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 occurred 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. REFACTORIZATION-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 extended ComeBack! functionality we also aim for benchmarking exercises for each major version exist. Along with study for our technology, because developers were not redeveloped at our department. It serves as an excellent case is motivated by the requirements of one of our industrial log/CIL parsers and pretty printers. Our support for .NET which we developed the corresponding Prolog/Java and Prolog/Superclass, ExtractMethod, 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 than 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.

**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: RenameMethod, 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.

![Figure 2: General Architecture of ComeBack!](image-url)