refers to the similarity between the names and the comments associated with two compared model elements. *UMLDiff* integrates three metrics of string similarity: (a) the longest common character subsequence (Char-LCS); (b) the longest common token subsequence (Word-LCS); and (c) the common adjacent character pairs (Char-Pair). All these metrics are computationally inexpensive to calculate, given the usual length of the names and comments of model elements. They are also case insensitive, since it is common to misspell words with the wrong case or to modify them with just case changes. They are all applicable to name similarity, while only Char-LCS and Word-LCS may be applied to compute comment similarity.

The name similarity of operations is calculated as the product of their identifier similarity and their parameter-list similarity, which is computed as one type of structure similarity for operations. The name similarity of packages is computed based on their dot-removed names. The comment similarity between two model elements is only consulted when both elements have associated comments, the *UMLDiff* parameter *ConsiderCommentSimilarity* is true, and the initial overall similarity metric between these elements is greater than the *UMLDiff* parameter *MinThreshold*.

The longest common character subsequence (Char-LCS) algorithm (Wagner and Fischer 1974) is frequently used to compare strings. Word-LCS applies the same LCS algorithm, using words instead of characters as the basic constituents of the compared strings. The names of model elements are split into a sequence of words, using dots, dashes, underscores and case switching as delimiters. Comments are split into words using space as delimiters. The actual metric used for assessing LCS-similarity is shown in the following equation:

$$\text{Char/Word-LCS}(s_1, s_2) = 2 \cdot \text{length}(\text{LCS}(s_1, s_2))/(\text{length}(s_1) + \text{length}(s_2)),$$

where LCS() and length() is based on the type of token considered, i.e., characters or words.

LCS reflects the lexical similarity between two strings, but it is not very robust to changes of word order, which is common when renaming a design entity. To address this problem, we have defined the third lexical-similarity metric in terms of how many common adjacent character pairs are contained in the 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 Char-Pair similarity metric, which is a value between 0 and 1, is computed according to the following equation:

$$\text{Char-Pair}(s_1, s_2) = 2 \cdot |pairs(s_1) \cap pairs(s_2)|/(|pairs(s_1)| + |pairs(s_2)|).$$

4.3.2 Structure similarity

Table 2 lists the relations that *UMLDiff* examines to compute the structure similarity between two model elements of the same type. The top-level subsystems, corresponding to the two compared versions of a UML logical model, are always assumed

Table 2 The UML relations for computing structure similarity

Model element type	Type of relations
Top-level Subsystem	Always match
Subsystem	[namespace – ownedElement] Incoming and outgoing usage
Package	[namespace – ownedElement] Incoming and outgoing usage
Class and Interface	[namespace – ownedElement] and [owner – feature] Incoming and outgoing usage
Attribute	$\text{Usage}_{\text{«read»}}$ $\text{Usage}_{\text{«write»}}$ and inherent Attribute.initValue
Operation	[BehaviorFeature – parameter] and [typedParameter – type] Outgoing usage: $\text{Usage}_{\text{«read»}}$, $\text{Usage}_{\text{«write»}}$, $\text{Usage}_{\text{«call»}}$, and $\text{Usage}_{\text{«instantiate»}}$ Incoming usage: $\text{Usage}_{\text{«call»}}$

to match. The structure similarity of subsystems, packages, classes and interfaces is determined by the elements they contain, the elements they use, and the elements that use them. The structure similarity of attributes is determined by the operations that read and write them and their initialization expressions. The structure similarity of operations is determined by the parameters they declare, their outgoing usage dependencies (including the attributes they read and write, the operations they call, and the classes/interfaces they create), and their incoming usage dependencies (including the attributes (through their *initValue*) and the operations that call them).

The structure similarity of two compared elements is a measure of the overlap between the sets of elements to which the compared elements are related, according to a given relation type. The intersection of the two related-element sets contains the pairs of model elements that are related to the compared elements and have already been established to be mapped or have the same name and element type. This intersection set effectively incorporates knowledge of any "known landmarks" to which both compared model elements are related. Given two model elements of the same type, e_1 and e_2, let Set_{before} and Set_{after} be their related-element sets, the structure similarity between e_1 and e_2 according to a given group of relations is a normalized value (between 0 and 1) as computed in the following equation:

StructureSimilarity $=$ matchcount/(matchcount $+$ addcount $+$ removecount), where the matchcount, addcount, and removecount are the cardinalities of $[Set_{\text{before}} \cap Set_{\text{after}}]$, $[Set_{\text{after}} - Set_{\text{before}}]$, $[Set_{\text{before}} - Set_{\text{after}}]$ respectively.

For a usage dependency, its *count* tag, which indicates the number of times that it appears between the client and supplier elements, is used to compute its matchcount, addcount, and removecount.

The similarity of the parameters of two compared operations is based on the names and types of their parameters. The computation of parameter-list similarity is insensitive to the order of parameters. For non-return parameters, if none of the two operations is overloading, the matchcount for a pair of same-name parameters is 1. If any

Table 3 The related model-element sets of Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$

Type of relations		Customer.statement()	Customer.plainStatement()
Parameter		Return : String	Return : String
Outgoing usage	Read	Customer._rentals	null
	Write	null	null
	Call	Customer.getName()	PlainStatement.value()
		Customer.getTotalCharge()	
		Customer.getTotalFrequent. . .Points()	
		Rental.getMove()	
		Rental.getCharge()	
		Movie.getTitle()	
		String.valueOf(double) [2]	
		String.valueOf(int)	
		Vector.elements()	
		Enumeration.hasMoreElements()	
		Enumeration.nextElement()	
	Instantiate	null	PlainStatement
Incoming usage	Call	Vids.main(String[])	Vids.main(String[])

of the two compared operations is overloading, the types of the two same-name parameters is further examined, in order to distinguish the overloading methods from each other, which often declare the same name parameters but with different parameter types. In the case of overloading, if the same-name parameters have the mapped types, their matchcount is 1; otherwise, their matchcount is set at 0.5. For the return parameters, if their types map, the matchcount is 1. Otherwise, it is set at 0. If the type of the return parameter of both operations is void, the matchcount for the return parameter is set at 0.

The similarity of the *initValue* of two compared attributes is computed in the same way as the outgoing usage similarity between two operations. The initValue-similarity value is added to the overall matchcount of the $Usage_{«write»}$ similarity between two attributes.

Take the operations Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$ as an example. Let us assume that *UMLDiff* has identified the matched model elements and is in the process of identifying renamings. It collects [Customer.statement()$_{23}$, Customer.plainStatement()$_{27}$] as a pair of renaming candidates (see Sect. 4.4.2). Table 3 shows the two related-element sets of Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$. Note that all the incoming and outgoing usages of these two operations, except for Customer.statement()$_{23}$ calling String.valueOf(double)$_{23}$ twice, happen to be one. We omit the *count* tag attached to such usage dependencies. If a usage dependency appears more than once, it is indicated at the end of the usage dependency, such as String.valueOf(double) [2]. In the case of comparing Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$, the similarity of their parameters is one, their incoming usage similarity is also one, and their outgoing usage similarity is zero.

Table 4 The transitive outgoing usage of Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$

Type of relations		Customer.statement()	Customer.plainStatement()
Outgoing usage	Read	Customer._rentals	Customer._rentals
		Customer._name	Customer._name
		Movie._title	Movie._title
		Movie._daysRented	Movie._daysRented
		Price._price	Price._price
		Rental._movie	Rental._movie
	Write	null	null
	Call	... (omit 17 matched operations)	PlainStatement.value()
			PlainStatement.headerString()
			PlainStatement.eachRentalString()
			PlainStatement.footString()
			Customer.getRentals()
			... (omit 17 matched operations)
	Instantiate	null	PlainStatement

When computing incoming and outgoing usage similarity between two operations, if the two compared operations are related to some other model elements but the intersection of the two related-element sets is empty, as in the case of the outgoing usage of Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$, *UMLDiff* proceeds to compute the transitive usage similarity between the two compared operations, if its *ConsiderTransclosureUsageSimilarity* parameter is set to true. The transitive usage similarity takes into account the model elements related through the transitive-closure of the given relation, in addition to the directly related elements.

Table 4 shows the transitive outgoing usage of Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$. The transitive usage similarity is still computed as per the above structure-similarity equation, but without considering the *count* tag. The matchcount, addcount, and removecount for the transitive outgoing usage similarity between Customer.statement()$_{23}$ and Customer.plainStatement()$_{27}$ are 23, 6, and 0 respectively. Thus, the transitive outgoing usage similarity is $23/(23+6+0) = 0.79$.

Determining the similarity when both related model element sets are empty is challenging, when, for example, two operations are not called by any other operations. In such cases, setting the structure similarity to be by default 0 or 1 is not desirable: without any explicit evidence of similarity, assuming that the structure is completely the same or completely different may skew the subsequent result. Therefore, in such cases, *UMLDiff* uses the name similarity with an increasing exponent. The effect is dampened as more empty sets are encountered. For example, when computing the structure similarity of two operations in the order of their parameter-list, outgoing usage and incoming usage similarities, if the two compared operations declare no parameters, have return type void, and have no outgoing and incoming usage dependencies, *UMLDiff* returns *name-similarity*^{1} for comparing parameter-list similarity, *name-similarity*^{2} for outgoing usage similarity, *and name-similarity*^{3} for incoming usage similarity.

4.3.3 Overall similarity assessment

Given two model elements e_1 and e_2 of the same type, their overall similarity metric, used for determining potentially renamed and moved model elements, is computed according to the following equation:

SimilarityMetric = (lexical-similarity + $\Sigma_N$structure-similarity)/(lexical-similarity + N), where lexical-similarity = name-similarity + comment-similarity, and N is the number of different types of structure similarities computed for a given type of model elements, as defined in Table 2.

The value of $\Sigma_N$structure-similarity is adjusted in the following cases. When comparing two operations, if anyone of them is overloaded, $\Sigma_N$structure-similarity is multiplied by the parameter-list similarity of the compared operations in order to distinguish the overloading operations from each other, which often have similar usage dependencies but with different parameters. When determining the potential moves of attributes and operations, if the declaring classes/interfaces of the compared attributes/operations are not related through inheritance, containment, or usage relations, the value of $\Sigma_N$structure-similarity is multiplied by the overall similarity metrics of the classes in which the compared attributes/operations are declared and then divided by the product of the numbers of all the not-yet-mapped model elements with the same name (same identifier for operation) and type as the two compared elements. This is designed to improve the low precision when identifying attribute and operation moves.

UMLDiff uses two user-defined thresholds (*RenameThreshold* and *MoveThreshold*): two model elements are considered as the "same" element renamed or moved when their overall similarity metric is above the corresponding threshold. If, for a given element in the "before" version, there are several potential mappings 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.

4.4 Mapping model elements

Given two versions, "before" and "after", of a UML logical model and their corresponding directed graphs $G_{\text{before}}(V_{\text{before}}, E_{\text{before}})$ and $G_{\text{after}}(V_{\text{after}}, E_{\text{after}})$, *UMLDiff* starts with the original vertex sets V_{before} and V_{after} that contain all the model elements and the initially empty mapped element set. After it finishes mapping the model elements, the mapped element set contains all the identified matched, renamed, and moved model elements, and the V_{before} contains all the elements that have been removed and the V_{after} contains all the elements that have been added when the system model evolves.

Table 5 presents the partial model-element sets of versions 23 and 27 of our running example, V_{23} and V_{27}, organized according to their containment hierarchy. "null" entries indicate that there is no model element of a given type contained in a particular model element. In the remainder of this subsection, we present how *UMLDiff* identifies same-name (i.e., matched), renamed, moved model elements using the running example presented in Sect. 4.1.

Table 5 The partial model-element sets V_{23} and V_{27}

Version23: Top-level subsystem			Version27: Top-level subsystem		
Elements	Their children		Elements	Their children	
Version23	Subsystem	null	Version27	Subsystem	null
	Package	default		Package	default
	PLDatatype	String[]		PLDatatype	String[]
Default	Package	null	Default	Package	null
	Class	Vids		Class	Vids
		Rental			Rental
		Movie			Movie
		Customer			Customer
					HTMLStatement
	Interface	null		Interface	null
Vids	Class	null	Vids	Class	null
	Interface	null		Interface	null
	Operation	main(String[])		Operation	main(String[])
	Operation$_{«c»}$	null		Operation$_{«c»}$	null
	Attribute	null		Attribute	null
Customer	Class	null	Customer	Class	PlainStatement
	Interface	null		Interface	null
	Operation	getName()		Operation	getName()
		getTotalChange()			getTotalChange()
		getTotal...Points()			getTotal...Points()
		htmlStatement()			htmlStatement()
		statement()			*plainStatement()*
					getRentals()
	Operation$_{«c»}$	Customer(String)		Operation$_{«c»}$	Customer(String)
	Attribute	_name		Attribute	_name
		_rentals			_rentals

4.4.1 "Matched" elements

UMLDiff assumes that enough model elements remain "matched" between two compared versions of the system, which serve as the "initial landmarks" set for recognizing renamed and moved elements. The term "matched" refers to two corresponding model elements, of the same UML type, contained in a pair of mapped elements, which have the same names, although their children, attributes, and relations with other elements may be different.

The very first step of *UMLDiff* is to identify as many matched model elements as possible. It starts at the top-level subsystems of the two compared versions of the system model, which are always assumed to match. The pair of the matched top-level subsystems is added into the mapped element set as the first pair of mapped elements. *UMLDiff* then progresses along the containment hierarchy of the models, moving

from one logical level to the next in both trees at the same time, from subsystems, to packages, classes and interfaces, and finally attributes and operations. Given a pair of mapped model elements of the current logical level in the mapped element set, *UMLDiff* identifies all their children of the same type with same names, adds them to the mapped element set as new pairs of matched elements, and removes them from the set V_{before} and V_{after} respectively. The pairs of matched children may be at the current logical level or one level below. The process continues until there are no more unprocessed pairs of matched elements of the current logical level in the mapped element set and *UMLDiff* then progresses down to the next logical level.

Consider, for example, our running example. The matched model elements are of regular font and left justified in Table 5. In this example, given the matched top-level subsystems, *UMLDiff* adds their contained same-name default packages into the mapped element set. Next, it maps the four same-name classes contained in the default package. Given the matched class Customer, it maps the same-name attributes, constructor, and operations it declares. Note that there is no mapped nested model element in this simple running example. *UMLDiff* proceeds directly from the subsystem, to the package level, to the class/interface level, and finally to the attribute/operation level.

UMLDiff may not recover all the pairs of the matched model elements in this round: same-name, same-type model elements contained in renamed and moved parent elements are also considered as matched, but, at this stage, the renamed and moved model elements have not yet been recovered. As the pairs of renamed or moved elements are added to the mapped element set, *UMLDiff* attempts again to identify the same-name, i.e. matched, children they contain recursively, starting at the given pair of renamed/moved model elements. The only difference is that, instead of traversing the whole containment hierarchy from the top-level subsystem, it traverses the subtree of the containment hierarchy rooted at the given pair of renamed or moved model elements.

4.4.2 Renamed[4] *elements*

After *UMLDiff* has completed its recognition of matched model elements, it proceeds to recover the renamed model elements. *UMLDiff* only considers potential renamings within the context of two mapped elements, such as the renaming of an operation within a mapped class. Identifying renamings between two arbitrary elements of the same type, such as the renaming of an operation that was moved from one class to another and then had its identifier renamed, is computationally expensive, since it requires the comparison of all the pairs of not-yet-mapped model elements of the same type. Again, *UMLDiff* starts at the matched top-level subsystems of the two compared versions of the system model and it traverses all the mapped model elements along the containment-spanning trees of the compared model graphs to identify pairs of renamed elements, moving from one logical level to the next when it has completed traversing all the model elements of the current logical level in both spanning trees.

[4]The renamings of operations include the changes to their identifiers and/or parameter lists. Furthermore, *UMLDiff* does not consider parameter renamings.

Note that *UMLDiff* may not recover all the pairs of the renamed model elements in its first round of recognizing the renamed elements due to two reasons. It may miss the pairs of renamed elements because their related elements have undergone renamings and/or moves as well. Some of these misses may be recovered in the following rounds of renaming identification (see Sect. 4.4.4). Furthermore, renamed model elements may be contained within moved elements or other not-yet-identified renamed elements. Once the pairs of such elements have been added to the mapped element set, *UMLDiff* attempts again to identify the pairs of renamed model elements they contain recursively starting at the given pair of moved or newly-identified renamed model elements.

Given a pair of mapped model elements of the current logical level in the mapped element set, *UMLDiff* first collects all their not-yet-mapped children of the same type and formulates sets of candidate renaming pairs. Suppose there are N not-yet-mapped elements of a particular type contained in the "before" version of the mapped elements and M in the "after" version: *UMLDiff* generates N sets of renaming candidate pairs, each of which contains M pairs. It then identifies the renamed model elements based on their lexical and structure similarities, adds the newly identified pairs of renamed elements to the mapped element set, and removes them from the V_{before} and V_{after} sets. Adding a pair of renamed elements to the mapped element set triggers *UMLDiff* to recursively recognize the matched descendants they contain. The pairs of newly identified renamed and matched children may be at the current logical level or one level below. The process continues until there are no more unprocessed pairs of mapped elements of the current logical level in the mapped element set and *UMLDiff* then progresses down to the next logical level.

For example, when comparing the version 23 and 27 of our running example, the matched top-level subsystems contain only a pair of matched default packages. The matched default packages contain four matched classes and one not-yet-mapped class, $\text{HTMLStatement}_{27}$. Thus, at this point, there are no potential renaming candidate pairs. However, when *UMLDiff* reaches the matched class Customer, it collects the following not-yet-mapped children: operation $\text{Customer.statement()}_{23}$, and operations $\text{Customer.plainStatement()}_{27}$ and $\text{Customer.getRentals()}_{27}$ and nested class $\text{PlainStatement}_{27}$. Since there are no not-yet-mapped nested classes contained in the class Customer_{23}, there is no need yet for identifying renamings of the Customer's nested classes. However, *UMLDiff* formulates one set of operation-renaming candidate pairs (italic font and right justified in Tables 5 and 7), which contains two pairs of renaming candidates: [statement()_{23}, $\text{plainStatement()}_{27}$] and [$\text{statement()}_{23}$, getRentals()_{27}].

The overall similarity of each of these pairs is computed according to the equations shown in Sect. 4.3.3, based on their lexical and structure similarities. When comparing $\text{Customer.statement()}_{23}$ and $\text{Customer.plainStatement()}_{27}$, *UMLDiff* computes three types of structure similarity between them, i.e. parameter list, outgoing usage, and incoming usage, which are 1, 0.793, 1 respectively. Their identifier similarity using Word-LCS[5] is 0.5. Thus, their overall similarity metric is 0.941. The overall similarity metric between [statement()_{23}, getRentals()_{27}] is similarly computed to be 0.139. Thus, $\text{plainStatement()}_{27}$ is much more similar to statement()_{23}

[5] Word-LCS is used for all the lexical-similarity computation in this paper.

Table 6 The sets of renaming candidate pairs

HTMLStatement			
[header?()[a] − ?Header()]	[footer?() − ?Header()]	[each?() − ?Header()]	[value() − ?Header()]
[header?() − ?Footer()]	[footer?() − ?Footer()]	[each?() − ?Footer()]	[value() − ?Footer()]
[header?() − ?Each...()]	[footer?() − ?Each...()]	[each?() − ?Each...()]	[value() − ?Each...()]
Customer			
[getTotalCharge() − getAllCharge()]		[getTotalFrequentRenterPoints() − getAllCharge()]	
[getTotalCharge() − getAllFrequentRenterPoints()]		[getTotalFrequentRen...() − getAllFrequentRen...()]	

[a]Replace the suffix "String" and the prefix "print" with "?" for the operations of HTMLStatement to fit them in table.

than getRentals()$_{27}$. Assuming that the *RenameThreshold* is less than 0.941, the pair [statement()$_{23}$, plainStatement()$_{27}$] is recognized as an instance of operation renaming.

Let us now look at the versions 27 and 28 of our running example. They involve much more complex changes, including many renamings and moves. Let us first examine renamings. Similar to the comparison of versions 23 and 27, *UMLDiff* first identifies all the matched model elements starting from the top-level subsystems. Then it proceeds to collect potential renaming candidates and to formulate the sets of candidate renaming pairs, such as those shown in Table 6 for the matched class HTMLStatement and Customer.

Note that HTMLStatement.value()$_{27}$ is collected as a renaming candidate at this stage of *UMLDiff* process. It is compared against three HTMLStatement.printXXX()$_{28}$ operations but it is not found similar to anyone of them; therefore, it is finally collected as one of the potential move candidates (bold font and left justified in Table 7). Furthermore, the abstract operations, such as those of the class Statement, are not collected as renaming or move candidates: since they have no outgoing usage, the identification of their renamings or moves tends to be error-prone. *UMLDiff* ignores them in its renaming and move identification process. However, the renamings of the abstract operations may be recovered by propagating the knowledge of the identified renamings of their implementation operations along the inheritance hierarchy as discussed in Sect. 4.4.5. Finally, at this stage, the not-yet-mapped operations of the PlainStatement class are not collected as renaming candidates, since the move of the class PlainStatement has not yet been identified. However, they will be processed when the move of PlainStatement is identified and added to the mapped element set.

UMLDiff computes the overall similarity metrics of all the pairs of renaming candidates contained in a given pair of mapped model elements and selects the pair with the highest similarity metric (above the *RenameThreshold*) to be added to the mapped element set as a renaming. It then removes from the candidate sets all other pairs that contain the elements of the selected pair. This process continues until there is no pair left.

For example, if the *UMLDiff RenameThreshold* parameter is 0.3, then all 12 pairs of operation-renaming candidates of the matched HTMLStatement class have sufficiently high similarity metric to be qualified for further examination. The pair

Table 7 The initial not-yet-mapped model elements after the match/renaming recognition steps

Version27: Top-level subsystem			Version28: Top-level subsystem		
Elements	Their children		Elements	Their children	
Version27	Subsystem	null	Version28	Subsystem	null
	Package	null		Package	null
	PLDatatype	null		PLDatatype	null
default	Package	null	default	Package	null
	Class	null		Class	**PlainStatement**
	Interface	null		Interface	null
Customer	Class	**PlainStatement**	Customer	Class	null
	Interface	null		Interface	null
	Operation	null		Operation	null
	Operation$_{«c»}$	null		Operation$_{«c»}$	null
	Attribute	null		Attribute	null
PlainStatement	Class	null	PlainStatement	Class	null
	Interface	null		Interface	null
	Operation	*headerString()*		Operation	*printHeader()*
		eachRentalString()			*printEachRental()*
		footerString()			*printFooter()*
		value()			
	Operation$_{«c»}$	null		Operation$_{«c»}$	null
	Attribute	null		Attribute	null
HTMLStatement	Class	null	HTMLStatement	Class	null
	Interface	null		Interface	null
	Operation	**value()**		Operation	null
	Operation$_{«c»}$	null		Operation$_{«c»}$	null
	Attribute	null		Attribute	null
			Statement	Class	null
				Interface	null
				Operation	**value()**
				Operation$_{«c»}$	null
				Attribute	null

[eachRentalString()$_{23}$, printEachRental()$_{27}$] ranks highest and is selected as a pair of renamed elements; then all other pairs that contain either eachRentalString()$_{23}$ or printEachRental()$_{27}$ are removed from the list. *UMLDiff* then selects the pair with the highest similarity-metric value in the current list until the pair list is empty.

4.4.3 Moved elements

Finally, *UMLDiff* proceeds to examine those model elements 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 first starts at the top-level subsystem of V_{before} and traverses all the not-yet-mapped model elements along the containment hierarchy of the model, moving from one logical level to the next, when there are no more unprocessed elements of the current logical level. Thus, *UMLDiff* first identifies all the potential subsystem moves, and then progresses down to package moves, class and interface moves, and finally attribute and operation moves.

When it encounters a not-yet-mapped model element e_{before}, *UMLDiff* collects all the not-yet-mapped same-type and same-name (same-identifier for operation) model elements in V_{after} and forms a set of move candidate pairs, if such elements exist. It then computes the overall similarity metrics for all these candidate pairs, selects the pair with the highest similarity metric (above the *MoveThreshold*), and adds it to the mapped element set as a pair of moved model elements. Adding a pair of moved elements to the mapped element set triggers *UMLDiff* to recursively recognize their matched and renamed descendants. This process continues until all the not-yet-mapped model elements of the current logical level have been processed; then *UMLDiff* proceeds to identify the potential moves at one logical level below.

Note that for operations, *UMLDiff* uses their identifiers instead of their full signatures to collect move candidates, which enables the identification of changes involving operation moves with simultaneous parameter-list modifications. Furthermore, the set of not-yet-mapped elements may change as the process goes on, because the descendants of the newly identified moved elements might be identified as matches and renamings when the pairs of moved elements are added to the mapped element set, as discussed for the moved PlainStatement class below. Finally, the identified moved elements are only removed from V_{before} and V_{after} after the whole move recognition step is complete. After all the not-yet-mapped elements in V_{before} have been processed, *UMLDiff* starts at the top-level subsystem of V_{after} and performs the same tasks as above. This step, together with not-immediately-remove-moved-elements-from-element-sets, enables *UMLDiff* to identify many-to-one and one-to-many mapping between moved elements.

Table 7 lists all the remaining not-yet-mapped model elements that are still in V_{27} and V_{28} of our running example, after *UMLDiff* has completed the match and renaming recognition steps. Note that the three abstract operations of the Statement class are not listed in Table 7, since *UMLDiff* does not consider the moves of the abstract operations, which tends to be error-prone due to the lack of outgoing usage dependencies from them. The top-level subsystem, Version27, and its default_{27} package do not contain any not-yet-mapped subsystems or packages. Thus, *UMLDiff* proceeds to the class/interface logical level. When traversing the classes and interfaces, it encounters a not-yet-mapped class $\text{Customer.PlainStatement}_{27}$ (bold font and left justified in Table 7 for move candidates). *UMLDiff* then searches the remaining not-yet-mapped classes contained in V_{28} and retrieves all the classes with the same name. It finds the not-yet-mapped class $\text{PlainStatement}_{28}$ in the default_{28} package. Given the moving candidates [$\text{Customer.PlainStatement}_{27}$, $\text{default.PlainStatement}_{28}$], *UMLDiff* computes their similarity metric to be 0.6. If the *MoveThreshold* is lower than 0.6, the pair of [$\text{Customer.PlainStatement}_{27}$, $\text{default.PlainStatement}_{28}$] is added to the mapped element set as a moved class.

Adding the moved class [$\text{Customer.PlainStatement}_{27}$, $\text{default.PlainStatement}_{28}$] to the mapped element set triggers *UMLDiff* to recognize the matched and renamed descendants they contain. The class PlainStatement has no matched children, but it has

four and three not-yet-mapped operations in version 27 and 28 respectively. *UMLDiff* collects them as renaming candidates and identifies three operation renamings (italic font and right justified in Table 7) that are added to the mapped element set.

After processing the class PlainStatement, there aren't any not-yet-mapped classes or interfaces and *UMLDiff* proceeds to the attribute/operation level. It encounters the not-yet-mapped operation HTMLStatement.value()$_{27}$ and retrieves from Version28 the not-yet-mapped operation Statement.value()$_{28}$. *UMLDiff* computes the overall similarity of the candidate move pair [HTMLStatement.value()$_{27}$, Statement.value()$_{28}$] to be 0.71. The [HTMLStatement.value()$_{27}$, Statement.value()$_{28}$] pair is added to the mapped element set as a moved operation, assuming that the *MoveThreshold* is lower than 0.71. After that, *UMLDiff* encounters the not-yet-mapped operation PlainStatement.value()$_{27}$. Similarly to HTMLStatement.value()$_{27}$, the pair [PlainStatement.value()$_{27}$, Statement.value()$_{28}$] is also identified as an operation move. Note that the operations headerString(), footerString(), and eachRentalString() of PlainStatement are not encountered as not-yet-mapped elements: they are identified as operation renamings when the move of PlainStatement class is recognized, which results in them being removed from the initial remaining not-yet-mapped model element sets. After processing all the not-yet-mapped elements in V_{27}, *UMLDiff* starts over at the top-level subsystem of V_{28}, which contributes no more moves in this running example.

When examining attribute/operation move candidates, if their declaring classes (interfaces) are related through inheritance, containment, or usage relations, their non-adjusted structure similarities are used in the computation of their overall similarity. Otherwise, *UMLDiff* computes the overall similarity metric of their declaring classes (parent-similarity) and calculates the product (amount-potential-moves) of the numbers of the not-yet-mapped, same-type, same-name model elements as the two compared elements in the two compared versions. In this case, the structure similarity of the two compared attributes/operations is adjusted as Σ_N structure-similarity $\cdot$ parent-similarity/amount-potential-moves. Intuitively, if the source and target classes of the moved attribute/operation have no special relation, *UMLDiff* takes into account the contexts from and to which the attributes/operations move: they must be similar enough in order for the moves of attributes/operations to make sense. Furthermore, the more the potential moves of the same kind are, the less likely it is that any of them will be recognized as a valid move.

Take the [HTMLStatement.value()$_{27}$, Statement.value()$_{28}$] as an example: since Statement is the superclass of HTMLStatement, the original structure similarity 1.83 is used to compute the overall similarity metric, which is 0.71. Let us assume that there is no special relation between HTMLStatement and Statement. The overall similarity metric (parent-similarity) of [HTMLStatement$_{27}$, Statement$_{28}$] is 0.7 and there are two potential moves of the value() operation. Thus, the structure similarity of [HTMLStatement.value(), Statement.value()] that is used to compute the overall similarity metric becomes $1.83 \cdot 0.7/2 = 1.3$, which brings the overall similarity metric down to 0.58.

This technique is designed to improve the precision of moves of the original *UMLDiff* (Xing and Stroulia 2005b), which tend to have low precision of attribute/operation moves. For example, in an interactive system, many classes implement the ActionListener interface and its actionPerformed() operation; in general,

these implementations handle different user actions and are used in different contexts. However, when some actionPerformed() method disappears (usually because its class is removed or has stopped implementing the ActionListener interface) and new ones appear between two compared versions of a model, the original *UMLDiff* algorithm tends to report those as pairs of moves, which usually does not make sense. The current *UMLDiff* algorithm integrates the above technique to filter out such moves.

4.4.4 *Propagating knowledge of identified renamings and moves along usage dependency*

UMLDiff computes the structure similarity of two compared model elements in terms of the intersection of their two related-element sets. It is sensitive to the order that a set of renamed and/or moved model elements are examined, which may result in some renamings and moves being missed during a particular round of renaming/move identification. On the other hand, the more renamings and moves *UMLDiff* recovers, the larger the current "landmarks" set (i.e., the mapped element set) becomes, and the more likely it becomes that *UMLDiff* may recover further related renamings and moves.

Let us look at versions 27 and 28 of our running example. The operations Customer.getTotalCharge()$_{27}$ and Customer.getTotalFrequentRenterPoints()$_{27}$ and their caller operations HTMLStatement.footerString()$_{27}$ and PlainStatement.footerString()$_{27}$ are renamed to Customer.getAllCharge()$_{28}$, Customer.getAllFrequentRenterPoints()$_{28}$, HTMLStatement.printFooter()$_{28}$ and PlainStatement.printFooter()$_{28}$ respectively.

First, the renaming candidate pair [PlainStatement.footerString()$_{27}$, PlainStatement.printFooter()$_{28}$] is examined after the PlainStatement class has been recognized as moved. Furthermore, the renaming candidates [Customer.getTotalCharge()$_{27}$, Customer.getAllCharge()$_{28}$] may be compared before [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$]; even if the order is reverse, the renaming of [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$] may not be recovered if the *RenameThreshold* is greater than 0.5 (see below). Therefore, at the time of determining the mapping between [getTotalCharge()$_{27}$, getAllCharge()$_{28}$], their incoming usage relations may be substantially different and their incoming usage similarity may be 0. However, the operations [getTotalCharge()$_{27}$, getAllCharge()$_{28}$] declare the same parameters and they use the same sets of other model elements; their parameter-list similarity and outgoing usage similarity are 1, which brings their overall similarity to 0.714, which is sufficiently high. Thus, even without the knowledge of the renaming [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$] and [PlainStatement.footerString()$_{27}$, PlainStatement.printFooter()$_{28}$], getTotalCharge()$_{27}$ and getAllCharge()$_{28}$ may still be recovered as a renamed pair, at a fairly high *RenameThreshold*. The case of the renaming candidates [getTotalFrequentRenterPoints()$_{27}$, getAllFrequentRenterPoints()$_{28}$] is similar.

On the other hand, the outgoing usage similarity of [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$] is 0.33, when the renaming pairs [getTotalCharge()$_{27}$, getAllCharge()$_{28}$] and [getTotalFrequentRenterPoints()$_{27}$, getAllFrequentRenterPoints()$_{28}$] have not yet been recovered, but it increases to 1 if these pairs have

already been established as renamings before *UMLDiff* considers the renaming candidate pair [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$]. The corresponding overall similarity metric of the pair [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$] increases from 0.5 to 0.7, which may push the pair above the *RenameThreshold*.

It is computationally expensive to keep track of all related not-yet-mapped model elements. Furthermore, as shown in the example, it is not necessary to update the similarity metric of two not-yet-mapped model elements as each of its related renamings and/or moves is recovered. For example, we only need to re-compute the similarity metric of the renaming candidate pair [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$] once after both the renamings of [getTotalCharge()$_{27}$, getAllCharge()$_{28}$] and [getTotalFrequentRenterPoints()$_{27}$, getAllFrequentRenterPoints()$_{28}$] are identified. Therefore, at the end of each round of renaming and move identification, *UMLDiff* collects the pairs of not-yet-mapped renaming and move candidates that are related, through usage dependencies, to the newly identified renamed and moved model elements in the last round and updates their similarity metrics to see if they may be qualified this time.

UMLDiff may be configured to perform up to *MaxRenameRound* and *MaxMoveRound* of renaming and move recognition or to continue until there is no more affected potential renaming and move candidates that are related to the new instances of renamings and moves identified in the last round. Allowing multiple rounds of renaming and move identification relieves the impact of the order of the model elements being processed by *UMLDiff* on its final mapping results.

4.4.5 Propagating identified operation renamings along inheritance hierarchy

When the move of PlainStatement class is identified, *UMLDiff* attempts to recover its operation renamings (see Sect. 4.4.3). When the pair [PlainStatement.footerString()$_{27}$, PlainStatement.printFooter()$_{28}$] is examined, the related operation renamings [getTotalCharge()$_{27}$, getAllCharge()$_{28}$] and [getTotalFrequentRenterPoints()$_{27}$, getAllFrequentRenterPoints()$_{28}$] have already been identified. However, the overall similarity of the pair [PlainStatement.footerString()$_{27}$, PlainStatement.printFooter()$_{28}$] is 0.475, still not sufficiently high, in comparison with the similarity (0.7) of the operation pair [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$], since we intentionally introduced more changes to the PlainStatement.printFooter()$_{28}$ (see Sect. 4.1). It is therefore possible that the renaming of [PlainStatement.footerString()$_{27}$, PlainStatement.printFooter()$_{28}$] is not recognized when the renaming of [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$] is, if, for example, the *RenameThreshold* is 0.5.

However, *UMLDiff* knows that both HTMLStatement and PlainStatement extend the Statement class and their corresponding footerString() and printFooter() operations implement the abstract Statement.footerString() and Statement.printFooter() operations in the two compared versions. Implementing (or overriding) operations must have the same signature (i.e., the same identifier and parameter list) as the operations they implement (override). Therefore, if any one of them is renamed, all the rest must be renamed as well. Based on this definition, *UMLDiff* propagates the knowledge of the identified operation renamings along (both up and down) their implementation

(overriding) hierarchy, which may result in recognizing the renamings of abstract operations (which are not explicitly compared) and the renamings of other implementation (overriding) operations, as yet missed.

For example, based on the identified renaming [HTMLStatement.footerString()$_{27}$, HTMLStatement.printFooter()$_{28}$], *UMLDiff* first searches up to the top-most mapped ancestor class or interface (Statement in this case) and collects the pair of not-yet-mapped operations (the abstract operations [Statement.footerString()$_{27}$, Statement.printFooter()$_{28}$]) that are implemented (or overridden) by the identified pair of renamed operations and asserts them as a pair of renamed operations. And then based on the recovered operation renaming of the top-most ancestor class ([Statement.footerString()$_{27}$, Statement.printFooter()$_{28}$]), *UMLDiff* searches down all the pairs of the not-yet-mapped operations ([PlainStatement.footerString()$_{27}$, PlainStatement.printFooter()$_{28}$]) that implement (override) them and asserts all of them as pairs of renamed operations.

4.5 Mapping relations

In Sect. 4.4, we discussed the *UMLDiff* process for mapping the elements of two UML logical models corresponding to two versions of an evolving software system. This process produces three sets that contain (a) the model elements for which mappings have been identified in the two compared versions (i.e., matched, renamed, and moved), (b) the removed elements, and (c) the newly introduced elements respectively. *UMLDiff* then proceeds to map the relations between the model elements, i.e., to map the edge set (E_{before}, E_{after}) of the model graphs. This process step also produces three relation sets that contain (a) the mapped relations between the two model elements, (b) the removed relations, and (b) the newly introduced relations respectively.

The UML relations are defined by their types as described in Tables 11 and 12 and the UML model elements they relate. Given two model elements (v_{before}, v_{after}), where $v_{\text{after}} = \text{null}$ if v_{before} is removed and $v_{\text{before}} = \text{null}$ if v_{after} is added, *UMLDiff* collects all their relations in the two compared models. Two same-type relations of the model elements v_{before} and v_{after} in the two compared versions are matched, if the model elements they relate are contained in the mapped model element set, i.e., they map to each other. After *UMLDiff* finishes comparing the relations of all the pairs of the model elements, all unmatched relations that are still contained in E_{before} are assumed to have been *removed* and all unmatched relations in E_{after} are assumed to have been *added* when system evolves from the "before" version and the "after".

Note that the removed model elements contained in ($V_{\text{before}} - V_{\text{after}}$) and the newly added model elements contained in ($V_{\text{after}} - V_{\text{before}}$) have no counterpart in the compared models. The relations from (to) a removed model element are all removed and the relations from (to) an added model element are all newly added. For a pair of mapped elements (v_{before}, v_{after}), they may have matched, newly added, and/or removed relations. A removed (added) relation between the two model elements does not indicate that any of the elements it relates has been removed (added). For usage dependencies, *UMLDiff* also compares their *count* tag and reports the changes to the number of times they appear between the model elements.

Table 8 Mapping relations of the renamed [Customer.statement()$_{23}$, Customer.plainStatement()$_{27}$]

Version23		Version27/Version28	
Type of relation	Instances of relation	Type of relation	Instances of relation
[owner–feature]	[Customer, statement()]	[owner–feature]	[Customer, plainStatement()]
Usage$_{«read»}$	~~[statement(), _rental]~~	Usage$_{«read»}$	null
Incoming Usage$_{«call»}$	[Vids.main(), statement()]	Incoming Usage$_{«call»}$	[Vids.main(), plainStatement()]
Outgoing Usage$_{«call»}$	~~[statement(), getName()]~~ ~~[statement(), getTotalCharge()]~~ ~~[statement(), getTotalFrequent…()]~~ ~~[statement(), getMove()]~~ ~~[statement(), Rental.getCharge()]~~ ~~[statement(), getTitle()]~~ ~~[statement(), valueOf(double)]~~ [2] ~~[statement(), valueOf(int)]~~ ~~[statement(), Vector.elements()]~~ ~~[statement(), hasMoreElements()]~~ ~~[statement(), nextElement()]~~	Outgoing Usage$_{«call»}$	[plainStatement(), PlainStmt.value()]
Usage$_{«instantiate»}$	null	Usage$_{«instantiate»}$	[plainStatement(), PlainStatement]
Usage$_{«write»}$	null	Usage$_{«write»}$	null
Usage$_{«send»}$	null	Usage$_{«send»}$	null
Parameter	[statement(), return:String]	Parameter	[plainStatement(), return:String]
raidedSignal	null	raidedSignal	null
Reception	null	Reception	null

Table 8 lists all the relations of the renamed operation [Customer.statement()$_{23}$, Customer.plainStatement()$_{27}$], grouped according to their types. Consider the incoming-call relation as an example: the statement()$_{23}$ and plainStatement()$_{27}$ operations are called by the operations Vids.main(String[])$_{23}$ and Vids.main(String[])$_{27}$ respectively, which are matched. Thus, the renamed operation [Customer.statement()$_{23}$, Customer.plainStatement()$_{27}$] has a matched (regular font and right justified in Table 8) incoming-call relation. Similarly, they have a matched composition relation (both are declared in the matched class Customer) and a matched *[BehaviorFeature–parameter]* relation (both declare a return parameter of type String). All the removed relations are highlighted with strikethrough lines, while all the newly introduced relations are underlined with dash lines. The renamed operation [Customer.statement()$_{27}$, Customer.plainStatement()$_{28}$] no longer uses the attribute Customer._rental (a removed $Usage_{«read»}$ relation), but the attributes Customer._rental exist in both versions 23 and 27, i.e., they are matched. Furthermore, two (indicated by [2] at the end of the usage dependency) operation calls to String.valueOf(double) are removed (a removed outgoing $Usage_{«call»}$ relation with tag $count = 2$) when Customer.statement()$_{23}$ evolves to Customer.plainStatement()$_{27}$.

4.6 Recognizing behavior redistribution

Developers, sometimes, redistribute the behavior in the system in order to reorganize the inheritance hierarchy, restructure the usage dependencies between objects, or refactor a long method. After *UMLDiff* finishes mapping the model elements and their relations, it attempts to detect the redistribution of the behavior among operations, by analyzing the removals and additions of $Usage_{«read»/«write»/«call»/«instantiate»}$ dependencies of the mapped operations and the related removed or added operations along their transitive usage and generalization/abstraction relations.

Behavior redistribution is reported in terms of *Extract operation* and *Inline operation* changes. Note that our concepts of *Extract operation* and *Inline operation* are broader than the *Extract Method* and *Inline Method* introduced in Fowler's refactoring catalog (Fowler 1999), which are limited to refactoring only class internals.

We discuss in detail how *UMLDiff* detects *operation extraction* (*operation inlining* is detected in the same manner). Given two mapped operations $[o_{\text{before}}, o_{\text{after}}]$ with some removed outgoing $Usage_{«read»/«write»/«call»/«instantiate»}$ relations originating from o_{before}, *UMLDiff* identifies the candidate targets (o_{target}) of the behavior redistribution, as all the newly added operations that have a transitive relation with o_{after} through the relations of $Usage_{«call»}$ (incoming and outgoing), *Generalization* (overriding or overridden), $Abstraction_{«realize»}$ (implemented by), or a combination of them. We consider a removed outgoing usage relation $[o_{\text{before}}, e_{\text{before}}]$ from o_{before} as equal (not equivalent to relation match) to a newly added relation $[o_{\text{target}}, e_{\text{after}}]$ from one candidate target operation, if they are of the same type and the elements $[e_{\text{before}}, e_{\text{after}}]$ have been mapped. If the set of the outgoing usage relations from a candidate target operation o_{target} is a subset of the removed outgoing usage relations from o_{before}, or their intersection set is greater than the user-specific threshold, then *UMLDiff* asserts that o_{target} is extracted from o_{before}.

Let us now compare the versions 23 and 28 of our running example. *UMLDiff* identifies the renamed operation [Customer.statement()$_{23}$, Customer.plain-

Statement()$_{28}$] and reports the relation differences as shown in Table 8. Since the operation Customer.plainStatement() has not changed between versions 27 and 28, Table 8 reflects the relation changes between its versions 23 and 27 as well as its versions 23 and 28. Given the renamed operation [Customer.statement()$_{23}$, Customer.plainStatement()$_{28}$], Customer.statement()$_{23}$ is o_{before} in this example and Customer.plainStatement()$_{28}$ is o_{after}, and they have some removed outgoing usage relations (shown with strikethrough lines in Table 8). *UMLDiff* then collects all the newly added operations (the candidate targets) that have transitive usage and/or generalization/abstraction relations with Customer.plainStatement()$_{28}$. The operation Customer.plainStatement()$_{28}$ calls Statement.value()$_{28}$, which in turn calls Customer.getRentals()$_{28}$ and the three abstract operations Statement.printHeader()$_{28}$/printFooter()$_{28}$/printEachRental()$_{28}$, which are implemented by PlainStatement.printHeader()$_{28}$/printFooter()$_{28}$/printEachRental()$_{28}$ respectively. All these operations are newly introduced in version 28. Thus, the candidate targets of the behavior redistribution include Customer.getRentals()$_{28}$ and PlainStatement.printHeader()$_{28}$/printFooter()$_{28}$/printEachRental()$_{28}$. Note that *UMLDiff* ignores the abstract operations, since they have no outgoing usage.

Table 9 lists the removed outgoing usages of the operation Customer.statement()$_{23}$ and the candidate target operations and their newly added outgoing usages, when the model evolves from version 23 to 28. The outgoing usage relations of the target operations Customer.getRentals()$_{28}$ and PlainStatement.printHeader()$_{28}$/printEachRental()$_{28}$ are the subset of the removed outgoing usage relations of the Customer.statement()$_{23}$. *UMLDiff* asserts that these three target operations have been extracted from the Customer.statement()$_{23}$. On the other hand, the intersection of the new outgoing usages of the target operations PlainStatement.value()$_{28}$/printFooter()$_{28}$ and the removed outgoing usages of the Customer.statement()$_{23}$ is not empty. Depending on the user-specific threshold, the target operations PlainStatement.value()$_{28}$/printFooter()$_{28}$ may or may not be asserted as being extracted from the Customer.statement()$_{23}$. Clearly, as the system evolved from version 23 to 28, the behavior of the operation Customer.statement()$_{23}$ has been redistributed and encapsulated in a separate strategy object in version 28, which is defined by the abstract class Statement and its two implementation classes PlainStatement and HTMLStatement.

4.7 Comparing attributes of mapped model elements

Finally, *UMLDiff* compares the inherent attributes and the tagged values of the mapped UML model elements. For the visibility attribute, *UMLDiff* reports the changes as either of the following: (a) *up*: the access modifier has become less restrictive in the "after" version; (b) *down*: the access modifier has become more restrictive in the "after" version; and (c) *match*: visibility has not changed. The access modifiers can be private, package, protected, or public, in decreasingly restrictive order. For all other attributes and tagged values, *UMLDiff* simply reports whether they are of the same value or not. For example, *UMLDiff* reports that the *visibility* of the two renamed operations [getTotalCharge()$_{27}$, getAllCharge()$_{28}$] and [getTotalFrequentRenterPoints()$_{27}$, getAllFrequentRenterPoints()$_{28}$] has been changed *up* from package to public. Furthermore, the deprecation status of the matched operation [Customer.plainStatement()$_{27}$, Customer.plainStatement()$_{28}$] has been changed from false to true.

Table 9 Redistribute semantic behavior among operations

Version23		Version28	
Type of relation	Instances of relation	Type of relation	Instances of relation
Customer.statement()		Customer.getRentals()	
Usage$_{«read»}$	~~[statement(), _rental]~~	Usage$_{«read»}$	[getRentals(), _rental]
Usage$_{«call»}$			
		PlainStatement.printHeader()	
	~~[statement(), getName()]~~	Usage$_{«call»}$	[printHeader(), getName()]
		PlainStatement.printFooter()	
	~~[statement(), getTotalCharge()]~~	Usage$_{«call»}$	[printFooter(), getAllCharge()]
	~~[statement(), getTotalFrequent…()]~~		[printFooter(), getAllFrequent…()]
			[printFooter(), Double.toString()]
			[printFooter(), Integer.toString()]
		Usage$_{«instantiate»}$	[printFooter(), Double]
			[printFooter(), Integer]
		PlainStatement.printEachRental()	
	~~[statement(), getMove()]~~	Usage$_{«call»}$	[printEach…(), getMoive()]
	~~[statement(), Rental.getCharge()]~~		[printEach…(), Rental.getCharge()]
	~~[statement(), getTitle()]~~		[printEach…(), getTitle()]
	~~[statement(), Str.valueOf(double)]~~		[printEach…(), Str.valueOf(double)]
		PlainStatement.value()	
	~~[statement(), Vector.elements()]~~	Usage$_{«call»}$	[value(), Vector.elements()]
	~~[statement(), hasMoreElements()]~~		[value(), hasMoreElements()]
	~~[statement(), nextElement()]~~		[value(), nextElement()]
			[value(), printHeader()]
			[value(), printFooter()]
			[value(), printEachRental()]
	~~[statement(), Str.valueOf(double)]~~		
	~~[statement(), Str.valueOf(int)]~~		

5 Evaluation

To date, we have conducted several case studies, analyzing the evolution of Java software systems using *UMLDiff* to compare their subsequent versions. Using various types of statistical analyses of the *UMLDiff* change reports, we showed that one could form intuitions regarding the evolution phases and styles of classes and the overall systems (Xing and Stroulia 2004a, 2005c). We showed how Apriori associative mining, of information derived from *UMLDiff* change reports, can detect co-evolution of sets of classes (Xing and Stroulia 2006a). Using a suite of complex queries, we demonstrated that various types of refactorings could be recognized with higher precision than what is covered in the system documentation (Schofield et al. 2006; Xing

and Stroulia, 2006b, 2006c). Furthermore, we discussed how, based on analysis of recurring design-evolution patterns, one could offer advice to developers maintaining the system (Xing and Stroulia 2004b, 2005a). The quality and usefulness of all these analyses depend on the quality of the original change reports produced by *UMLDiff*.

In our own earlier work (Xing and Stroulia 2005b), we used JFreeChart (http://www.jfree.org/jfreechart/) as the subject system, to qualitatively evaluate the factors that may impact the quality of *UMLDiff* results, such as the user-defined renaming and move thresholds, the regularity of the usage of the versioning system, and the time lapse between subsequent versions. We showed that *UMLDiff* has high precision and recall as long as the versioning system is regularly used and is fairly robust to the user's choice of parameters. More specifically, we pointed out three typical situations in which *UMLDiff* may get confused:

- *UMLDiff* is based on lexical-similarity and structure-similarity heuristics. If two "irrelevant" model elements have very similar names and relations to other elements, they may be erroneously identified as renamings or moves.
- *UMLDiff* assumes that enough entities remain the "same" between two compared versions. If all or most of the model elements related to two renamed or moved elements were also renamed and/or moved, the structure-similarity heuristic may fail and thus *UMLDiff* may miss the renamings or moves.
- When two renamed or moved model elements have very few relations with other elements, it is difficult for *UMLDiff* to determine whether or not they represent a single conceptual element in the two compared system versions.

In this paper, we still use the JFreeChart system as the subject to evaluate the expanded *UMLDiff* algorithm.

5.1 The run-time performance of *UMLDiff*

First, we examined the run-time performance of *UMLDiff* algorithm. The run-time complexity of *UMLDiff* is determined by the renaming and move recognition process, which require the pair-wise comparison of the not-yet-mapped model elements in two compared versions of the system model. Through the use of appropriate in-memory data structures and efficient database indexing, the run-time complexity of the renaming and move recognition process is $O(\alpha \cdot N \cdot M)$, where N and M are the number of not-yet-mapped model elements of the same type in two compared versions respectively, and in the real world software systems, the α is usually very small. Furthermore, the actual time cost of *UMLDiff* is affected by the size of the system and the number of its versions, i.e., the size of JDEvAn database.

Table 18 (see Appendix 3) summarizes the run-time performance of *UMLDiff* when comparing the subsequent releases of the JFreeChart system, with *Rename Threshold* $= 0.3$ and *MoveThreshold* $= 0.3$. The column "Versions" indicates that the information summarized in a particular row is collected when the system evolved from the version of one row above to this version. The columns "N" and "M" list the number of not-yet-mapped model elements in the two compared versions. The column "#Comparison" lists the number of comparisons that *UMLDiff* performed for identifying the attribute/operation renamings and moves. The term α is computed as $(\#\text{Comparison})/(N \cdot M)$. As shown in Table 18, the α is very small. Table 18 also

presents the recalls of attribute/operation renamings and moves, which indicates that the *UMLDiff* structure-similarity heuristics is quite effective at recovering the reamed and moved model elements by comparing only a very small subset of not-yet-mapped candidates.

5.2 *UMLDiff* robustness

Next, we examined several factors that can impact the quality (in terms of precision and recall) of the renamings and moves reported by *UMLDiff*. In the *UMLDiff* context, given the total number of changes that have occurred between two versions (M_{actual}) and the number of changes reported by *UMLDiff* (M_{reported}), precision is the percentage of the correctly reported changes $(M_{\text{actual}} \cap M_{\text{reported}})/M_{\text{reported}}$ and recall is the percentage of changes reported $(M_{\text{actual}} \cap M_{\text{reported}})/M_{\text{actual}}$.

Table 19 summarizes the design changes (including the changes correctly identified by *UMLDiff* and the ones *UMLDiff* missed, which were manually added through our inspection of the *UMLDiff* results with JDEvAn tool) in the evolution of JFreeChart system. The contexts of the table serve as the ground truth, i.e. M_{actual}, for evaluating the impact of various factors that can affect the *UMLDiff* quality. To discuss the impact of each particular factor, we fix the others at the values that enable the identification of most renaming or move instances. Furthermore, we focus on the renamings and moves of classes/interfaces, attributes, and operations, since in our experience with several case studies, no subsystem/package renamings and moves were ever erroneously reported or missed by *UMLDiff*.

We comparatively evaluated the appropriateness of different lexical-similarity metrics for assessing the name similarity of two compared model elements. Table 20 summarizes the number of identified renaming instances, with different name-similarity metrics and the $RenameThreshold = 0.3$. The impact of choosing a particular name-similarity metric was most pronounced when recognizing attribute renamings (precision ranging from 87.5% to 90.1% and recall ranging from 94.6% to 99.7%), in contrast to recognizing class/interface renamings and operation renamings with identifier changes, where the choices of different name-similarity metrics were almost indistinguishable. Overall, none of the three used metrics, Char-LCS, Word-LCS and Char-Pair, is significantly better, although the Char-Pair metric seems to produce results with a better balance of precision and recall.

We examined the effectiveness of the two techniques for propagating the knowledge about the identified renamings and moves through usage and inheritance relations. They are both useful in increasing the recall of renamings and moves; the corresponding slight decrease in precision should not be a major concern, since the users should be able to easily recognize and filter out the false positive instances reported. For example, with the renaming and move thresholds set to 0.3, about 1.2% of all the operation renamings and moves were recovered through second and third rounds of renaming and move recognition; about 7.6% of all the operation renamings were recovered through propagating the operation renamings depending on the generalization/abstraction relations. Less than 10 instances were erroneously reported due to the application of these two techniques. Thus, we believe these two tech-

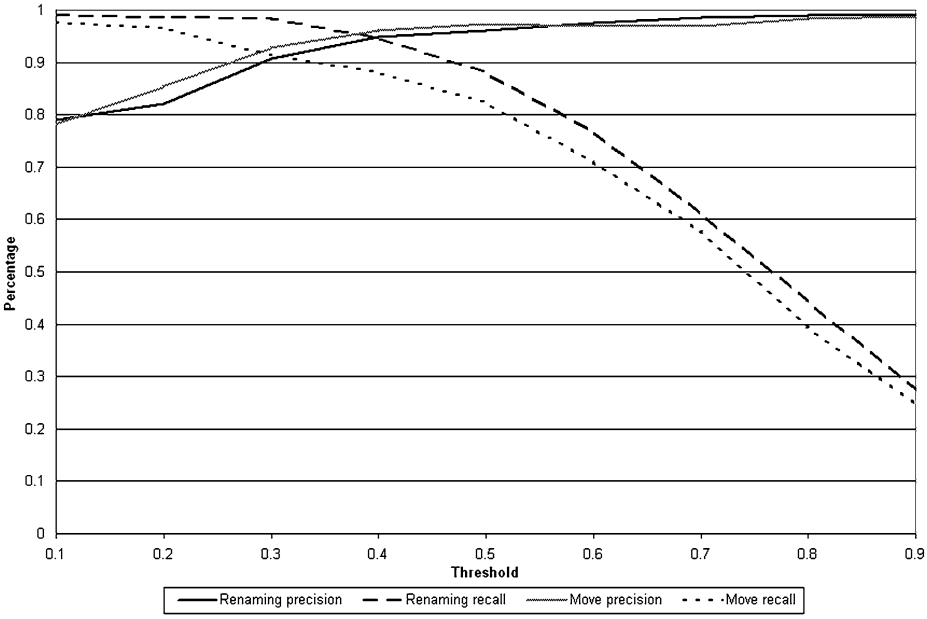

Fig. 6 The impact of the user-specific renaming and move thresholds

niques are very effective at recovering renamings and moves that would otherwise be missed.

We evaluated the impact of additional sources of information, i.e., comment and transitive usage dependency, on *UMLDiff*'s accuracy. Tables 21 and 22 summarize the impact of comment similarity and transitive usage similarity on the precision and recall of identified renamings and moves of classes/interfaces, attributes, and operations. Overall, the comments of model elements and their transitive usage dependencies can effectively inform the process to further increase its recall, albeit at a small precision cost. Based on the estimated number of changes, the time lapse between two compared versions and the need for the more coverage of changes or the more precise results with shorter comparison time, the users may turn on or off these additional sources of information when comparing renaming and move candidates.

We examined the impact of the user-specified renaming and move thresholds on the quality of the *UMLDiff* results. We run *UMLDiff* on JFreeChart with the renaming and move thresholds set to 0.1 through 0.9, with 0.1 increment (using the Char-LCS lexical-similarity metric, with comment-similarity, and transitive-usage-similarity) and computed the precision and recall of renamings and moves at each threshold. The results are summarized in Fig. 6. We found that a renaming threshold slightly higher than the move threshold, with both being within the 30% to 50% range, is an effective setting for accurately recognizing both renamings and moves. Furthermore, by considering the context from and to which the

model elements are moved, this version of *UMLDiff* exhibits a more balanced combination of precision and recall for moves, as compared to the original algorithm (Xing and Stroulia 2005b), although it suffers a slight deterioration in the move recall.

6 Conclusions

In this paper, we described *UMLDiff,* an object-oriented logical-design differencing algorithm. This algorithm advances the state of the art in software-evolution analysis in several ways. Because it compares software versions at the design level, its results are more directly relevant to the evolutionary-development process than either lexical or metrics differencing. Because it is aware of the UML semantics, its results are more intuitive than other structure-differencing algorithms, relying on low-level representations such as ASTs, program-dependency graphs or XML. Because it is automated, it does not rely on subjective interpretation, as do visualization approaches.

In our earlier work, we had examined an earlier version of *UMLDiff* and we had found it to be sensitive to irregular usage of the versioning system, but with high precision and recall of design changes when the versioning system is used regularly. Furthermore, it is robust to the user's choice of parameters about how similar the two model elements have to be, in order for the latter to be recognized as a move or a renaming of the former.

In this paper, we describe several extensions to the original *UMLDiff* algorithm, which are illustrated through a concise running example. We also experimentally evaluate the new algorithm variant, focusing on its run-time performance, the appropriateness of different lexical-similarity metrics, the effectiveness of the two techniques for propagating the knowledge about the identified renamings and moves along usage dependency and inheritance hierarchy, the impact of model element's comments and transitive usage dependencies on *UMLDiff*'s accuracy, and the effective range for the user-specified renaming and move thresholds.

An accurate *UMLDiff,* that can precisely recall the design changes that an object-oriented system has suffered, can provide the basis on which further analyses can be developed for classifying the evolution of individual classes into a taxonomy of evolution profiles (Xing and Stroulia 2004a), discovering co-evolution of sets of classes (Xing and Stroulia 2006a), recognizing phases and styles in the evolution of systems and their parts (Xing and Stroulia 2005c), detecting refactorings (Schofield et al. 2006; Xing and Stroulia, 2006b, 2006c). In turn, these analyses can provide insights for supporting decision making for how to further evolve the system (Xing and Stroulia 2004b, 2005a).

In the future, we plan to extend *UMLDiff* to discover changes in the dynamic behavior, such as UML sequence diagrams, of the subsequent system versions and to infer correlations between static structure and dynamic behavior changes for the purpose of impact analysis.

Acknowledgement This work was supported by NSERC. The authors also wish to thank the anonymous reviewers whose insightful comments steered the evolution of this paper to its current version.

Appendix 1

Table 10 The UML model elements

Metaclass stereotype	Description
Subsystem	A subsystem is a grouping of model elements that represents a behavioral unit in a physical system
Package	A package is a grouping of model elements
Class	A class declares a collection of attributes, operations and methods that fully describe the structure and behavior of a set of objects. A class acts as the namespace for various kinds of contained elements defined within its scope, including classes and interfaces
Interface	An interface is a named set of operations that characterize the behavior of an element
DataType	A data type is a type whose values have no identity
Attribute	An attribute is a named piece of the declared state of a classifier, which refers to a static feature of a model element. An attribute may have an initValue specifying the value of the attribute upon initialization
Operation «create» «initialize»	An operation is a service that can be requested from an object to effect behavior, which refers to a dynamic feature of a model element
Method	A method is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation
Parameter	A parameter is a declaration of an argument to be passed to, or returned from an operation
Exception	An exception is a signal raised by behavioral features typically in case of execution faults
Reception	A reception is a behavioral feature and declares that the classifier containing the feature reacts to the signal designated by the reception feature

Table 11 The UML relations among model elements

Metaclass stereotype	Description
Generalization	A generalization is a taxonomic relation between a more general element (parent) and a more specific element (child)
Abstraction «realize»	An abstraction is a dependency relation that relates two elements or sets of elements that represent the same concept at different levels of abstraction
Usage «call» «instantiate» «send» «read» «write»	A usage is a dependency relation in which one element requires another element (or set of elements) for its full implementation or operation
Association	An association is a declaration of a semantic relation between classifiers that can be of three different kinds: (1) ordinary association, (2) composite aggregate, and (3) shareable aggregate. There are three meta-composition and five ordinary meta-associations defined in the metamodel, which are described in Table 12

Table 12 The compositions and associations among model elements

Metarelation	Description
namespace–ownedElement	A namespace is a model element that can own other model elements. The element ownership is used for unstructured contents such as the contents of a package or a class declared inside the scope of another class
owner–feature	A classifier declares a collection of features. The features are the inherent semantic parts of a classifier
BehaviorFeature–parameter	An operation declares an ordered list of parameters. The parameters are the inherent semantic parts of an operation
typedParameter–type	Designates a classifier to which an argument value of a parameter must conform. The type must be a class, interface, or datatype
typedFeature–type	Designates a classifier as whose instances are values of the attribute. The type must be a class, interface, or datatype
context–raisedSignal	Designates exceptions that may be raised by behavioral features, such as operations when execution faults happen
reception–signal	Designates reception features that handle the signal
Method–specification	Designates an operation that the method implements

Table 13 *UMLDiff*-specific tagged values attached to model elements

Tagged values	Base metaclass	Description
comment	ModelElement	Any documentation attached to the model element
isFromModel	ModelElement	If the model element is imported from a model other than the current one, false. Otherwise, true
deprecated	ModelElement	If the model element is obsolete and will be removed from the model in the future, true. Otherwise, false
overloaded	ModelElement	If the operation is overloaded, true Otherwise, false
count	Usage	The number of times a usage dependency appears between the client and supplier elements

Appendix 2

Table 14 Mapping Java language constructs to UML model elements

Java constructs	UML metaclasses
Java primitive type	ProgrammingLanguageDataType
Java array type	ProgrammingLanguageDataType
Java software subsystem	Subsystem
Java package	Package
Java class	Class
Java interface	Interface
Java field	Attribute
Java method	Operation
Java constructor	Operation$_{\text{«create»}}$

Table 14 (*Continued*)

Java constructs	UML metaclasses
Java class initializer	$\text{Operation}_{\text{«initialize»}}$
Java field initializer	Attribute's initValue
Java parameter	Parameter
The return type of Java method	Parameter whose name = 'return' and kind = return

Table 15 Mapping Java relations to UML metarelations

Java relations	UML metarelations
Contain	meta-composition [namespace–ownedElement]
Declare	meta-composition [owner–Feature]
Method/constructor parameter	meta-composition [BehaviorFeature–parameter]
extends	Generalization
implements	$\text{Abstraction}_{\text{«realize»}}$
new XXX(...)	$\text{Usage}_{\text{«instantiate»}}$
Use field	$\text{Usage}_{\text{«read»}}$
Change field value	$\text{Usage}_{\text{«write»}}$
Method/Constructor call	$\text{Usage}_{\text{«call»}}$
throw statement	$\text{Usage}_{\text{«send»}}$
Field data type	meta-association [typedFeature–type]
Parameter type	meta-association [typedParameter–type]
Method return type	meta-association [typedParameter–type] for the parameter whose kind = return
throws clause	meta-association [context–raisedSignal]
catch clause	meta-association [reception–signal]

Table 16 Mapping Java modifiers to the attributes of UML metaclasses

Java modifiers	The attributes of UML metaclasses
public, protected, private	Visibility of ElementOwnership or feature
static	ownerScope = classifier of feature
final	isLeaf = true of GeneralizableElement or operation
synchronized	Concurrency = guarded of operation
abstract	isAbstract = true of GeneralizableElement or operation
transient	Persistence = transitory of attribute

Table 17 Mapping Java language features to *UMLDiff*-specific tagged values

Java language features	*UMLDiff-specific tagged values*
Javadoc description before block tags	comment
Java construct belongs in the source code	isFromModel = true
Javadoc contains @deprecate tag	deprecated = true
Several methods/constructors with the same identifier exist	overloaded = true

Appendix 3

Table 18 The run-time performance of *UMLDiff*

Versions	Attribute and operation renamings					Attribute and operation moves				
	N	*M*	#Comparison	α	Recall	*N*	*M*	#Comparison	α	Recall
0.6.0	154	357	1119	0.05	0.98	203	378	64	0.002	0.95
0.7.0	6	33	36	0.36	1.0	4	188	2	0.004	1.0
0.7.1	72	169	298	0.06	1.0	48	190	16	0.003	1.0
0.7.2	56	101	130	0.06	1.0	51	104	7	0.003	N/A[a]
0.7.3	5	10	6	0.12	1.0	1	18	2	0.13	1.0
0.7.4	32	34	14	0.04	1.0	22	57	18	0.03	1.0
0.8.0	42	69	37	0.04	1.0	83	233	37	0.004	1.0
0.9.0	170	313	1519	0.06	0.94	216	842	244	0.003	0.85
0.9.1	2	10	2	0.11	1.0	0	89	0	0	N/A
0.9.2	99	99	1168	0.29	0.97	60	105	9	0.004	1.0
0.9.3	109	207	618	0.06	0.95	144	1003	111	0.001	1.0
0.9.4	212	297	1737	0.07	1.0	121	406	74	0.004	0.7
0.9.5	564	721	4842	0.02	0.97	422	1303	267	0.001	1.0
0.9.6	10	43	35	0.34	1.0	1	42	0	0	N/A
0.9.7	138	242	213	0.01	1.0	190	583	227	0.004	0.92
0.9.8	52	54	24	0.01	1.0	31	96	85	0.04	1.0
0.9.9	309	625	3475	0.04	0.99	406	876	470	0.002	0.92
0.9.10	233	361	3044	0.07	1.0	137	213	23	0.001	0.30
0.9.11	16	73	10	0.02	1.0	8	237	11	0.01	1.0
0.9.12	106	383	690	0.03	1.0	124	550	345	0.008	0.90
0.9.13	83	174	410	0.04	1.0	6	161	3	0.005	1.0
0.9.14	168	333	821	0.02	0.96	89	427	27	0.001	0.80
0.9.15	21	43	23	0.04	1.0	13	180	2	0.001	1.0
0.9.16	70	76	95	0.04	1.0	85	202	67	0.008	0.98
0.9.17	272	495	3389	0.06	0.94	168	974	282	0.003	0.92
0.9.18	59	119	230	0.10	1.0	35	191	4	0.001	1.0
0.9.19	219	385	3444	0.08	0.98	242	418	108	0.002	0.92
0.9.20	9	21	5	0.08	1.0	4	64	1	0.008	N/A
0.9.21	118	272	574	0.02	1.0	1043	349	75	0.001	0.35
1.0.0	129	354	641	0.03	0.97	53	531	29	0.002	1.0

[a]Indicates that there are no moves of model elements in the compared models

Table 19 The number of design changes *UMLDiff* reports in the evolution of JFreeChart

Element renaming	2180
Element move[a]	957
Extract operation	533
Inline operation	95
Data type and return type change	1056
Abstraction$_{\text{«realize»}}$ change	1032
Generalization change	186
Visibility change	868
Other attribute/tagged-value change	607
Total	**7514**

[a]The moved methods may also involve identifier changes. Such instances are manually added during the inspecting session of *UMLDiff* results

Table 20 Recognizing renamings with different name-similarity metrics

		Correct	Wrong	Precision	Recall
Class and interface	Char-LCS	123	48	71.9%[a]	98.4%
	Char-Pair	123	47	72.4%	98.4%
	Word-LCS	123	47	72.4%	98.4%
Attribute	Char-LCS	295	42	87.5%	99.7%
	Char-Pair	290	33	89.7%	97.9%
	Word-LCS	280	31	90.1%	94.6%
Operation	Char-LCS	794	126	86.3%	97.2%
	Char-Pair	792	116	87.2%	97.0%
	Word-LCS	793	118	87.0%	97.1%
Overall	Char-LCS	1212	216	84.9%	97.9%
	Char-Pair	1205	196	86.0%	97.3%
	Word-LCS	1196	196	85.9%	96.6%

[a]The low precision of class renaming is due to the large amount (31 out of 48) of demo and unit test classes being identified as renamed. Most of them can be prevented with higher renaming threshold

Table 21 Recognizing renamings and moves with and without comment-similarity

			Correct	Wrong	Precision	Recall
Renamings	Class and interface	Without	121	36	77.0%	96.8%
		With	123	48	71.9%	98.4%
	Attribute	Without	278	26	91.5%	94.0%
		With	295	42	87.5%	99.7%
	Operation	Without	1590	62	96.3%	91.9%
		With	1696	129	93.0%	98.3%
Moves	Class and interface	Without	296	0	100.0%	95.8%
		With	299	0	100.0%	96.8%
	Attribute	Without	186	5	97.4%	83.4%
		With	200	9	95.7%	89.7%
	Operation	Without	343	25	93.2%	80.7%
		With	375	58	86.7%	88.3%

Table 22 Recognizing operation renamings and moves with/without transitive usage similarity

		Correct	Wrong	Precision	Recall
Operation renamings	Without	1672	74	95.8%	96.6%
	With	1696	129	93.0%	98.3%
Operation moves	Without	375	55	87.2%	88.3%
	With	375	58	86.7%	88.3%

References

Apiwattanapong, T., Orso, A., & Harrold, M. J. (2004). A differencing algorithm for object-oriented programs. In *Proceedings of the 19th international conference on automated software engineering* (pp. 2–13).

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2000). Finding refactorings via change metrics. ACM SIGPLAN Notices, **35**(10), 166–177.

Egyed, A. (2001). Scalable consistency checking between diagrams—the ViewIntegra approach. In *Proceedings of the 16th international conference on automated software engineering*.

Eick, S. G., Steffen, J. L., & Sumner, E. E. (1992). SeeSoft—A tool for visualizing line-oriented software statistics. IEEE Transactions on Software Engineering, **18**(11), 957–968.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. IEEE Transactions on Software Engineering, **27**(1), 1–12.

Fischer, M., Pinzger, M., & Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th international conference on software maintenance* (pp. 23–32), September 2003.

Fowler, M. (1999). Refactoring: Improving the design of existing code. Reading: Addison–Wesley.

Godfrey, M., & Zou, L. (2005). Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering, **31**(2), 166–181.

Horwitz, S. (1990). Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN'90 conference on programming language design and implementation* (pp. 234–246), June 1990.

Jackson, D., & Ladd, D. A. (1994). Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of 9th international conference on software maintenance* (pp. 243–252), September 1994.

Lanza, M. (2001). The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th international workshop on principles of software evolution* (pp. 37–42).

Lehman, M. M., & Belady, L. A. (1985). Program evolution-processes of software change. London: Academic Press, p. 538.

Lorenz, M., & Kidd, J. (1994). Object-oriented software metrics: A practical approach. New York: Prentice Hall.

OMG (2003). *Unified Modeling Language Specification*. formal/03-03-01, Version 1.5, http//www.omg.org.

Ohst, D., Welle, M., & Kelter, U. (2003). Difference tools for analysis and design documents. In *Proceedings of the 19th international conference on software maintenance* (pp. 13–22), September 2003.

Ryder, B. G., & Tip, F. (2001). Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering* (pp. 46–53).

Rysselberghe, F. V., & Demeyer, S. (2003). Reconstruction of successful software evolution using clone detection. In *Proceedings of the international workshop on principles of software evolution* (pp. 126–130), September 2003.

Schofield, C., Tansey, B., Xing, Z., & Stroulia, E. (2006). Digging the development dust for refactorings. In *Proceedings of the 14th international conference on program comprehension* (pp. 23–34), June 2006.

Simon, K. (1986). An improved algorithm for transitive closure on acyclic digraphs. In *Theoretical computer science: Vol. 58, Automata, languages and programming* (pp. 376–386).

Spanoudakis, G., & Kim, H. (2001). Reconciliation of object interaction models. In *Proceedings of the 7th international conference on object oriented information systems* (pp. 47–58), August 2001.

Tu, Q., & Godfrey, M. W. (2002). An integrated approach for studying architectural evolution. In *Proceedings of the 10th international workshop on program comprehension* (pp. 127–136).

Wagner, R. A., & Fischer, M. J. (1974). The string-to-string correction problem. Journal of the ACM, **21**(1), 168–173.

Xing, Z., & Stroulia, E. (2004a). Understanding class evolution in object-oriented software. In *Proceedings of the 12th international workshop on program comprehension* (pp. 34–43), June 2004.

Xing, Z., & Stroulia, E. (2004b). Design mentoring based on design evolution analysis. In *Proceedings of eclipse technology exchange workshop, OOPSLA 2004*, Vancouver BC, Canada.

Xing, Z., & Stroulia, E. (2005a). Towards mentoring object-oriented evolutionary development. In *Proceedings of the 21st international conference on software maintenance* (pp. 621–624), September 2005.

Xing, Z., & Stroulia, E. (2005b). UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th international conference on automated software engineering* (pp. 54–65), November 2005.

Xing, Z., & Stroulia, E. (2005c). Analyzing the evolutionary history of the logical design of object-oriented software. IEEE Transactions on Software Engineering, **31**(10), 850–868.

Xing, Z., & Stroulia, E. (2006a). Understanding the evolution and co-evolution of classes in object-oriented systems. International Journal of Software Engineering and Knowledge Engineering, **16**(1), 23–52.

Xing, Z., & Stroulia, E. (2006b). Refactoring practice: How it is and how it should be supported—an eclipse case study. In *Proceddings of the 22nd international conference on software maintenance* (pp. 458–468), September 2006.

Xing, Z., & Stroulia, E. (2006c). Refactoring detection based on UMLDiff change-facts queries. In *Proceedings of the 13th working conference on reverse engineering* (pp. 263–274), October 2006.

Yang, W. (1991). Identifying syntactic differences between two programs. Software—Practice and Experience, **21**(7), 739–755.