Refactoring, Way for Software Maintenance

Madhulika Arora[1], Dr. S. S. Sarangdevot[2], Vikram Singh Rathore[3], Jitendra Deegwal[4], and Sonia Arora[5]

[1] PhD Scholar,Gyan Vihar University,
Jaipur (Raj)302025, INDIA

[2] Director of IT & CS
Department, Janardan Rai Nagar Rajasthan Vidyapeeth University,
Udaipur (Raj)331401, INDIA

[3] PhD Scolar, University of Rajasthan,
Jaipur (Raj)302004, INDIA

[4] PhD Scholar, Gyan Vihar University,
Jaipur (Raj)302025, INDIA

[5] Lecturer, Phonics Engineering College
Imlikehra Rorkee, Uttrakhand- 247667 INDIA

Abstract
Now days, most object-oriented software systems are developed using an evolutionary process model. In evolutionary development lifecycle, it needs to change from time to time. An important kind of change to object-oriented software is Refactoring. The motive 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.

Keywords: *Refactoring, Software Maintenance, Eclipse, Quality of Software.*

1. Introduction

Quality of Software can be improved by Good modularity. It facilitates extensibility and evolution, independent development of components, improves comprehensibility, eases verification. The value of modularity is even quantifiable. Developing software requires working with a number of *concerns*, or considerations a developer might have about the implementation of a software system. If any software is enriched with good modularity, each concern is implemented in only one module, and each module implements only a small number of concerns. This type of structure helps the developer manage complexity. To deal with any one concern they only have to look at one module, and to understand any one module they only have to think about a small number of concerns.

Any useful software system requires constant evolution and change. Often those changes require that the software be re-modularized, so that the system becomes easier to understand, extend, or maintain. For this type of need, researchers and developers have developed the practice of *refactoring*. As described in, refactoring are parameterized transformations of a system's source code intended to improve a system's structure with regards to informally expressed goals, such as maintainability, changeability, readability, performance, or memory demands. Traditional refactoring are generally behavior preserving. Modern software development environments include built-in support for semi-automated refactoring.

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

Refactoring activities are more challenging when we talk about reuse-based development. Software reuse simplifies the design of new systems but, at the same time, 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, 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.

As the underlying component framework is adopted, the cost of breaking client code becomes higher, which is why components 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 refactoring and of what type. This may be because there has been no substantial tool support for detecting and classifying structural evolution which is coming from refactoring 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 type of support should modern IDEs provide and how might this support be implemented?
- What proportion of the structural changes in the evolution of a system are the results of refactoring?
- What are the typical refactoring 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?

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.

2. Related Work

Griswold and Opdyke officially introduced the term refactoring virtually at the same time. Their work provides the theoretical basis for automated refactoring realized in many refactoring tools, most particularly the Smalltalk Refactoring Browser. Contemporary IDEs, such as Eclipse, typically offer some forms of refactoring support. Fowler popularized refactoring by providing a catalogue of refactoring.

As the main concentration is object oriented programming now a days so most refactoring research has targeted low-level program transformations in functional and object-oriented systems. Refactoring has recently become an integral part of the evolutionary software development methodology, such as "Extreme Programming". The books of Fowler and Kerievsky present a good general idea of the refactoring and how they can be used to carry out architectural and design changes. Opdyke's Ph.D. thesis catalogs a number of refactoring, 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 illustrate 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 refactoring. Demeyer et al define four heuristics based on the comparison of source-code metrics of two subsequent system snapshots to identify refactoring of three general categories. Rysselberghe investigated the use of clone-detection to identify move and renaming refactoring. Godfrey and Zou 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, that enables the identification of a rich set of elementary structural changes and fairly complex refactoring, which provides a firm base for us to study the structural evolution of a large object-oriented software project at fine-grained level.

Modern IDEs, such as Eclipse, offer automated support for most commonly used refactoring, such as rename or move. In this paper, we compare the refactoring that were in reality performed in the development of the object-oriented software system with those supported by IDEs, and extract that modern IDEs do not supply automated support for all frequently used refactoring, especially high-level refactoring, 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.
The results of our study confirm the usefulness of such migration tool support in the refactoring-based development environment beside the limitations of current tool support, especially the lack of support for high-level refactoring. Dig and Johnson also conducted a similar empirical study on the role of refactoring 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, which reports changes in much more detail than what is covered in the documentation. This enables us to understand the actual refactoring practice and draw out some high-level design requirements for refactoring-based development environments.

3. Eclipse

There are numerous reasons why we choose Eclipse as the subject for 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 refactoring. This was an especially interesting question; given the fact the Eclipse is an IDE that supports refactoring. 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 possible 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. There is 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.

4. Types of Refactoring

Refactoring, which can be viewed as series of elementary structural changes to a set of related entities, should be performed one step at a time, Fowler shows how a series of “small” refactoring 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 refactoring 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 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 refactoring.

4.1. CONTAINMENT-HIERARCHY REFACTORING

Projects are organized in terms of subsystems, packages, and 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 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 refactoring” and “ui refactoring” folders) respectively; several packages were either moved or extracted into the new plugins.
Classes can be grouped into Package, 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 refactoring: 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.

4.2. INHERITANCE - HIERARCHY REFACTORING

Programming to interfaces and not to implementations is an important tenet of object-oriented development. 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 consequence of the programming-to-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 carve up 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 Import- MatchLocatorParser 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.

4.3. CLASS-RELATIONSHIP REFACTORING

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

In Java, interfaces are used to define static final constants; the classes may apply 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 repetition. 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 detached and the constants it defined were inlined to the class Binding.

Complex classes are sometimes inconsistent because they are liable 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 pass on 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.

4.4. INTERNAL CLASS REFACTORING

Eclipse supports various types of refactoring that rearrange 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 refactoring in Eclipse's evolution history.

However, Eclipse does not support the refactoring of information hiding, downcast type, introduce parameter object, which also often being applied.

On the other hand, Eclipse supports several refactoring that change the code within a method, such as extract local variable, extract constant, convert local variable to field.

Eclipse is basically built as a plugin-based framework. It is an IDE as well as a software development kit. The developers can easily build their own plugins by extending the existing ones and then integrate them into Eclipse. About 70% of structural changes can be expressed in terms of refactoring from the perspective of the Eclipse framework developers, as we discussed in last section. 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.

About 60% of structural changes that can be expressed in terms of refactoring, the references to the affected entities in client applications can be automatically updated, if the relevant information of the refactored components can be gathered through the refactoring engine. This means that a refactoring-based development environment can benefit a lot from refactoring-migration tools, such as CatchUp . However, the refactoring 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 refactoring, 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 refactoring, such as inheritance- hierarchy refactoring.

5. Support is still missing for higher-level refactoring

Modern IDEs, such as Eclipse, support the most commonly used, low-level refactoring, including renaming, move generalize type.

But they do not support "downcast type" and "information hiding" refactoring, 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" refactoring 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" refactoring, 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 refactoring, we believe that, by combining the relevant low-level changes into composite high-level refactoring, 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, 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 refactoring 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. 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 refactoring, usually the simpler ones and there has been some initial research as to how API-breaking refactoring 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 refactoring as composite commands possibly consisting of a set of other refactoring. 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 refactoring. A design differencing capability 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.

7. References

[1] 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.

[2] Balaban, F. Tip and R. Fuhrer. Refactoring support for class library migration. *Proceedings of the 20th OOPSLA*, pp. 265-279, 2005.

[3] D. Dig and R. Johnson. The role of refactoring in API evolution.*Proceedings of ICSM*, 2005.

[4] D. Roberts, J. Brant and R.E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object System* 3, 4 (1997), 253–263.

[5] Eclipse, http://www.eclipse.org

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison- Wesley, 1994.

[7] 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.

[8] J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar and S. Midkiff. Escape analysis for Java. *Proceedings of OOPSLA*, pp. 1-19, 1999.

[9] J. Henkel and A. Diwan. CatchUp! Capturing and replaying refactoring to support API evolution. *Proceedings of the 27th ICSE*, pp. 274-283, 2005.

[10] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[11] K. Beck. *Extreme Programming Explained: Embrace Change.*Addison-Wesley, 1999.

[12] K.J. Lieberherr and I. Holland. Formulations and Benefits of the Law of Demeter. SIGPLAN Notices 24(3):67-78, March 1989.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[14] 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.

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

[16] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12, 6 (1995), 42-50.

[17] S. Demeyer, S. Ducasse and O. Nierstrasz. Finding refactoring via change metrics. *ACM SIGPLAN notices*, 35, 10 (2000),166-177.

[18] W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[19] Z. Xing and E. Stroulia. UMLDiff: An algorithm for objectoriented design differencing. *Proceedings of the 20th International Conference on Automated Software Engineering*, 2005.