ComeBack!: a Refactoring-Based Tool for Binary-Compatible Framework Upgrade *

Ilie Şavga
Institut für Software- und Multimediatechologie
Technische Universität Dresden, Germany
is13@inf.tu-dresden.de

Michael Rudolf
Institut für Software- und Multimediatechologie
Technische Universität Dresden, Germany
s0600108@mail.inf.tu-dresden.de

Sebastian Götz
Institut für Software- und Multimediatechologie
Technische Universität Dresden, Germany
sebastian.goetz@mail.inf.tu-dresden.de

ABSTRACT

Maintenance of a software framework often requires restructuring its API (*refactoring*). Upon framework upgrade structural API changes may invalidate existing plugins—modules that used one of its previous versions. To preserve plugins, we use refactoring trace to automatically create an adaptation layer that translates between plugins and the framework. For each encountered refactoring we formally define a *comeback*—a refactoring to construct adapters. Given an ordered set of refactorings occured between two framework versions our tool ComeBack! executes the corresponding comebacks and yields the adaptation layer.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Experimentation, Design, Languages

Keywords

Maintenance, adaptation, refactoring, software framework

1. REFACTORING-BASED ADAPTATION

A software framework [7] may evolve considerably due to new or changing requirements, bug fixing, or quality improvement. As a consequence, existing modules (*plugins*) that use the framework may become invalid. Either plugin developers are forced to manually adapt their plugins or framework maintainers need to write update patches. Both tasks are usually expensive and error-prone. When the application has been delivered to a customer, it even may be undesirable to require plugin recompilation.

We want to achieve *binary compatibility* of framework plugins—existing plugins link and run with new framework releases without recompiling [6]. We use the information about framework refactorings — behavior-preserving source transformations — to automate the adaptation. According to the empirical study of Dig and Johnson [4], who investigated the evolution of five big frameworks, most (from 81% to 97%) of the application-breaking changes were comprised by refactorings. We treat a refactoring operator as a specification of a syntactic change to create the corresponding adapters (Fig. 1). The adapters shield the plugins by representing the public types of the old version, while delegating to the new version. Our adapter generation is not limited to two consecutive framework versions; adapters can be generated for any previous API version. Moreover, the adapted plugins of different versions may co-exist. In addition, the adapters are not stacked on top of each other; each adaptation layer delegates directly to the latest framework version.

Comebacks For each supported refactoring we formally define a *comeback*—a behavior-preserving transformation, which defines how a compensating adapter is constructed. That is, we create an adaptation-oriented pattern problem-solution library of transformations, where a problem pattern is the occurrence of a component refactoring and its solution

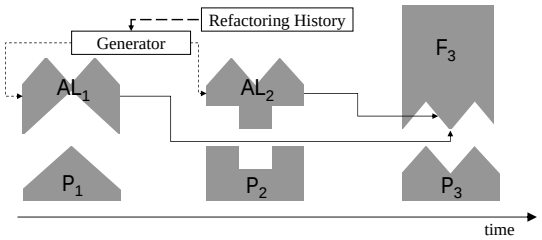

Figure 1: Refactoring-based plugin adaptation. The last framework version F_3 is deployed to the user. While new plugins (P_3) are developed against the latest version, existing ones (P_1 and P_2) are preserved by creating adapter layers AL_1 and AL_2.

*The presented work is funded by the Sächsische Aufbaubank, project number 11072/1725.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

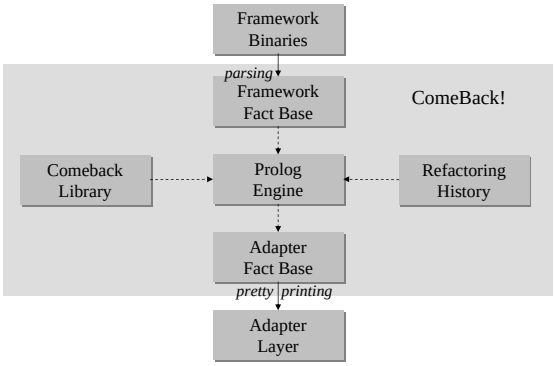

Figure 2: General Architecture of ComeBack!.

is the corresponding comeback (refactoring on adapters). Technically, a comeback is realized in terms of refactoring operators executed on adapters. For some refactorings, the corresponding comebacks are simple and implemented using a single refactoring. For example, the comeback that corresponds to the refactoring `RenameClass(name, newName)` consists of a refactoring `RenameClass(newName, name)`, which renames the adapter to the old name. The comebacks of other refactorings consist of sequences of refactorings. For instance, the comeback of `PushDownMethod` is defined by the `DeleteMethod` and `AddMethod` refactoring operators, the sequential execution of which effectively moves (pushes up) the method between adapters. Moreover, complex comebacks may be defined by composing other, more primitive comebacks. This is the case for the comeback of `Extract-Subclass` defined by combining the comebacks of `PushDown-Method` and `AddClass`. To ensure the validity of comeback specifications, we formally ground their definitions [3].

Tool Validation Figure 2 shows the main modules of our adaptation tool ComeBack!. For a number of common refactorings we provide a comeback library consisting of comeback transformations specified as Prolog rules. Given the latest framework binaries, the information about the API types (type and method names, method signatures, inheritance relations) is parsed into a Prolog fact base. After examining the history of framework refactorings, the corresponding comebacks are loaded into the engine and executed on the fact base as described in the previous section. Once all comebacks have been executed, the fact base contains all the necessary information for generating adapters (it describes the adapters) and is serialized to the adapters binaries. The serialization logic encapsulates implementation-specific decisions, such as special treatment of constructors and of object schizophrenia implied by delegation.

Currently we are supporting Java and .NET binaries, for which we developed the corresponding Prolog/Java and Prolog/CIL parsers and pretty printers. Our support for .NET is motivated by the requirements of one of our industrial partners and for Java by a plethora of open source frameworks available for case studies. We perform now a case study with a medium-size framework called SalesPoint [2] developed at our department. It serves as an excellent case study for our technology, because developers were not restricted in changing its API and because a number of clients (student exercises) for each major version exist. Along with the extended ComeBack! functionality we also aim for benchmarking the overhead implied by delegation and the tool's *recall*—the percentage of adapted changes from the overall number of detected ones. The tool's sources and documentation as well as the relevant publications are on [1].

Currently we adapt the following refactorings: `Rename-Method`, `RenameClass`, `AddMethod`, `AddClass`, `MoveMethod`, `PullUpMethod`, `PushDownMethod`, `Extractclass`, `Extract-Superclass`, `ExtractInterface`, `ExtractMethod`, `Extract-Subclass`. Because we assume, that all API fields are encapsulated, which is a general requirement in our project, support for field refactorings is implied by the adaptation of the corresponding accessor methods.

Related Work In their binary adapter tool ReBA Dig et al. [5] define, for each original refactoring, a compensating refactoring that *inlines* the corresponding code directly in the library. For example, for `RenameMethod(oldMd, newMd)` compensates `AddMethod(oldMd)` inserting the method that delegates to `newMd`. Given an old library and a refactoring trace, they execute the compensating refactorings on the old library in the same order as the original refactorings. Effectively, instead of putting a wrapper around the library, they give it two (the old and the new) interfaces. Similarly to our approach, the object identities are preserved and the side-by-side execution is supported. As they do not use delegation, the performance penalties are reported to be less then 1%. Moreover, having access to the old implementation they recover deleted methods. However, their adaptation does not handle refactorings contradicting each other in the scope of a class (e.g. deleting a method `M` and then renaming another one to `M`). Moreover, ReBA is Java-centric, cannot be re-used for other languages and supports only several refactorings (e.g., no interface refactorings). Last, and most important, the compensating refactorings are ad hoc defined, implementation-specific, and not formally validated.

2. REFERENCES

[1] ComeBack! homepage. `http://comeback.sf.net/`.

[2] SalesPoint homepage. `http://www-st.inf.tu-dresden.de/SalesPoint/v3.1/index.html`.

[3] I. Şavga and M. Rudolf. Refactoring-based adaptation for binary compatiblity in evolving frameworks. In *GPCE'07: Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, Salzburg, Austria, October 2007.

[4] D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[5] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *ICSE'08: International Conference on Software Engineering*, May 2008.

[6] I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.

[7] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.