Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 107 (2004) 57–69
www.elsevier.com/locate/entcs

Refactoring-aware versioning in Eclipse

Torbjörn Ekman[1] Ulf Asklund[2]

Deparment of Computer Science
Lund University
Lund, Sweden

Abstract

To fully support refactorings in a team development environment we have implemented a refactoring-aware repository provider as an extension plug-in to the Java Development Tools in Eclipse. The versioning system treats refactorings as first-class changes described as semantic actions rather than the set of resulting changes scattered over the source tree. We also introduce refactoring-aware merge, which merges refactorings as well as traditional changes utilizing the semantics of the refactorings to detect and resolve merge conflicts. It also ensures that the semantic meaning of a refactoring is preserved after the merge.

Keywords: Refactoring, Versioning, Merge, Eclipse, Software Configuration Management

1 Introduction

Refactorings are a safe and efficient way to improve code quality in a controlled manner. Refactorings differ from traditional changes in that they are guaranteed to be semantically valid and have a semantic meaning, e.g. after renaming a variable all uses of that particular variable should still reference the same variable.

Refactorings are safe in that a set of preconditions must be met prior to execution, ensuring that the required source code transformation is valid. Refactorings are efficient in that proper tools support enables the developer to apply a large set of related changes, scattered over the entire source tree, in one operation. As refactorings tend to be global in their nature, e.g. renaming

[1] Email: torbjorn.ekman@cs.lth.se
[2] Email: ulf.asklund@cs.lth.se

doi:10.1016/j.entcs.2004.02.048

a class may affect any other class as the renamed class is most often visible in the entire source tree, tool-support will increase the speed of these operations dramatically.

These benefits apply to refactorings in a single user environment, but the introduction of traditional team development repository providers decreases the above mentioned benefits due to insufficient tool support. When merging code from several developers the preconditions have been checked for each individual branch, but not for the merged source tree. Thus, all preconditions ensuring a safe refactoring may not be met for the merged tree. During merge, the changes originating from a refactoring are viewed as numerous small unrelated changes that the developer may have to merge manually, in case of a merge conflict. Since a refactoring is only applied to one of the merged branches, and not both involved versions, the merged result may not include the semantic meaning of the refactoring. Changes in the branch where the refactoring is not applied may violate pre-conditions, prohibiting the refactoring to execute, or newly introduced code that should have been affected by that refactoring is unaffected, e.g. adding a invocation of a method that is renamed in a parallel branch should be changed to use the new name instead of the old name.

We notice three major deficiencies in traditional tools providing versioning and merge support for team development:

- A developer can not view the differences between two versions of a file in terms of refactorings, but only as the resulting individual, unrelated changes. This because the semantics of a refactoring is lost when stored in the repository.
- The lack of support for refactoring-aware merge when using refactorings during team development, i.e. the semantic meaning of a refactoring applied to one branch can not easily be transfered to another branch during merge.
- The lack of traceability when source elements are affected by a refactoring, e.g. moved and renamed code blocks are not traceable to their origins in the version history, but seen as unrelated delete and add operations.

To better support refactorings in a team development environment, we have developed a model for a refactoring-aware repository provider. This provider stores and retrieves both traditional changes and entire refactorings, which enables the presentation of differences between two versions in terms of refactorings. Moreover, when two branches are merged this semantic knowledge is used to better detect potential conflicts and (when possible) to do an automatic merge. The result is that some cases of unintended merges and false merge conflicts now can be correctly automatically merged and that real

conflicts more often are detected and better presented to the developer. We have implemented parts of this model in a prototype as an extension plug-in to the Java Development Tools in Eclipse. Currently the prototype supports refactoring-aware automatic merge of two refactorings (rename and move) with manual changes.

The rest of this paper is structured as follows. Section 2 gives an introduction to refactorings and various merge strategies. The two concepts of refactoring-aware merge and refactoring-aware versioning are introduced and motivated by examples in section 3 and section 4. Section 5 outlines a versioning model and merge strategy to support refactoring-aware versioning. Experiences from integrating that model and strategy in Eclipse is given in section 6. Related work is described in section 7 and section 8 concludes the paper and discusses some future work.

2 Preliminaries

2.1 Refactorings

Refactorings [1] are high-level source code changes that preserve the semantic meaning of the program but improves the source code, e.g. extracting common code in a method to avoid code duplication. Each refactoring has a set of pre-conditions that must be true when performing that refactoring to ensure that the code transformation results in a semantically equivalent program, e.g. there may not already exist a method with the chosen name when performing an extract method refactoring.

The following operations are typical refactorings, available in Eclipse, that will also be used to illustrate improved versioning proposed in this article:

rename Renames a class-, field-, or method-declaration as well as all uses of that same entity. When renaming a method, polymorphism and overriding must be taken into account. Pre-conditions involve checking that the new name is not already in use or shadowed by another declaration. A refactoring like rename class, that changes both declarations and use sites, often involves numerous changes scattered over the entire code base.

move Moves a field or a method from one class to another. Pre-conditions ensure that the move does not violate visibility and overriding requirements from invoking use sites.

2.2 Version control and history tracking

Version control is used to track the evolution of a document and to compare any two versions of a document. Every stable issue of a document's content

is considered a version and often explicitly created by the developers. The most important property of a version is its immutability, i.e. when a version has been frozen its content can never be modified. Versions can be organized in different ways. When organized in a sequence they are often called revisions and versions that are organized as parallel development lines are called branches. The difference between two documents is often represented as a series of add, change, and delete operations that are also used to recreate older versions of a document.

2.3 Merge

Merge is the task of combining changes made to one document in parallel into one unified version. Incompatible parallel changes results in a merge conflict that must be manually resolved. The following techniques represent the current state of the art and are orthogonal to each other and may thus be used in any combination. For a thorough survey on software merging see [7].

2.3.1 Textual, Syntactic, or Semantic merge

An important difference between merge tools is the way documents are represented. A textual representation of documents treats each document as a flat structure. The most common approach is to divide the document in single lines of text and perform merge on these lines. Because of this granularity parallel changes to the same line is not handled very well, since only one of these lines can be selected even when a combination of the two modifications to the same line is desirable.

A better approach is to take the syntactical structure of the documents in account. This is particular powerful in the context of programming languages where the syntax is clearly defined. The most common structural representation of software documents is a parse tree matching the context-free grammar for the used programming language. The merge tool can detect context-free conflicts and ensure that the merged result is syntactically correct.

An even more advances approach is to take the static semantics of the used language in account as well. The goal is to detect semantic merge conflicts that would generate static semantic compile-time errors or unintended run-time behavior when merged. This approach often rely on complex mathematical formalisms and current implementations do not work for full blown languages.

2.3.2 State-, Change-, or Operation-based merge

Another distinction between merge tools is the amount of information that is used during merge. In state-based merging only the original document and

the final result of the two concurrently changed versions are considered. A change-based merging tool uses information about the exact changes made to the documents. These changes often correspond the the actions taken in the integrated development environment. The extra information can be used for improved conflict detection. Operation-based merge is a special case of change-based merging that models the changes as explicit transformations.

3 Refactoring-aware merge

The main goal of a merge tool is to find changes made in parallel that are independent and then automatically merge them. Changes that are dependent result in a merge conflict that needs to be resolved manually. In a naive row-based merge-tool only changes made to the same row are considered dependent while a structural merge tool consider changes made to the same node dependent. While certainly useful, this type of dependency detection fails to detect certain dependences and detects some unintended dependences as well. Since the semantics and intention of a refactoring is clearly defined, this information can be used to perform a more intelligent merge that better reflects the intention of the developer.

The following examples, operating on Java [2] code, illustrate scenarios where a refactoring-aware merge tool can improve current state-of-the-art merge behavior and the result is compared to a traditional merge.

3.1 Automatic refactoring-aware merge instead of unintended merge

An automatic merge of two versions is not guaranteed to have the semantic behavior that the developers intended. The following example illustrates how a refactoring-aware merge can perform an automatic refactoring-aware merge that better reflects the developers intention when renaming a method in parallel with adding an invocation of that same method in another class, as illustrated in Figure 1. The semantics of a rename refactoring states that the declaration as well as all uses of the selected entity are to be renamed. Thus, a refactoring-aware merge will ensure that the added invocation in the parallel branch is renamed as well. A traditional merge performs an automatic merge but the added invocation still has the old class name and thus results in a compile-time error instead of a semantically correct program.

3.2 Automatic refactoring-aware merge instead of merge conflict

A traditional merge conflict may, in certain situations, be resolved automatically using a refactoring-aware merge. Consider moving a method from one

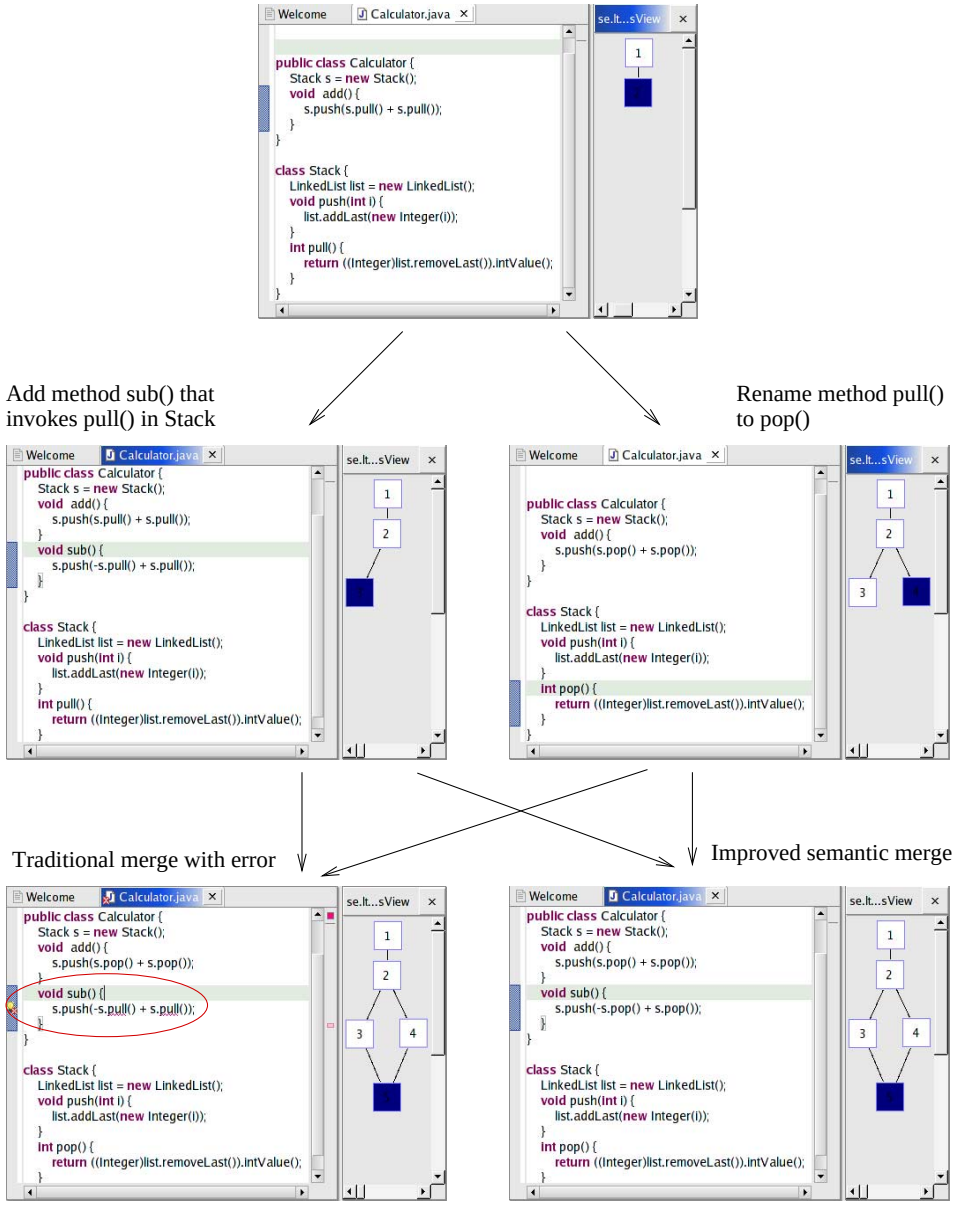

Figure 1. Rename method and add an invocation of the same method in parallel

class to another, e.g. pulling a method to a super class, and in parallel changing that same method. Traditional merge tools will view the single move operation as two unrelated delete and add operations done in parallel with the change operation. The delete and change operations affect the same code

and thus result in a conflict while the add operation is merged automatically. A refactoring-aware merge can detect that the changed method is to be moved and apply the changes to that method prior to moving it to its new location. The merge conflict in a traditional merge is replaced by an automatic refactoring-aware merge.

3.3 Refactoring merge conflict instead of unintended merge

The pre-conditions for a refactoring can be used to detect cases where an automatic traditional merge will not be semantically correct. Such errors are better resolved as merge conflicts than as compile-time errors. A merge conflict that originates from refactoring pre-conditions often turns out to result in numerous compile-time errors as will be discussed in section 3.4. Consider pulling down a method from a super class to one of its sub classes in parallel with adding a use of that same method in the super class. The pre-conditions for the pull down method refactoring states that the method may only be moved to the sub class when all invocations of that method are done through references that are statically qualified to the sub class. The added invocation violates these pre-conditions and the refactoring-aware merge will detect a merge conflict in contrast to the semantically incorrect automatic merge performed by a traditional merge.

3.4 Single refactoring merge conflict instead of numerous errors

A single refactoring often represents numerous changes and the pre-conditions guarantee that all those changes can be applied as a single transaction. By detecting errors that originates from a merge as a merge conflict instead of compile-time error can often reduce the number of experienced errors from the developer's point of view. Consider inlining a method, i.e. replacing all invocation sites with the method body, in parallel with changing the visibility of an attribute used in that method. The pre-conditions for inline states that all attributes used in the inlined method body must be visible from all method invocation sites. A refactoring-aware merge will detect this as an merge conflict that is shown as a single error to the developer while the traditional merge would merge the changes and there would be a compile-time error at each method invocation site where the method has been inlined.

4 Refactoring-aware history tracking

History tracking involves having full traceability of the evolution for each versioned element such as Java classes, methods, and fields. Traditional ver-

sioning systems loose the traceability of made refactorings. When two versions are compared, the semantic meaningful refactoring is not shown, but only its resulting changes.

4.1 Complete history even in the context of structural changes

To maintain full traceability for an element's evolution it is important that changes result in change operations instead of unrelated delete- and add-operations. Renaming a field should thus result in a change operation instead of a deletion of that field and an unrelated addition of a field with the new name. A similar problem stem from that several refactorings change the structure of the versioned tree and this is often recognized as two unrelated delete- and add-operations and the history is thus lost as described above. A typical example of the above described situation is extracting a method. Many refactorings that at first glance seem to preserve the tree structure in fact perform structural changes. Renaming a class may for instance result in a structural change if the the class nodes are ordered by their names. Thus, it is important that the version deltas contain refactoring information to be able to maintain as complete history traceability as possible.

4.2 More descriptive history

One important benefit from refactorings often neglected is the definition of a nomenclature for common high-level changes that improve source code. These refactoring names can also be used to provide more descriptive history tracking. Indicating that an element is renamed is more descriptive than the more traditional unrelated changes to the definition and all uses of that element that are scattered all over the source code.

5 Version model and merge strategy

The version model is designed to benefit from an existing refactoring functionality in integrated development environments. Therefore a client-server model is used where the server's only responsibility is to store a structural representation of the versioned data, while delta handling and merge is performed by the client. Thus, the server need not be aware of refactorings or the used merge strategy while the client can re-use existing infrastructure in the IDE for applying refactorings and pre-condition checking.

5.1 Structural versioning with node identity

It is natural to use structural versioning since programming languages are hierarchical in their structure. We let the tree nodes represent the abstract syntax tree for the language and the node contents contain the concrete syntax with full layout and comments.

Each node has identity through an individual immutable id. This id makes it easy to trace a node's history even when it has been moved or had its contents or type changed. The node identity enables us to describe a refactoring delta that is independent of parallel non-conflicting refactorings. For instance, if a method has its body changed and then moved in one branch while renamed in another branch, we can still locate the target node for that delta through its node id even when its content and location is changed.

5.2 Refactorings are first class changes

Instead of storing the result of refactorings as a set of concrete changes we store the refactoring operation itself, e.g. we store the operation *rename node N* instead of the individual changes to the declaration node N and all nodes using that same declaration. That way we not only group a possibly large set of changes as one operation but also store the desired semantics. This ensures that the programmer's intention is stored and may be used when presenting a diff or at a future merge.

Manual changes to the source code are stored as traditional delete-, add-, and change-operations but make use of node identity to specify the affected nodes. Since node identity is preserved when refactorings are executed these changes can still be applied after structural changes, e.g. a changed method body can be located even after the affected method is moved.

5.3 Refactoring-aware merge

The refactoring-aware merge tool has two main tasks: to detect semantic merge conflicts and, if possible, perform an automatic refactoring-aware merge. The merge is done in several steps. First, the traditional changes are merged and merge conflicts that originate from these changes are detected. Then, refactoring pre-conditions are checked to detect possible semantic merge conflicts that would be caused by applying that refactoring to the merged version. If the pre-conditions hold the merge proceeds with an automatic refactoring-aware merge. The traditional changes are first applied followed by the refactorings. This ensures that all traditional changes are affected by the whole source transformations caused by refactorings. Because node identity always is preserved, both for traditional changes and when refactorings are executed,

structural changes can successfully be merged with other changes, e.g. a changed method body can be located (and changed) even after the affected method is moved.

6 Implementation in Eclipse

We have a implemented a plugin that adds a refactoring-aware repository provider to Java Development Tools (JDT) in Eclipse for proof of concept. The current implementation supports refactoring-aware automatic merge for the rename and move refactorings. We believe that the approach is generic enough to be extended with other refactorings as well. The goal was to re-use as much infrastructure as possible in JDT while implementing the versioning model described in section 5.

6.1 Model

The Java Development Tools provide a high-level structural representation of a Java project called the Java Model where the nodes are handles to packages, compilation units, classes, class members, etc. These handles are used in refactoring operations to select target elements and also used to report affected elements. The node identity in that model is not preserved during refactorings and since our model requires immutable node identity we chose to create a separate Java Model (further called RAJM - Refactoring Aware Java Model) that replicates the JavaModel and also preserves node identity during refactorings.

Manual changes are directly replicated in RAJM. We register a Java ElementChangedListener to receive events when changes to the JavaModel occur. These are reported as traditional add, delete, change, and move deltas. While add and delete report changes all the way down to type members, change and move seem to only report changes down to the affected compilation unit. We therefore compare the entire compilation unit with RAJM in order to find the actual changes, which then are store in RAJM.

Refactorings should be more carefully traced. The series of changes caused by a refactoring should not always be directly replicated, but a the semantic behavior should be better stored. When, for example, a move is executed the delete and add operations made in the Java Model should be transformed to a true move to the affected node in RAJM.

To detect when a refactoring is executed we have added our own hooks in the current API (such hooks will be available in the forthcoming 3.0 to enable participation in the refactoring for third party plug-ins). In this way we have added the possibility to notify observers when a refactoring is initiated

and ended and can in that way bind model changes to a certain refactoring. In RAJM each refactoring has its own rules to transform the Java Model operations to RAJM.

Why not add these transformations directly to the existing refactoring implementation? It may seem like we need a lot of semantic knowledge to replicate these transformations in the version model. However, many refactorings have little or no net effect on the actual structure and sometimes provide information of the affected nodes. E.g. a rename refactoring does not change the structure but only the node contents, and a move element refactoring provides access to the source and target location elements. I.e. the actual transformation is a match of delete and add operations, transforming them to one move.

Since we used the JavaModel structure for RAJM the most fine-grained element is type members, and the node contents may thus be the entire method body. A more appropriate solution would be to use the underlying AST as the structure to replicate. However, the JavaModel is still needed for refactoring parameters and has a more convenient access to its syntactical source range than the AST nodes. Refactoring parameters are needed to represent refactoring deltas and the source range for the node contents. Our current implementation replicates the JavaModel and relies on a traditional merge for method bodies while still demonstrating the central concepts.

6.2 Deltas

We store two different types of deltas, deltas for manual changes and deltas for refactorings. Each set of manual changes between two refactorings or two explicit versions is stored as a single delta consisting of conventional add, remove, and change operations that operate on explicit node ids. Each refactoring results in a separate delta that stores the operation with the used parameters. Since JavaModel handles are used to select target elements and to report affected elements in refactorings, each parameter is first translated from a JavaModel handle to an immutable node id. That way the deltas can be applied to the desired target node even when the node has moved or had its content changed.

7 Related work

The use of static semantic information when merging versions is not a new idea but used in several approaches to semantic merge, e.g. [3]. Our approach differs in that we limit the semantic merge to refactorings where not only the semantic is well defined but where the intention from a developer point of view

is well understood.

By storing the refactorings as first class changes, our system is similar to ope-ration-based merging [5] and change-oriented versioning [4]. These approaches do not, however, consider operations of as high-level as refactorings and the strict semantic behavior defined for the operations.

Two approaches that detect and merge renamed entities in software are described in [8] and [6]. In [8], a generic rename detection for languages with a nested block structure is described and implemented using contextual information. However, this approach does not work on object-oriented languages. In [6], a statistical rule based method is used to detect renamed entities. Neither approach tries to generalize these ideas for other semantic sensitive changes.

8 Conclusions and future work

We have presented refactoring-aware versioning and refactoring-aware merge, two techniques to better support refactorings during team development. Refactoring-aware versioning treats refactorings as first-class changes described as semantic actions rather than the set of resulting changes scattered over the source tree. Refactor-ing-aware merge of branches supports automatic semantic merge of refactorings as well as traditional changes. The semantic meaning of a refactoring is used both to detect semantic merge conflicts during the merge and to ensure that the semantic meaning is preserved also after the merge.

The improvement from an refactoring-aware versioning tool is illustrated by the following scenarios:

- Automatic refactoring-aware merge instead of unintended merge
- Automatic refactoring-aware merge instead of merge conflict
- Refactoring merge conflict instead of unintended merge
- Single refactoring merge conflict instead of numerous errors

The technique has been implemented by extending a generic syntactic merge with refactoring-aware merge support for Java in a prototype plug-in for Java Development Tools in the Eclipse environment. Since the technique uses the existing refactoring support we believe it is suitable for integration in any integrated development environment supporting refactorings and versioning.

There are many interesting ways to continue this research.

Interactive merge We would like to further investigate how the improved detection of merge conflicts can be used to simplify the interactive merge needed to resolve conflicts.

Visible diff The improved version history tracking that is discussed in sec-

tion 4 provides further challenges for graphical visualization of version differences.

More refactorings Currently the rename- and move-refactorings are implemented but we plan to implement more refactorings to verify that the approach is generic enough.

Version model granularity The current implementation uses the Java model to detect changes to the source tree as well as the model to version. Since the finest granularity in the Java model is at a type member level, we plan to use the abstract syntax tree to improve the granularity down to expressions.

References

[1] Fowler, M., "Refactoring: improving the design of existing code," Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] Gosling, J., B. Joy, G. Steele and G. Bracha, "The Java Language Specification Second Edition," Addison-Wesley, Boston, Mass., 2000.

[3] Horwitz, S., *Identifying the semantic and textual differences between two versions of a program*, in: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation* (1990), pp. 234–245.

[4] Lie, A., R. Conradi, T. M. Didriksen and E.-A. Karlsson, *Change oriented versioning in a software engineering database*, SIGSOFT Softw. Eng. Notes **14** (1989), pp. 56–65.

[5] Lippe, E. and N. van Oosterom, *Operation-based merging*, in: *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments* (1992), pp. 78–87.

[6] Malpohl, G., J. J. Hunt and W. F. Tichy, *Renaming detection*, in: *Automated Software Engineering*, 2000, pp. 73–80.

[7] Mens, T., *A state-of-the-art survey on software merging*, IEEE Transactions on Software Engineering **28** (2002), pp. 449–462.

[8] Westfechtel, B., *Structure-oriented merging of revisions of software documents*, in: *Proceedings of the 3rd international workshop on Software configuration management* (1991), pp. 68–79.