Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study

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

Abstract

Refactoring is an important activity in the evolutionary development of object-oriented software systems. Yet, several questions about the practice of refactoring remain unanswered, such as what fraction of code modifications are refactorings and what are the most frequent types of refactorings. To gain some insight in this matter, we conducted a detailed case study on the structural evolution of Eclipse, an integrated-development environment (IDE) and a plugin-based framework. Our study indicates that 1) about 70% of structural changes may be due to refactorings; 2) for about 60% of these changes, the references to the affected entities in a component-based application can be automatically updated by a refactoring-migration tool if the relevant information of refactored components can be gathered through the refactoring engine; and 3) state-of-the-art IDEs, such as Eclipse, support only a subset of commonly applied low-level refactorings and lack support for more complex ones, which are also frequent. Based on our findings, we draw some conclusions on high-level design requirements for a refactoring-based development environment.

1 Introduction

Today, most object-oriented software systems are developed using an evolutionary process model. Change is an integral part of the evolutionary development lifecycle. An important kind of change to object-oriented software is refactoring [5,6,12,15]. The goal of refactoring is to improve the quality of the software system, such as its understandability, extensibility and maintainability, without affecting its overall functionality and behavior.

Because of this beneficial impact to software design, some modern integrated development environments (IDEs), such as Refactoring Browser [16] and Eclipse [19], provide semiautomatic support for applying the most commonly used, low-level refactorings, such as for example "Rename Field" and "Move Method" [6]. Refactoring support within IDEs has made it less cumbersome and expensive to improve code quality.

However, refactoring activities become more challenging in the context of reuse-based development, which is a common practice today. To some extent, software reuse simplifies the design of new systems but, at the same time, it implies that their design and implementation heavily depends on the components they reuse. If the application developer wants to refactor the application code, their activity has to be limited to changing the internal implementation of the application elements. At the same time, ideally, the developers of the reusable components should limit themselves to extending the components' application programming interface (API) and should not remove or change existing parts of the API, or else they may cause the client applications to fail. In practice, however, the new versions of reusable components do change their APIs [18], which implies a need for the systems that use these earlier APIs to adapt. The more widely adopted the underlying component framework, the higher the cost of breaking client code becomes, which is why component developers often have to refrain from making changes that might improve the quality of the components.

The potential challenges that refactoring may bring about in the context of reuse-based development gives rise to the need for understanding how this activity is actually practiced. Unfortunately, little work has been done on investigating what fraction of changes over the lifecycle of object-oriented software system are refactorings and of what type. In part, this may be due to the fact that there has been no substantial tool support for detecting and classifying structural evolution and, more often than not, available documentation – change logs and release notes – reports only a subset of the actual changes. Thus, several important questions remain unanswered:

- What proportion of the structural changes in the evolution of a system are the results of refactoring?
- What are the typical refactorings applied in practice?
- What aspects of a system's structural evolution can be automatically gathered?
- Which of these types are "safe" to client applications that reuse the refactored system?
- What type of support should modern IDEs provide and how might this support be implemented?

In this paper, we describe a detailed case study we conducted on the structural evolution of Eclipse. Eclipse is a large-scale industrial framework that has been under development for about four years. In the process, it has acquired a large user base and a multitude of applications have been built on it. Eclipse is built as a plugin-based framework. Its users can simply use it as an IDE, but they can also extend or build their own plugins from the existing ones. Since version 3.0, Eclipse introduced a concept of a rich client platform, which allows its users to build stand-alone applications from a subset of plugins. Therefore, studying the structural evolution of Eclipse can help us understand the design requirements for refactoring-based development environment from the perspectives of both the component developers and component users.

We studied the structural evolution of Eclipse with JDEvAn tool, which implements a design-level structural differencing algorithm, *UMLDiff* [18]. *UMLDiff* reports the structural changes between the two software versions in terms of (a) additions, removals, moves, renamings of packages, classes, interfaces, fields and methods, (b) changes to their attributes, such data type, visibility, and modifiers, and (c) changes of the dependencies among these entities. *UMLDiff* builds a detailed and accurate picture of the software system and its components' structural evolution, and provides a solid base for us to analyze and classify the structural changes that are refactorings.

The rest of the paper is structured as follows. Section 2 relates this work to previous research. Section 3 describes briefly the methodology we adopt to conduct this case study. Section 4 outlines the case study and Section 5 assesses the empirical data we collect on the structural evolution of Eclipse. Section 6 concludes with some interesting insights we obtained through this case study.

2 Related work

Refactoring has recently become an integral part of the evolutionary software development methodology, such as "Extreme Programming" [2]. The research area of refactoring was pioneered by Opdyke [15]. The books of Fowler [6] and Kerievsky [12] provide a good overview of the refactorings and how they can be used to accomplish architectural and design changes. Opdyke's Ph.D. thesis catalogs a number of refactorings, and lists a set of invariants that a refactoring must conform in order to be behavior-preserving, such as "type-safe assignments". In this paper, we describe our empirical study on the structural evolution of a large software project and summarize what the fraction of changes are behavior-preserving program transformations.

There has been some work at investigating the detection of refactorings. Demeyer et al. [4] define four heuristics based on the comparison of source-code metrics of two subsequent system snapshots to identify refactorings of three general categories. Ryssel-berghe [17] investigated the use of clone-detection to identify move and renaming refactorings. Godfrey and Zou [8] use origin analysis to detect the merging and splitting of source-code entities. The empirical study we conducted relies on a novel structural differencing algorithm, *UMLDiff* [18], that enables the identification of a rich set of elementary structural changes and fairly complex refactorings, which provides a solid base for us to study the structural evolution of a large object-oriented software project at fine-grained level.

Opdyke developed the first tool that provides automated refactoring support, which was implemented in Refactoring Browser [16]. Modern IDEs, such as Eclipse [19], offer automated support for most commonly used refactorings, such as rename or move. In this paper, we compare the refactorings that were actually performed in the evolution of the object-oriented software system with those supported by IDEs, and point out that modern IDEs do not provide automated support for all frequently used refactorings, especially high-level refactorings, such as for example inheritance-hierarchy reorganizations, that involve a set of relevant program entities.

Refactoring the reused components is often limited by the fear of breaking client code. When the breaking API changes happen, the developers of component-based applications take the burden of migrating their codes to the new version of reused components. Balaban et al.[1] developed a tool that allow the developers to define a mapping specification between legacy classes and their replacements so that obsolete library classes (such as Java Vector) can be replaced with their newer counterparts (such as Java ArrayList). CatchUp [9] is another attempt to relieve this burden by recording the refactorings, such as renamings, made by the component developers within an IDE, such as Eclipse, and then replaying them by the application developers on the client code to keep it updated. The results of our study confirm the usefulness of such migration tool support in the refactoring-based development environment. At the same time, they also point out the limitations of current tool support, especially the lack of support for high-level refactorings.

On the same base line, Dig and Johnson [5] conducted a similar empirical study on the role of refactorings in API migration. Both their study and ours found similar results. Their analysis relies on the changes documented in the release notes shipped with software systems. Our analysis is based on the structural changes automatically detected by the *UMLDiff* algorithm, which reports changes in much more detail than what is covered in the documentation (see Table 3, 4, and 5). This enables us to understand the actual refactoring practice and elicit some high-level design requirements for refactoring-based development environments.

3 JDEvAn and *UMLDiff*

The JDEvAn - "Java Design Evolution and Analysis" - tool, analyzes the evolving structure of design-level software artifacts and automatically produces reports of interesting trends and events during their evolution, such as refactorings that we discuss in this paper. In this section, we briefly discuss the basic JDEvAn functionalities relevant to recognizing structural changes.

JDEvAn focuses on the logical view [11] of object-oriented software systems, which concerns classes, the information they may own, the services they can deliver, and the associations and relative organization among them. Its primary input is the system's source code, residing in a versioning system. JDEvAn's fact extractor (based on the Eclipse DOM/AST model) recovers a model of the subject system's class design (according to the semantics of the UML static-structure model [14]). The extracted data models are stored in a PostgreSQL relational database.

JDEvAn's *UMLDiff* [18] is a domain-specific structural differencing algorithm, aware of UML semantics. Given two subsequent system versions, *UMLDiff* establishes that two entities of the same UML type in the corresponding models represent the "same" conceptual entity, based on their name similarity and structure similarity to other "established-to-be-same" entities. The comparison result is represented as a *change tree*, summarizing the modifications to the various design entities of the system, their attributes and dependencies, which are also stored in JDEvAn's database. Overall, there are five types of elementary structural changes identified by *UMLDiff*:

- Additions of new packages, classes, interfaces, methods, and fields;
- Removals of packages, classes, interfaces, methods, and fields;
- Movements of methods and fields from one class to another, movements of classes and interfaces from one class or package to another;
- Renamings of packages, classes, interfaces, methods (including parameter list changes), and fields; and
- Modifications of signature, such as visibility, modifiers, data (return) type, and parameter list, of classes, interfaces, methods and fields, and modifications of usage dependency, class inheritance and interface implementation.

Once the elementary structural changes between the two versions have been recognized, a suite of queries are applied to elicit more complex structural change patterns, as compositions of elementary changes, such as refactorings [6]. For example, an instance of the "Extract Superclass" refactoring can be recognized by a query for a set of related changes: (a) the *addition* of a new class, (b) the *modification* of one or more classes from extending other class, such as java.lang.Object to extending the newly added class, and (c) the *moving* of one or more fields and methods from these existing classes to their new superclass.

4 Overview of the Eclipse case study

At this time, Eclipse has nine releases between the first official version 2.0, released on June 27 2002, and the latest version 3.1, which was released on June 27 2005. In particular, we chose to compare three pairs of major releases 2.0 and 2.1, 2.1.3 and 3.0, and 3.0.2 and 3.1, because there were substantial changes between them (see Table 3, 4 and 5). According to their associated documentation, the remaining versions, 2.1.1, 2.1.2, and 3.0.1, included mostly bug fixes and minor modifications and we ignored them in this case study.

There are several reasons why we used Eclipse as the subject of our case study. First, the system has been undergoing substantial evolution in the past three years and we were interested to see how much of this evolution involves refactorings. This was an especially interesting question, given the fact the Eclipse is an IDE that supports refactorings. Second, it is well documented, especially the major releases on which we have focused, and that enables us to better assess the correctness of our refactoring extraction method. Third, the system is a platform on which multiple applications have been developed and this gives us the opportunity to study the potential impact of refactoring of reusable frameworks to their applications.

Eclipse consists of three subprojects and in this case study, we have focused on the JDT subproject, which defines about 40% of the classes and interfaces of the whole Eclipse platform. Table 1 and Table 2 report the numbers of ground entity and relation facts extracted by JDEvAn. They show a substantial increase in the number of program entities and relations between the pairs of versions we examined as opposed to small changes in the in-between versions that we excluded from our case study. This is additional evidence that Eclipse, most likely, underwent many changes when evolving from previous versions to these major releases.

Clearly, the numbers of program entities (407720) and relations (2220707) preclude us from manually inspecting them all. In addition, there is no complete documentation covering all the changes to these entities. To gain an insight on the modifications that these changes represent and their rationale, we applied *UMLDiff* to pairwise compare releases 2.0 and 2.1, 2.1.3 and 3.0, 3.0.2 and 3.1. Once all pairs of successive versions were UMLDiffed, the reported structural-evolution changes [18] were stored in the JDEvAn database. We then manually inspected the correctness of each reported structural change against the Eclipse source code, the accompanying Javadocs and source-code comments, and the change logs shipped with each major release version. Erroneously reported and missed changes were corrected with the support of JDEvAn tool. A substantial number of changes were discovered (with relationship similarity threshold set to 0.3) and are summarized in Table 3 (sorted by their frequency). Its intensive and varied structural-evolution history makes Eclipse an appropriate test-bed for the evaluation of the role of refactoring in the evolution of object-oriented software system.

For the time being, we did not allow *UMLDiff* to compute the complete set of usage-dependency differences for all the entities, which is a feasible but time-consuming process for such systems as Eclipse with a huge amount of entities. As a result, the analysis of some types of low-level refactorings, such as inline or extract method, encapsulate field, etc., which depend on the usage differences, is incomplete[1].

5 The empirical assessment of the structural evolution of Eclipse

In this section, we describe and assess the empirical data we collect in our study on the structural evolution of Eclipse. We will summarize our findings in next section.

5.1 Elementary structural changes

First, let us examine seven types of elementary structural changes reported by *UMLDiff* algorithm. We start our discussion with renamings, which we expect to be the more benign changes, i.e., changes that are likely to be behavior preserving and therefore relatively easy to propagate their implications to the client applications of the earlier version. We then proceed to examine increasingly "suspect" modifications, such as moves, modifier and visibility changes, data-type changes, inheritance-relation changes, and entity additions and removals.

5.1.1 Program-entity renamings

There are 4891 renamings of various types of program entities, including packages, classes and interfaces, methods (including constructors) and fields (see Table 4). Note that 2 moved classes, 1 moved interface, 42 (5+8+29) moved fields, and 264 (19+162+83) moved methods were renamed as well as moved. Renamings of packages, classes and interfaces, and fields involve changes to their identifier. Method renamings may involve changes to the whole method signature (including identifier and/or parameter list). About 24% (984/4170) renamed methods had only their identifiers changed.

Through inspection, we identified several plausible motivations behind renamings:

1) *Conformance to a consistent naming scheme;*
2) *Reflecting the semantics of an internal implementation change to the entity*;
3) *Concept merging or splitting; and*
4) *Maintaining backward compatibility with earlier versions.*

A consistent naming scheme improves code readability and understandability, especially when the identifiers allude to the functions of the program entities. 29 renamings were simply to correct spelling or wrong names. Some of the renamings were motivated by the adoption of a more meaningful name for the entity: clearly, fSelectedCU, isOnBuildPath() and AccessorClassCreator reveal the purposes of the program entities much more clearly than their precursors fCU, checkJavaElement() and AccessorClass. In other cases, the renamings were more "syntactic" aiming to simply conform with the adopted naming convention. For example, 79 fields were renamed to remove the prefix "f"; at the same time, it is interesting to note that 11 other fields were renamed by adding the same prefix "f". This phenomenon may be because different Eclipse plugins adopt different and occasionally contradictory naming conventions. In another case, 12 fields were capitalized because they were declared "static" and/or "final", while 6 fields were converted to lowercase when they stopped being static final constants.

Renamings also reflect implementation changes. For example, the data (return) type of 723 renamed fields (methods) was also changed. In version 3.0, 7 classes RenameXXXRefactoring were renamed to RenameXXXProcessor, which corresponds to the introduction of the new concept of processor-based refactoring. In version 3.0, package org.eclipse.jdt.internal.ui.text.template was renamed to org.eclipse.jdt.internal.ui.text.template.preferences since it holds only the preferences-related classes in 3.0 and its two classes related to content-assist features were extracted to a newly created package, named org.eclipse.jdt.internal.ui.text.template.contentassist.

The member class ProjectCache was created to encapsulate two fields allPkgFragmentRootsCache and allPkgFragmentsCache, whose role was replaced by a field of type ProjectCache. Finally, as an example of backward-compatibility renaming, consider class ASTRewrite, which was renamed to OldASTRewrite that delegates to the new (with totally different implementation) ASTRewrite in a new package.

The question then becomes: "how easy is to modify the clients of the renamed entities if they are carried over to the new Eclipse version?"

References to entities with modified identifiers can be automatically updated with little cost by parsing the source code and scan-

[1] The experiment is still in progress.

ning the abstract syntax tree (AST). The case of method signature changes (including parameter-list changes) is a bit more intricate. The parameter lists of about 16% (654/4170) renamed methods were changed in some combination of the following three types: (a) a parameter type was renamed or moved; (b) the parameter order was changed; (c) a parameter was removed. The combination of these three types of parameter list changes cannot be handled as easily as simple identifier changes and would require special support by the refactoring IDE and corresponding refactoring-migration tool (see section 6.4).

A small fraction (less than 2% of 4170) of renamed methods changed the parameter type to its supertype to make the method more general. Such changes would be transparent to client code.

In about 58% of method renamings, the parameter list was extended with at least one additional parameter. In 33% (1352/4170) of the cases, there was only newly added parameter(s) without removed parameter(s) (but may have other types of parameter list changes listed above). Such types of changes often indicate that the method delivers some additional functionality by making use of the additional parameter(s). In the case of constructors, additional parameter(s) are frequently used to initialize corresponding newly added field(s). Since additional parameter(s) most commonly indicate new behavior, these renamings are in effect non-behavior-preserving and should not be considered as refactorings.

Finally, in 25% (1110/4170) of method-renaming cases, the new parameter lists included newly added parameters as well as removed ones. In some cases, a parameter is replaced by several others. For example, the method javadocDuplicatedParamTag(JavadocSingleNameReference) used only a few pieces of information from its parameter object. It was subsequently renamed to javadocDuplicatedParamTag(char[],int,int) in version 3.1 to take in just-enough information as parameters, since it was not concerned with the whole JavadocSingleNameReference object. There are also cases, such as 6 methods defined in interface ISourceElementRequestor, where several parameters were replaced by a single parameter, which may be the result of the "Introduce Parameter Object" refactoring [6].

Finally, there were several types of parameter-type changes, such as replacing a boolean type with an int or long flags, replacing a primitive type with an object type (e.g. int with Integer), replacing a type with a collection of that type. In order to regard these changes as automated refactorings, one would need to invoke a proper wrapper, such as [1], for the relevant parameters and would also need to know how to access the member from the wrapper. However, the relevant methods and classes most likely exhibit other substantial changes, which cannot be expressed in terms of refactorings and would require that the developers of client applications manually modify their software.

5.1.2 Program-entity moves

There are 2315 move instances of various types of program entities (see Table 5). We identified several kinds of moves with different underlying motivations:

1) *Reorganizing or redistributing the information among different parts of a software system;*
2) *Moving responsibilities to eliminate Law-of-Demeter violations;*
3) *Maintaining backward compatibility with earlier versions;*
4) *Deprecation plus delegation.*

For example, in version 3.0, three packages were moved to the new source folder of jdt.launching plugin; one package org.eclipse.jdt.internal.junit.runner was moved from the jdt.junit plugin to the newly added plugin jdt.junit.runtime. In version 3.0, the abstract class SearchPattern was moved from org.eclipse.jdt.interal.core.search to org.eclipse.jdt.core.search to replace the role of the deprecated interface org.eclipse.jdt.core.search.ISearchPattern. In version 2.1.3, there were three Util classes scattered in three different packages of the jdt.core plugin; some of the features they provide were duplicate; in 3.0, their features were moved (merged) into a single Util class. The overall intention of these moves is to reshape the software system so that it is easier to understand and maintain.

According to the "Law of Demeter" [13] – "only talk to your friends" - methods of a class should only manipulate the class' own fields and should call methods defined in the class or the classes whose instances it contains. It is essentially an object-oriented formulation of the general "low coupling" software-engineering principle. Often, moves aim at refactoring entity responsibilities so that this law is not violated. For example, in version 2.0, JavaBasePreferencePage used to declare a public static method doubleClickGoesInto(), which was only called by PackageExplorerActionGroup.handleDoubleClick(); in 3.0, this method was moved to PackageExplorerActionGroup and it was made private and no longer declared static. Such moves often involve the fields and methods defined in one class but are mostly used in other classes, which is the exact intention of "Move Field/Method" [6] as described in Fowler's refactoring catalog. They enhance encapsulation and reduce coupling.

Similarly to renamings, some moves aim at maintaining backward compatibility. For example, when evolving to version 3.0, the class TextChange was redeveloped. In order to maintain the backward compatibility, two of its public methods were moved to a new class TextChangeCompatibility and were declared as static; they were also given one more parameter of the type TextChange to which they delegate their implementation.

In some cases, the "old" entity is replaced by the "new" entity but the "old" is not removed, instead it simply delegates to the "new" entity that now implements its logic. For example, in version 3.1, a new class BasicSearchEngine was extracted from SearchEngine, which was deprecated; 13 fields and methods of SearchEngine were moved to BasicSearchEngine; for the remaining 12 public methods, 12 corresponding same-signature methods were declared in BasicSearchEngine that implement the same logic as their counterparts in SearchEngine, and SearchEngine simply delegates to BasicSearchEngine for its functionalities.

Finally, in some cases, moves are the integral part of "bigger" refactorings. For example, 60 (about 50% of 121) class and interface moves are part of 16 "Extract Package" refactorings (see section 5.2.1).

Let us now consider again the issue of the support required to carry the clients of the moved entities over to the new version.

In principle, all types of program entities can be moved. About 76% (1759/2315) of entities were moved with no other changes made to them. This is not surprising since the general intention of moving program entities is just to redistribute features in order to enhance encapsulation, understandability and maintainability, instead of modifying entities for other purposes. These entity moves represent true behavior-preserving refactorings and the references to them can be automatically updated (the information about the context of moved entities may be needed).

However, sometimes, moved methods also experienced changes to their parameter lists. They may take the "old" home class as an additional parameter, such as TextChangeCompatibility described above. More frequently, moved entities also undergo modifier and visibility changes. About 18% (416/2315) of moved entities had

their declared modifiers and/or visibility levels changed (such as PackageExplorerActionGroup and TextChangeCompatibility discussed above). In 107 cases, their static and/or final status was toggled. In 331 cases, the visibility was modified. As discussed in sections 5.1.3 and 5.1.4, many of the modifier and visibility changes can be easily wrapped. Finally, less than 7% of total 2315 moved entities came with other changes, such as data (return) type change, class inheritance and/or interface implementation change. By closer inspection, they are just a sequence of separate (not inherently related) changes applied to the same program entity, which often require that the developers of client applications manually update their software.

5.1.3 Modifier changes

There were 1088 modifier changes made to 1064 program entities (including newly added and removed modifiers). About 50% of these changes should not cause compilation problem or could be easily wrapped.

Java synchronization operations may incur significant performance overhead, which might affect the applications' performance and behavior. However, an entity newly declared as synchronized will not cause a compilation failure in its client application. For entities that changed from being synchronized to not being synchronized, an escape analysis [3] can be applied to determine where it is safe to replace a synchronized object with an unsynchronized one. In cases where synchronization cannot be safely removed, a synchronization wrapper (similar to the Java standard library class java.util.Collections, which is an instance of the Decorator pattern [7]) can be inserted around the object, which delegates to the given object, but makes the forwarding method synchronized.

An entity newly declared as final may or may not break client applications, depending on whether the application assigns, overrides or extends the changed program entity. Entities that used to be, but are no longer, declared as final should not cause compilation failures to the client application.

Fields (methods) newly declared as static may cause compiler warnings such as "The static entity should be accessed in a static way", but should not cause any problems on client code. For those fields (methods) that are no longer declared as static, a factory method that returns an instance of the declaring class may be inserted; the returned instance can then be used to call the corresponding instance entities.

It is interesting to note that about 34% modifier changes were made to a very small fraction (less than 4%) of 1064 entities that had modifier changes. For example, in version 2.1, 26 public fields of JavadocOptionsManager were no longer declared with static; in version 3.0, 25 methods of the class DefaultBindingResolver were newly declared with synchronized.

5.1.4 Visibility changes

In our analysis, we found 1842 program entities that changed their visibility: of them, 1091 changed to a less restrictive visibility level and 751 changed to a more restrictive one.

Object-oriented languages provide explicit support for defining the scope of the various design elements of a system. Frequently, developers make elements "too accessible" in the beginning. As the picture of the scope of the valid clients of each element becomes clearer, the element visibility may be restricted. For example, in about 24% (441/1842) of the visibility-change cases, an entity was made private: in about 70% of these cases (299/441) there was no incoming usage from outside their corresponding declaring classes. Most of the others gradually became used only inside their declaring classes and were finally made private in a subsequent release.

For those entities whose visibility is decreased, the changes may be safe within the component (such as Eclipse) itself. However, the client applications that depend on those entities may break as a result. In that case, a wrapper (similar to the effect of "Encapsulate Field" refactoring) may be used to provide the access to more restrictive entities.

About 15% (331/2315) of moved entities also changed their visibility. When entities are pulled up to a superclass or moved to helper or delegate classes, their visibility often changes to a less restrictive level in order to allow the subclasses or the original class to access them. When entities are pushed down to a subclass or moved closer to where they actually get used, their visibility often decreases since they can be accessed within the declaring scope of the current class. Sometimes, when converting nested types to top-level, their visibility may increase; on the other hand, when top-level types are converted to nested types, their visibility often decreases.

Similarly to modifier changes, about 30% of visibility changes were made to a very small fraction (less than 3%) of the 751 entities that changed their visibility to a more restrictive one.

5.1.5 Data-type changes

We found 1524 data-type changes (including field data-type and method return-type). 7% of them (107/1524) were generalizations to a supertype and 6% (85/1524) were specializations to a subtype.

The clients that are now forced to use a supertype may fail to compile successfully, depending on whether they access the members that are not visible through the supertype's interface. In such cases, an explicit downcast that wraps the changed field (method) may be necessary.

Specializations to subtypes may be the result of the "Encapsulate Downcast" refactoring [6]: for example, when evolving to version 3.1 the return type of CompilationUnitRewrite.createChange() was changed from TextChange to CompilationUnitChange (but createChange() returns an instance of CompilationUnitChange in both versions 3.0.2 and 3.1). Although using a subtype will not cause a compilation failure to the client code, it may behave differently. For example, the data type of ExceptionBreakpointFilterEditor.fFilterViewer was changed from TableViewer to CheckBoxTableViewer. But since it is initialized with TableViewer and CheckBoxTableViewer in 2.0 and 2.1 respectively, the client gets a table viewer with check boxes instead of a plain table viewer in 2.1.

There were about 20% (302/1524) of date-type changes that might be wrapped, such as String <=> StringBuffer, type <=> collection or array of type, Vector <=> List, etc. A refactoring tool can swap the corresponding types if there is a specification, such as [1], that can be used for guiding the migration. Furthermore, about 9% (129/1524) of data-type changes involved the change of method return-type from void to some type. The clients of these methods can simply ignore the returned object. Finally, the remaining 60% (901/1524) of data-type changes were too radical to be considered as refactorings. For example, the return type of method getChangedClassFiles() was changed from List to ChangedClassFilesVisitor. The field binaryPath of type String was renamed to binaryFolder of type IContainer.

5.1.6 Generalization and abstraction changes

The class hierarchy of Eclipse is relatively stable. There were 304 instances of class-inheritance changes in total. 72 XXXMessages classes started extending org.eclipse.osgi.NLS in version 3.1, and 34 dialog classes changed their superclass to org.eclipse.jface.dialogs.StatusDialog since the duplicate StatusDialog scattered in several plugins were finally removed in version 3.1. Among the remaining 198 changes, 90 classes changed to extend a

subclass of the one previously extended (76 are newly introduced suclasses); 32 classes changed to extend the superclass of the one previously extended (16 are because the classes previously extended were removed or inlined in the new version).

There are 466 instances of classes newly implementing an interface. In 186 of these cases, the interfaces in question were newly introduced. There were 389 cases where a type was changed not to implement an interface any longer (in 203 among them, the interface in question was also removed).

Although there are some inheritance hierarchy changes resulting from such refactorings as "Extract Superclass", "Inline Superclass", or "Extract Interface" (see section 5.2), most of inheritance-hierarchy changes bring about behavior modifications: the client application may compile fine with the new version, however, it may behave differently.

5.1.7 Program-entity additions and removals

Table 3 summarizes the newly added and removed public and protected program entities between compared versions. Clearly, Eclipse grew fast in the past four years. Compared with the corresponding previous versions, the versions 2.1, 3.0, and 3.1 contain 7127, 14095, and 17343 newly introduced packages, classes and interfaces, fields and methods (including constructor), respectively. In the mean time, a certain amount of public or protected program entities (much less than the newly introduced entities) were removed, 1298, 4157, and 2455 for version 2.1, 3.0, and 3.1 respectively. The removed public or protected program entities may cause the application to fail to compile. A small fraction of newly added or removed program entities are the results of various "Extract…" or "Inline…" refactorings, as discussed in section 5.2. But most of these changes represent newly introduced API or removed obsolete API.

5.2 "Bigger" refactorings

In this section, we discuss "bigger" refactorings, which are composed of a coherent series of elementary structural changes to a set of related entities. Although, in principle, refactorings should be performed one step at a time, Fowler [6] and Kerievsky [12] demonstrate how a series of "small" refactorings can lead to the "big" changes, such as the introduction of design pattern. By looking at a set of changes as a coherent whole, we may gain a better understanding of the design evolution of a software system and the refactorings it has suffered, and consequently be in a better position to assess the state-of-the-art in tool support for the practice.

The refactoring support that Eclipse provides is representative of the state-of-the-art today. We reviewed the currently available refactoring tools and IDEs (www.refactoring.com/tools.html) and Eclipse supports a superset of the refactorings supported by each of them. The only interesting exception is IntelliJ IDEA, which supports two more types of refactorings than Eclipse does, "Extract Super/Subclass" and "Replace Inheritance with Delegation".

5.2.1 Containment-hierarchy refactoring

Large software projects are often organized in terms of subsystems, packages, and (nested) reference types; such organization makes the dependencies among the various components explicit and makes it easier to identify the use of a component by its implied container. The developers often restructure the containment hierarchy at different levels.

The Eclipse plugins work as subsystems that contribute different features to the platform. A new plugin may be introduced as the appropriate placeholder for features that were originally placed in other plugins. In version 3.0, three new plugins, jdt.junit.runtime, ltk.core.refactoring and ltk.ui.refactoring, were split from two existing plugins, jdt.junit and jdt.ui (the "core refactorings" and "ui refactorings" folders) respectively; several packages were either moved or extracted into the new plugins.

Type of refactoring	# detected	Eclipse support
Convert anonymous class to nested*[2]	12	√
Convert nested type to top-level	19	√
Convert top-level type to nested	20	×
Move member class to another class	29	√
Extract package	16	×
Inline package	3	×

Package is one way of grouping together related classes depending on their behavioral dependencies. When a package has too many classes to be easily understandable and is not cohesive because these classes are responsible for very different features, a new package may be extracted to hold some important groups of classes. For example, org.eclipse.jdt.internal.ui.refactoring.reorg was extracted from org.eclipse.jdt.internal.ui.refactoring in the same plugin, and org.eclipse.jdt.internal.formatter.comment in jdt.core was extracted from org.eclipse.jdt.internal.ui.text.comment in the jdt.ui plugin. Other times, a package is removed and its contents may be inlined to other package(s). For example, three classes of the removed package org.eclipse.jdt.internal.corext.template were inlined to org.eclipse.jdt.internal.corext.template.java package.

Java classes and interfaces can define their own nested types. Sometimes, the top-level types may be converted to nested type of a particular class in order to group together the relevant classes and make the dependencies among them clear. On the other hand, nested types may be converted to top-level so that they are available to other classes. In Java, anonymous classes are widely used to avoid creating a bunch of simple subclasses or implementations of interfaces. However, when the anonymous classes grow so large that the code becomes difficult to read or maintain, they may be converted to nested type.

All these changes can be accomplished by various types of refactorings: convert anonymous class to nested, convert nested (top-level) type to top-level (nested), move member class, and extract or inline package. Three of them are supported in modern IDEs, such as Eclipse, while the other three are not explicitly supported.

5.2.2 Inheritance-hierarchy refactoring

Type of refactoring	# detected	Eclipse support
Pull up field/method	279	√
Push down field/method	53	√
Extract interface	28	√
Extract superclass	15	×
Extract subclass	4	×
Inline superclass	4	×
Inline subclass	7	×

Programming to interfaces and not to implementations is an important tenet of object-oriented development [7]. When the client is implemented to be agnostic of the internal implementation of the server class, assuming only the specification of its public behavior interface, the server retains the flexibility to evolve. As long as the public interface remains the same, modifications to its implementation will not break its clients. A corollary of the programming-to-

[2] The * indicates that the results for the particular row are partial and the analysis is still in progress.

interfaces principle is the "Extract Interface" refactoring. For example, in version 3.1, a new interface IChangeAdder was introduced for class JUnitRenameParticipant and its two subclasses ProjectRenameParticipant and TypeRenameParticipant.

When two (or more) classes share a substantial part of their behaviors, their common features may be extracted to a superclass. For example, in version 3.1, a superclass HierarchyRefactoring was extracted (involving 57 field and method pull-ups) from PullUpRefactoring and PushDownRefactoring. When a class defines features that are only applicable in some cases, a subclass may be extracted for that subset of features. For example, a subclass ImportMatchLocatorParser was extracted from MatchLocatorParser, which holds two methods that are used only for compilation unit.

Collapsing hierarchies is another important refactoring that deals with generalization. When a superclass does not deliver much functionality or a subclass is not that different from its superclass, the two may be merged. For example, in version 2.1, the superclass BufWriter was inlined into subclass VerboseWriter; in version 3.0, three subclasses MemberTypeDeclaration, LocalTypeDeclaration, and AnonymousLocalTypeDeclaration were inlined into their superclass TypeDeclartion.

Finally, within the inheritance hierarchy, common fields and methods of subclasses were pulled up to the superclass, while the fields and methods that were only applicable to some subclasses were pushed down to them.

5.2.3 Class-relationship refactoring

Type of refactoring	# detected	Eclipse support
Extract constant interface	5	√
Inline constant interface	2	×
Extract class	95	×
Inline class	31	×

Object-oriented systems are designed around classes that model abstractions of real-world entities and/or encapsulations of a coherent set of behaviors. Classes collaborate with each other to deliver the application functionalities.

In Java, interfaces are often used to define static final constants; the classes may implement them to access the constants or access them in the static way. For example, in version 2.0, class JavaPartitionScanner and FastJavaPartitionScanner used to define four same constants, which were extracted to a new interface IJavaPartitions implemented by the two classes in subsequent release 2.1. This refactoring also removed the duplication. When the constants are only used by a single class and its subclasses, the interface may be inlined. For example, in version 3.1, the constant interface BindingIds was removed and the constants it defined were inlined to the class Binding.

Complex classes are sometimes incohesive because they are responsible for delivering many responsibilities. Such classes should be simplified by extracting some of their features into other classes, created for exactly that purpose. The simplified class can then delegate to the newly created class to deliver its responsibilities. For example, in version 3.0, a new class DeltaProcessingState was extracted from DeltaProcessor; DeltaProcessor newly declared a field of type DeltaProcessingState, to which it delegates the maintenance of the global state of delta processing.

Another frequent case involves the extraction of helper or utility class. For example, the helper class RefactoringExecutionStarter was extracted from ReorgMoveAction in version 3.1.

When a class does not have many responsibilities, its features may be inlined. For example, class ReferenceScopeFactory that used to define a single public method creating an instance of IJavaSearchScope was inlined to JavaSearchScopeFactory in version 3.1. Sometimes, the helper class may be inlined to the class depending on it. For example, SuperReferenceFinder was inlined into PullUpRefactoring.

Developers often introduce new entities before they realize that similar features already exist. In such cases, the inline-class refactoring can be used to remove duplication. For example, in version 2.1.3, there were three Util classes scattered in three packages; in version 3.0, they were inlined into a single Util class.

5.2.4 Internal class refactoring

Type of refactoring	# detected	Eclipse support
Information hiding	751	×
Generalize type	107	√
Downcast type	85	×
Introduce factory	19	√
Change method signature	4497	√
Introduce parameter object*	4	×
Extract method*	45	√
Inline Method*	31	√

Eclipse supports various types of refactorings that reorganize the code within a class, including, generalize type, introduce factory, change method signature, and extract or inline method. We identified a large number of such refactorings in Eclipse's evolution history. However, Eclipse does not support the refactorings of information hiding, downcast type, introduce parameter object, which also often being applied.

On the other hand, Eclipse supports several refactorings that change the code within a method, such as extract local variable, extract constant, convert local variable to field. However, at the current stage, JDEvAn does not take into account the statement-level information so that it does not support the analysis on these statement-level refactorings.

5.3 Structural-change sequences

Finally, let us look at the program entities that undergo two or more types of changes. 27% (2104/7851) of the modified entities underwent two or more types of changes. We have already discussed several such cases in section 5.1, including renaming program entities to reflect their data-type change; renaming to conform to a naming convention for static final fields; moving methods and using an additional parameter of the type of the "old" home class; moving program entities and changing their visibility correspondingly.

By closer inspection, we noticed that in most cases, subsequent modifications to an entity were separate and not inherently related, such as the "bigger" refactorings discussed in section 5.2. For example, in version 3.1, ASTParser.convert() was moved to class CompilationUnitResolver; its visibility changed from private to public; it was newly declared with static; its return type was downcast from ASTNode to CompilationUnit; and its signature was modified to take three more parameters as input.

6 Analysis of the case-study findings

Conducting this comprehensive study has given us some interesting insights into the structural evolution of object-oriented software system. We discuss them in this section.

6.1 Refactoring is a frequent practice

Refactoring is advocated as a good practice for improving the design of software through local behavior-preserving code restructur-

ings [6]. In recent years, refactoring has been popularized in object-oriented software development, especially in the context of agile, lightweight development processes such as "extreme programming" [2]. However, it is not clear how prevalent refactoring is actually in practice. In our Eclipse-evolution case study, there were 58973 (see Table 3) changes reported by *UMLDiff*. Most of the radical design and implementation changes were made in the major releases 2.1, 3.0, and 3.1. Many new features were introduced, and many existing features were redeveloped with a totally different implementation, such as the AST-rewrite feature. We excluded from our analysis about 75% of all the changes that, according to our understanding – based on code inspection, the *UMLDiff* result, the help document, and the javadoc comments – represent the introduction of new features or the removal of obsolete API.

When considering the remaining changes, about 70% of them were the results of refactoring or a sequence of refactorings, including renamings, moves, downcasting or generalizing type, information hiding, reorganizing containment or inheritance hierarchy, changing the relationships among classes, and changing the code within a class, as disused in section 5. Overall, about 16% of all the changes (including adding and removing) can be expressed in terms of "standard" refactorings [6], which we believe is an indicator that a considerable amount of effort has been spent on intentionally restructuring the existing system in the evolution of Eclipse.

This is evidence that a refactoring engine would be a valuable functionality for the development environment in order to provide (semi-)automatic refactoring support to developers instead of them having to perform refactorings manually.

6.2 Many structural changes can be collected and updated automatically

Eclipse is built as a plugin-based framework. It is an IDE as well as a software development kit (SDK). The developers can build their own plugins by extending the existing ones and then integrate them into Eclipse. The JDEvAn tool we used in this case study is one of such plugins we have developed in our research group. Even for such a small-size research prototype, we have suffered from breaking API changes as the underlying Eclipse platform evolved.

In the last section, we discussed that about 70% of structural changes can be expressed in terms of refactorings from the perspective of the Eclipse framework developers. To them, a refactoring, such as move method, affects only the structure of the software and not its behavior. However, it is simply impossible for Eclipse developer to update all the third-party plugins built on it when they refactor the code. Thus, to third-party plugin (framework-based application) developers, such a refactoring may be a breaking change, which indicates that they have to migrate their code to the new version of Eclipse. Such migration is often perceived as disturbing.

However, our case study shows that for about 60% of structural changes that can be expressed in terms of refactorings, the references to the affected entities in client applications can be automatically updated by a refactoring-migration tool if the relevant information of the refactored components can be gathered through the refactoring engine. This indicates that a refactoring-based development environment can benefit a lot from refactoring-migration tools, such as CatchUp [9]. However, the refactorings that CatchUp can record and replay are only renamings and moves. These account for about 70% of the tedious updating tasks that may be handled automatically for applications that use the refactored components. However, there exist several other frequently used low-level refactorings, such as "information hiding", "downcast type", which CatchUp do not support. Furthermore, the current refactoring-migration tools are unaware of the impact of higher-level refactorings, such as inheritance-hierarchy refactorings.

6.3 Support is still missing for higher-level refactorings

Modern IDEs, such as Eclipse, support the most commonly used, low-level refactorings, including renaming, move generalize type. But they do not support "downcast type" and "information hiding" refactorings, which our case study shows are also frequently applied. Especially for the "information hiding" changes, we found out (see section 5.1.4) that a class may have several members to hide; manually hiding all of them could be error-prone.

Eclipse supports moving static fields and methods to a specified type, but it treats moving instance fields simply as a textual move and the references to the moved instance fields will not be updated. Furthermore, Eclipse only supports moving instance methods to types of its parameters or types of fields declared in the same class as the method. The Eclipse "pull up" and "push down" refactorings support moving instant fields and methods to their direct superclass or subclass. However, in our case study, instance fields and methods may be moved to any type, which may or may not be directly related to their current declaring class.

Eclipse supports some of the "bigger" refactorings discussed in section 5.2, but it lacks support for the refactoring of the containment and inheritance hierarchies and general class relationships. Although one may still achieve the same results by applying a set of small, primitive refactorings, we believe that, by combining the relevant low-level changes into composite high-level refactorings, it becomes more efficient to convey and implement the specific intent of the change. Suppose, for example, that we want to extract a helper class C that contains an instance method M declared in D. With current tool support, the developer may perform the following activities: create a new class C; declare a new field F of C in class D; move M to C and then remove the field F. It seems that copy and paste would be an easier solution. However, as summarized in [10], about 22% of the copies the developer leaves off-screen references unchanged or only copies part of the code being distributed within several files.

Based on our findings of the refactorings actually applied to Eclipse throughout its evolution history, an effective refactoring tool should support the following (in addition to what are commonly supported in current IDEs):

- information hiding refactoring, such as "hide a group of method in a class",
- more flexible move of instance field and method in terms of object-oriented entity instead of simply text;
- a refactoring user interface to collect the information about more complex refactoring tasks, such as those refactoring inheritance-hierarchy.

6.4 Tools should implement refactorings using the command and composite patterns

The question then becomes: "What might an appropriate internal representation for refactorings be, such that it would enable a tool to meet the above requirements?"

As discussed in section 5.3, about 27% of all the program entities that have been modified underwent two or more types of changes, which can be any combination of the elementary structural changes discussed in section 5.1. Furthermore, there were about 370 "bigger" refactorings that have been applied to refactor the containment, inheritance, class relationships and class internals. These

"bigger" refactorings are composed of a series of coherent related structural changes to a set of relevant entities.

These facts imply that a good possible implementation of an automated refactoring functionality would be to view a structural change as a command object: thus, simple refactoring commands could be composed into larger ones [7] and they could also be done, undone and replayed. For example, 264 methods (see Table 4) moved methods change their identifiers and/or parameter lists as well. Suppose that a method is moved and then one of its parameters is removed. These changes can be stringed together as a MoveMethodCommand followed a RemoveParameterCommand, which are contained in a CompositeCommand. A memento object [7] may be used to record which parameter is removed. As another example, consider the extract superclass refactoring: it can be an instance of CompositeCommand, composed of a NewClassCommand, several ModifyClassInheritanceCommands, and several PullUpCommands. A PullUpCommand can further be an instance of CompositeCommand, which may be composed of a MoveFieldCommand and a ModifyVisibilityCommand.

The other benefit with command objects is that they can be executed at different times [7]. The refactoring tool can record the command objects and replay them (if possible) on the applications that reuse the refactored components. Such "refactoring deferral" would effectively constitute refactoring migration.

7 Conclusions

Refactoring is an activity crucial for evolutionary-development processes. The basic idea is that the design can become more cohesive, less coupled and therefore easier to read and maintain through local code restructurings. Several IDEs support some types of refactorings, usually the simpler ones and there has been some initial research as to how API-breaking refactorings can be migrated to client applications. The objective of our case study has been to (a) examine the actual refactoring practice in the context of a realistic framework with substantial evolution history and many client applications and (b) to come up with some requirements and design suggestions for tools purported to support the practice.

Although, we cannot argue that Eclipse is a typical software project – in fact it is difficult to characterize the properties that a typical project should have – it is certainly a software framework of realistic size and interesting evolution history and that makes it an appropriate test-bed for evaluating our method.

We examined three pairs of subsequent major Eclipse releases and we discovered that indeed refactoring is a frequent practice and it involves a variety of restructuring types, ranging from simple element renamings and moves to substantial reorganizations of the containment and inheritance hierarchies. Although many of them are behavior preserving from the point of view of Eclipse – as advocated – they may still affect the behavior of the client applications. To support the developers of these applications to carry them over to the next release of the API, a tool should be able to treat refactorings as composite commands possibly consisting of a set of other refactorings. Each such refactoring command should remember all its effects to the framework and should be able to replay them and also propagate them in the context of the application. Current refactoring-tool support falls short on the compositional requirement and refactoring-migration tools are also limited in that they are not aware of the whole impact of complex refactorings. A design-differencing capability, such as JDEvAn's, could potentially be a helpful utility in both contexts: it could recognize related changes when the refactoring is not applied explicitly through using the refactoring tool which could then be replayed by the refactoring-migration tool.

References

1. I. Balaban, F. Tip and R. Fuhrer. Refactoring support for class library migration. *Proceedings of the 20th OOPSLA*, pp. 265-279, 2005.
2. K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.
3. J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar and S. Midkiff. Escape analysis for Java. *Proceedings of OOPSLA*, pp. 1-19, 1999.
4. S. Demeyer, S. Ducasse and O. Nierstrasz. Finding refactorings via change metrics. *ACM SIGPLAN notices*, 35, 10 (2000), 166-177.
5. D. Dig and R. Johnson. The role of refactoring in API evolution. *Proceedings of ICSM*, 2005.
6. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
7. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31, 2 (2005), 166-181.
9. J. Henkel and A. Diwan. CatchUp! Capturing and replaying refactorings to support API evolution. *Proceedings of the 27th ICSE*, pp. 274-283, 2005.
10. A.J. Ko, H.H. Aung and B.A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. *Proceedings of 27th ICSE*, pp. 126-135, 2005.
11. P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software,* 12, 6 (1995), 42-50.
12. J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.
13. K.J. Lieberherr and I. Holland. Formulations and Benefits of the Law of Demeter. SIGPLAN Notices 24(3):67-78, March 1989.
14. *OMG Unified Modeling Language Specification*, formal/03-03-01, Version 1.5, (2003), http://www.omg.org.
15. W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.
16. D. Roberts, J. Brant and R.E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object System* 3, 4 (1997), 253–263.
17. F.V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. *Proceedings International Workshop on Principles of software Evolution*, pp. 126–130, September 2003.
18. Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. *Proceedings of the 20th International Conference on Automated Software Engineering,* 2005.
19. Eclipse, http://www.eclipse.org

Table 1. The number of program entities

	2.0	2.1	2.1.3	3.0	3.0.2	3.1	Total
Package	138	144	144	177	177	188	968
Class	3546	4326	4332	5610	5612	6466	29892
Interface	692	768	769	935	935	1024	5123
Array Type	562	294	296	383	383	439	2357
Field	11440	14213	14245	18812	18862	29029	106601
Method	27623	33829	33878	42923	42927	49187	230367
Constructor	3929	4737	4751	6025	6027	6943	32412
Total	47930	58311	58415	74865	74923	93276	**407720**

Table 2. The number of entity relations

	2.0	2.1	2.1.3	3.0	3.0.2	3.1	Total
Contains	53623	65925	66034	84963	85076	105162	460783
Extends	3253	4003	4009	5134	5135	5921	27455
Implements	1449	1790	1792	2298	2300	2596	12225
Reads	44583	54842	54954	73754	73827	98120	400080
Writes	17781	21597	21638	27755	27815	32648	149234
Calls	90924	117813	117815	151629	151858	179775	809814
Class usage	31658	39284	39362	51700	51830	61594	275428
Class instantiation	9915	12273	12315	16025	16123	19037	85688
Total	253186	317527	317919	413258	413964	504853	**2220707**

Table 3. Eclipse's changes

Type of change	2.1 – 2.0	3.0 – 2.1.3	3.1-3.0.2	Total
Entity renaming	809	2285	1488	4582
Entity move[#3]	387	1244	684	2315
Visibility change	435	857	550	1842
Data (return) type change	245	718	561	1524
Non-access modifier change	167	484	425	1076
Interface implementation change	190	391	274	855
Class inheritance change	33	109	162	304
Entity addition	7127	14095	17343	38565
Entity removal	1298	4157	2455	7910
Total	10691	24340	23942	**58973**

Table 4. Rename program entities

	2.1 – 2.0	3.0 – 2.1.3	3.1 – 3.0.2	Total
Rename package	1	1	0	2
Rename class[#]	20	47 + *2*	24	91 + *2*
Rename interface[#]	2	1 + *1*	0	3 + *1*
Rename field[#]	107 + *5*	274 + *8*	199 + *29*	580 + *42*
Rename method[#]	559 + *19*	1647 + *162*	1042 + *83*	3248 + *264*
Rename constructor	120	315	223	658
Total	833	2455	1600	**4891**

Table 5. Move program entities

	2.1 – 2.0	3.0 – 2.1.3	3.1 – 3.0.2	Total
Move package	0	4	0	4
Move class	18	62	31	111
Move interface	1	6	3	10
Move field	172	318	331	808
Move method	196	854	319	1369
Total	387	1244	684	**2315**

[3] The [#] indicates that the moved entities may also have identifier and/or parameter list changes (shown in italic font in Table 4)