Refactoring-Aware Version Control

Towards Refactoring Support in API Evolution and Team Development

Tammo Freese
Software Engineering Group
Department of Computing Science
University of Oldenburg
Oldenburg, Germany
tammo.freese@informatik.uni-oldenburg.de

ABSTRACT

Today, refactorings are supported in some integrated development environments (IDEs). The refactoring operations can only work correctly if all source code that needs to be changed is available to the IDE. However, this precondition neither holds for application programming interface (API) evolution, nor in team development. The research presented in this paper aims to support refactoring in API evolution and team development by extending IDE and version control to allow refactoring-aware merging and migration.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control, restructuring, reverse engineering, and reengineering*; D.2.3 [**Software Engineering**]: coding tools and techniques; D.2.9 [**Management**]: software configuration management; K.6.3 [**Software Management**]: software development, software maintenance

General Terms

Design, Languages

Keywords

refactoring, version control, software evolution, application programming interface, library, eclipse, subversion

1. MOTIVATION

According to Martin Fowler, a refactoring is *"a change made to the internal structure of a software to make it easier to understand and cheaper to modify without changing its observable behavior."* [7]

Some integrated development environments (IDEs) support semi-automated refactoring operations [4, 11]. Using an IDE's refactoring support instead of applying refactorings manually has two benefits: First, the preconditions of each refactoring are checked. Second, necessary changes are identified and applied consistently on all available source code.

Frequent refactoring is supposed to preserve design quality while developing the software. Refactoring is also crucial in application programming interface (API) evolution: Danny Dig and Ralph Johnson analyzed the changes between releases of four APIs (Eclipse, Mortgage, Struts and log4j), finding that between 81% and 97% of the changes were refactorings [2].

A refactoring operation can only affect those parts of the code that are available to the IDE when the operation is applied. Unfortunately, there are at least two scenarios where not all code that needs to be changed is available at that time: team development and API evolution.

In team development, refactoring operations applied in different local workspaces can conflict with each other, and they can conflict with non-refactorings, i.e. editing, as well. Conflicts are likely for global refactorings that touch many parts of the source code, like renaming or inlining a widely used method, or improving the package structure.

Using pessimistic locking for affected parts of the code does not solve the problem, since developers may still write new code that references the locked parts. Teams may work around this problem by applying global refactorings either at the same time in the whole team, or with exclusive access to the version control after all previous changes have been committed. These strategies, however, will delay or hinder global refactorings, and they only work well for small teams in one time zone.

While refactorings are important in API evolution, they break backward compatibility in most cases, so that clients of the API must be migrated to the new version. A manual migration is very tedious, and sometimes error-prone, since some refactorings do not lead to compile errors, although changes are required. An example is the reordering of method parameters which have the same type: Clients of the API would still compile after the refactoring, but they would use the parameters in the old order which is now wrong.

As a result, refactoring of APIs is typically discouraged. Recommendations are to publish as little as possible as late as possible [6], or to handle changes as additions by leaving old signatures intact, and letting them delegate to the new API. But publishing as little as possible as late as possible defers publication of the API. Solving the problem with additions forbids real changes and deletions, and in consequence bloats the API with items only kept for backward compatibility.

2. PROPOSED APPROACH

The goal of the author's PhD thesis is to provide a concept for coping with refactoring in team development and in API

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

evolution. The focus will be on merging parallel changes in team development, and semi-automated migration of client code in API evolution. A software prototye will be integrated in a version control and an IDE as a proof-of-concept. Compared with [9] and [8], this paper gives a revised and extended description of the author's research.

The first step towards reaching the goal is obvious: The information which refactorings were applied must be available at the time of merging and migration. Therefore, the IDE must not only allow the application of refactorings, but also keep the information which refactorings were applied.

However, capturing refactoring information only is not sufficient. As an example, the body of a method may be edited before the method is inlined. If the information about the editing is not captured, inlining the method would be a different operation. It may even be not applicable anymore, e.g. if the editing removes a call to a private method, the refactoring's precondition may not hold without the editing. This leads to the first hypothesis of this research:

> *To support refactoring in merging and migration, all development steps – refactoring operations as well as edit steps – have to be taken into account.*

2.1 Capturing Development Steps

To capture all development steps, an IDE extension is used. A development step is either a refactoring step, or an edit step. For the state after each step, compilation information is captured if available.

When changes should be committed to the version control, the development steps are first condensed. After checking that there are no pending updates from the version control, the commit stores the condensed development steps as additional information in the version control.

The condensed version of the steps is determined by first removing all development steps that were undone. Then refactoring steps are replaced with edit steps if they do not lead from compiling code to compiling code. Finally, subsequent edit steps are summarized to one edit step.

Steps which were undone are left out since they have no influence on the committed version. Replacing refactoring steps with edit steps is necessary because some development environments allow applying refactorings to non-compiling code, although in these cases it cannot be guaranteed that they are behavior-preserving. Subsequent edit steps are summarized to reduce the amount of work if merging is required.

2.2 Merging

If a commit fails due to pending updates from the version control repository, the local changes have to be merged with the changes from the repository.

As an example, we assume we have checked out program p which is version 7 in the repository, and have changed it locally by applying steps $L = l_1, \ldots, l_n$ to $p_l = (l_n \circ \cdots \circ l_1)(p)$. In the meantime, version 8 has been committed in the repository, in which p has been changed with steps $R = r_1, \ldots, r_m$ to $p_r = (r_m \circ \cdots \circ r_1)(p)$.

This scenario is similar to operation-based merging [13], where a snapshot of an object management system (OMS) is changed by two sequences of transformations (here the program p is changed by the sequences L and R). In the approach presented in [13], the transformations from both sequences are mixed, and conflicts are resolved by imposing an ordering on the transformations, deleting transformations, or by editing transformations.

So in [13], the merge result can be understood as a sequence of transformations from version 7 to a new version 9. Using this sequence of operations poses a problem: There is no simple sequence of transformations available from version 8 to version 9. So if another developer checked out version 8, he could not use operations in merging.

For this reason, another approach is used here: Instead of mixing the sequences of the steps, the output of the merge should be adapted steps $L' = l'_1, \ldots, l'_k$ and $R' = r'_1, \ldots, r'_s$ such that applying the adapted repository steps R' to the local version p_l has the same result as applying the adapted local steps L' to the repository version p_r:

$$(l'_k \circ \cdots \circ l'_1)(p_r) = (r'_s \circ \cdots \circ r'_1)(p_l)$$

After determining these steps, we may apply the steps R' to the local version, leading to the merge result. Then the local version is the same as applying the steps L' to the repository version, so we may then commit the changes L' to the repository. This approach leads to the second hypothesis of this research:

> *To support refactoring in merging and migration, the changes between two subsequent versions of a program have to be available as a sequence of steps.*

Only two requirements for the version control repository can be identified: First, the steps have to be stored. This should pose no problem, since the information may be stored in an additional flat file. Second, the repository has to support atomic commits, because in a partial commit, the changed files would not match the stored steps. The repository does not need to be aware of the steps, as these as captured, replayed, and merged on the client side.

2.3 API Evolution

When an IDE is extended with the capturing and merging capabilities described in the previous two sections, the version control contains development steps from each version to each subsequent version.

If a library is developed, the steps from one release to another may be obtained from the version control, and then applied on clients of the library. In this semi-automated migration, most of the changes are expected to work automatically, reducing migration costs drastically. In conflict cases, the client code has to be changed to remove the conflict, because the evolution of the API cannot be changed by the client.

2.4 Merging in Detail

In merging, a program p is changed locally to p_l by applying steps L, and to p_r on the repository by the steps R. The merge output should be steps $L' = l'_1, \ldots, l'_k$ and $R' = r'_1, \ldots, r'_s$ such that $(l'_k \circ \cdots \circ l'_1)(p_r) = (r'_s \circ \cdots \circ r'_1)(p_l)$.

Each step can either be a refactoring step or an edit step, and is modelled as a partial function. For a refactoring step, the domain of the function contains all programs that satisfy the precondition of the refactoring, and the function maps each of these programs to the program after the refactoring. An edit step that changed the program p_1 to the program p_2 is defined as the partial function that maps p_1 to p_2, and is undefined for all other programs.

2.4.1 Local and Global Commutation

At first, we only look at merging two single steps l and r applied to program p. We define local commutation and global commutation as follows: l and r commute locally for p if and only if $(l \circ r)(p) = (r \circ l)(p)$. This implies that $r(p)$ is in the domain of l and $l(p)$ is in the domain of r. The two steps commute globally if and only if they commute locally for every program contained in the domains of both functions. These definitions deviate from those presented in [13] in using equality instead of equivalence, and in explicitely defining global commutation for partial functions.

If two steps commute locally, then the merge output is $l' = l$ and $r' = r$. The merge result $(l \circ r)(p) = (r \circ l)(p)$ conforms to the basic assumption of operation-based merging that Lippe and Ooesterom define as follows:

> *"When two transformations commute locally on their initial state, the final result is a good candidate for the result of the merge."* [13]

If the steps l and r do not commute locally, there is a conflict, which has to be resolved either manually or automatically.

2.4.2 Detecting and Resolving Conflicts

To detect and resolve conflicts, the meaning of the steps is crucial, as there are cases where we can either conclude that the steps commute globally, or that there is a conflict and how it may be resolved – both without taking the program p into account.

For two refactoring steps, let us look at two *rename method* steps. If source and target names are all different, the steps commute globally. If the same method `a` is renamed to `b` on the repository and to `c` locally, there is a conflict, which may be resolved by choosing l' to be *rename* `b` *to* `c` and r' to be the identity function.

For a refactoring step and an edit step, conflicts may be ruled out by examining the changes in the edit step. If the refactoring renames method `a` to `b` and the editing neither changes existing class hierarchies, nor introduces calls to `a` or defines a method `a` or `b`, there cannot be a conflict.

Edit steps are special because they always conflict by definition, unless they are equal. There are cases where the conflicts can be resolved by examining the changes in both steps. For example, if both only change different method bodies, the program p_m in which all changes of both edit steps are applied will compile. The adapted local step l' can then be defined as the edit step from $r(p)$ to p_m and the adapted repository step r' as the edit step from $l(p)$ to p_m.

There are cases where the conflict is so severe that the local step has to be changed. An example would be a local *rename method* from `a` to `c` and a *rename method* from `b` to `c` in the same class on the repository. In this case, l' may be renaming `a` to `d`, and r' then needs two steps: First, renaming `c` to `d`, and then renaming `b` to `c`.

2.4.3 Merging Two Step Sequences

The merging of two step sequences relies on the third assumption of this research:

> *In the majority of all cases, conflicts can either be ruled out, or detected and resolved, both without inspecting the program source.*

To merge the local step sequence $L = l_1, \ldots, l_n$ with the repository step r_1, first the step l_1 is merged with r_1. The result are adapted steps l'_1 and r'_1. Then l_2 is merged with r'_1 to get l'_2 and r''_1. This is repeated for all local steps to get $l'_1, \ldots, l'_n$ that may be applied after r_1, and the adapted r^n_1. By repeating this for every repository step, we get the merge result.

Regarding the number of conflict detection/resolving computations, the best case scenario is that local steps and repository steps are equal: With n local and m repository steps, $n = m$ computations are needed. If there are no conflicts in the whole merge, $n \cdot m$ computations are required. The worst case scenario is a severe conflict on every check, leading to $2^n \cdot m$ computations, but if there are that many conflicts, it is likely that the local steps are undone anyway.

3. RELATED WORK

Using operations in merging as well as using refactoring for migration can be found in other approaches for version control and API evolution.

3.1 Operation-based Merging

Lippe and van Oosterom propose *operation-based merging* for a snapshot of an object management system (OMS) that is transformed by two sequences of transformations [13, 12]. In merging, the transformations from both sequences are mixed to build a sequence of blocks, where only blocks contain conflicting transformations. The conflicts are resolved by imposing an order on the transformations, or by deleting or editing transformations. The merge result is then retrieved by applying the transformations from each block.

To build the blocks, global commutation information is used without considering the transformations applied before. This is possible as the OMS for which operation-based merging was written has unique object identifiers, and stores references between objects explicitely [12]. In the context of this research, however, there are no unique identifiers, and therefore, the merge algorithm cannot work. As an example, let us assume that locally, method `a` is renamed to `c`. In the repository, it is renamed to `b`, then edited, then renamed from `b` to `d`. The first local step and the third repository step commute globally, and would as such not be considered to be in conflict by the algorithm of Lippe and van Oosterom, although the same method is renamed.

Furthermore, the algorithm proposed by Lippe and van Oosterom mixes the steps, and unfortunately gives no hint how the merge result is included in the OMS so that subsequent merging works.

3.2 Refactoring-Aware Versioning

Ekman and Asklund present a refactoring-aware versioning [5]. The version control stores the abstract syntax tree (AST), a structural representation of the program. Each node gets an individual immutable id, which allows to describe refactorings independent to parallel, non-conflicting refactorings. Refactorings as well as traditional changes are recorded by an extension to the Eclipse IDE [4]. During the merge process, the changes are reordered:

> *"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 the refactorings."* [5]

The approach of using unique ids has the appeal that some refactoring operations may indeed be applied last. However, this does not work in all cases. As an example, we consider that method `a` was inlined and then one of the places where the method got inlined was edited, while `a` has been edited on the repository. Applying the refactoring last would result in a successful merge, although there should be a conflict, since the code edited locally is not the code that is inlined when applying the refactoring last.

As in operation-based merging, the steps are mixed to generate the merge result, and how subsequent merging works is not shown. The structural versioning used requires a specialized version control.

3.3 Automated API Evolution

Perkins proposes an approach for automatically generating refactorings to support API evolution for Java:

> *"The idea is a simple one: replace calls to a deprecated method by the method's body."* [14]

To achieve this, a tool at the client side finds deprecated methods in the library. It then generates source code for these methods from the byte code, and replaces client calls to the methods by inlining the source code. The solution is very lightweight: It does only need a tool at the client side, and it does not require the source code of the library.

Relying on the `deprecated` tag for migration limits the applicable changes: Repackaging or renaming classes can only be done by subclassing, which is dangerous as it may break `equals()` implementations. Changing method return values or method behavior by appropriate refactoring sequences is not possible, because in these cases old and new method signature cannot both exist at the same time. Furthermore, even simple changes like renaming a method cannot be finished in one new release of the software, since an intermediate release is required that contains the old method signature as a deprecated delegator to the new one.

3.4 CatchUp!

Henkel and Diwan address API evolution with CatchUp!, a tool that captures API refactoring actions, and allows clients of the API to replay those refactorings later [10]. Both features are realized by an extension to the Eclipse IDE [4]. Both capturing and replaying are thus independent of the version control system used. In replaying the operations, method bodies are not used, but stubs are generated to represent the API.

While the approach is lightweight, it has two disadavantages. First, as only refactorings are considered, and method bodies are not used, refactorings that need the method body cannot be applied. This is a severe restriction, as *inline method* which uses the method body is crucial for getting rid of old API parts [8, 14]. The second disadvantage is that the approach restricts refactorings to be done at one workplace in one refactoring session, thus refactoring can neither be done by every team member, nor during normal development.

3.5 Refactoring Support for Class Library Migration

Balaban, Tip and Fuhrer present an approach in which a programmer may specify mappings between legacy classes and their replacements [1]. By an analysis of *type constraints*, it is determined which places in the code may be updated automatically.

The approach aims at migrating applications that use legacy library classes, and looks very promising for this scenario. The research presented in this paper tackles the problem at an earlier stage: By providing automated migration for client code, there should be little need to keep legacy code for backward compatibility.

4. CURRENT STATUS

The concepts for merging and migration are formally specified using Object-Z [3], an object-oriented extension of Z [15]. At the time of writing, capturing edit steps and refactoring steps, condensing steps as well as the basic merge algorithm are specified. The current focus is on specifying how single steps are merged.

The implementation of the approach requires an IDE which allows to capture and replay refactoring and editing steps, and a version control with support for atomic commits. As a proof-of-concept, a prototype is under development for the Eclipse IDE [4] and the Subversion version control [16].

To evaluate the concept, controlled experiments in student classes are planned to examine how the merging approach performs compared to traditional merging, and whether the migration support influences the change frequency in library evolution.

5. REFERENCES

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05 Proceedings*, pages 265–279, 2005.

[2] D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proceedings of the ICSM 2005*, 2005.

[3] R. Duke, G. Rose, and G. Smith. Object-Z: a specification language advocated for the description of standards. Technical Report 94-45, Software Verification Research Centre, The University of Queensland, 1994.

[4] Eclipse. http://www.eclipse.org/.

[5] T. Ekman and U. Asklund. Refactoring-aware versioning in Eclipse. *Electronic Notes of Theorical Computer Science*, 107, 2004.

[6] M. Fowler. Public versus published interfaces. *IEEE Software*, March/April 2002.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] T. Freese. Inline method considered helpful: An approach to interface evolution. In *Proceedings of XP 2003*, volume 2675 of *LNCS*, pages 271–278, 2003.

[9] T. Freese. Towards software configuration management for test-driven development. In *Software Configuration Management ICSE Workshops SCM 2001 and SCM 2003*, volume 2649 of *LNCS*, 2003.

[10] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proceedings of ICSE '05*, 2005.

[11] IntelliJ IDEA. http://www.jetbrains.com/idea/.

[12] E. Lippe. *CAMERA – Support for distributed cooperative work*. PhD thesis, Utrecht University, 1992.

[13] E. Lippe and N. van Oosterom. Operation-based merging. *Software Engineering Notes*, 17(5):78–87, 1992.

[14] J. H. Perkins. Automatically generating refactorings to support API evolution. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, 2005.

[15] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[16] Subversion. http://subversion.tigris.org/.