API Changes - How Much Would You Allow?

Michael Rudolf, Ilie Savga
Technische Universität Dresden
Nöthnitzer Str. 46, 01187 Dresden, Germany
{michael.rudolf,ilie.savga}@inf.tu-dresden.de

Jan Lehmann, Jacek Śliwerski, Harald Wendel
Comarch Software AG
Chemnitzer Str. 50, 01187 Dresden, Germany
{jan.lehmann,jacek.sliwerski,harald.wendel}@comarch.com

Abstract

Once a software library is deployed, it is extremely difficult to modify published types. Every new version has to preserve backward compatibility *with existing applications. This limits the number of changes that may be applied to the public classes defined in the library. We present a tool that assists developers in maintaining the consistency of a shared library with the existing software. It allows for greater flexibility in evolving the library's functionality by supporting a wide range of changes.*

1. Introduction

Backward compatibility of public classes and interfaces is one of those concerns that may limit the development of any software product. Once published (i.e. deployed), classes and interfaces may only change in a very limited way, otherwise the applications compiled and linked against them may stop working correctly. For instance, if a method in the API is renamed, the developer is obliged to keep the old one, too, marking it as *deprecated*. Although widely used, this practice bloats the API and does not guarantee that the method will not be used by new software packages.

Figure 1. Interface adaptation. After the upgrade applications use a type-identical set of interfaces that redirect the calls to their new versions.

In order to alleviate the problem of API evolution of one of our ERP systems, we have created a tool that allows for the adaptation of different versions of shared libraries. As the system's API is mostly used by third-party companies, we assumed unavailability of their sources at the adaptation time.

Figure 1 presents an overview of our approach. The general idea is to supplement the new version of the library with a set of adapters [3] that constitute a redirection layer between the old and new API types. This approach allows for a large spectrum of changes to the classes and interfaces, such as renaming and removing types and their members, or modifying the signatures of methods.

2. Adapter Development Environment

The development of adapters is a tedious and monotonous process. The environment we have created is meant to ease this work.

Figure 2. Adapter Development Environment encapsulates the compiler and a heuristics engine that help developers create adapters between two versions of assemblies.

1. First, old and new versions of assemblies are compared by the heuristic engine which tries to identify elements they have in common.
2. Next, the resulting source file is edited by the developer, who amends and completes the source.
3. Finally, the compiler generates the adapters.

Figure 2 presents the components of the tool and the related workflow: All the work is performed within a development environment the developers are accustomed to (see Figure 3). The method bodies are defined in a special-purpose C#-like programming language, which supports multiple versions of the same type in one source file and features partial type inference and strict type checking.

Figure 3. The Adapter Development Environment integrated into the Visual Studio suite.

3. Related Work

Griswold and Notkin [4] discuss how to automatically propagate changes in the software libraries to the application code. Their approach requires the full availability of the application source code at the time of library change. This requirements is relaxed in the approach of Chow and Notkin [2], where the library maintainer annotates the changes in the API interfaces and specifies rules for adapting application code, that used the old library version, to the new release. Based on change and rule specifications, the old application source code is then re-written by a transformation engine.

The idea of recording the *refactorings* (behaviour-preserving source-to-source program transformations) applied to a software library for their later application to the client code is the basis of CatchUp! [5]. The tool, implemented as a plugin for Eclipse, is able to listen for occurrences of, capture and record framework refactorings in a file. The file is then delivered to the application developer, who "replays" them on the application code, which used the old framework version.

The work closest to ours in spirit is of Balaban et al. [1]. To automatically migrate (Java) applications that use an evolved legacy system, they provide a *migration specification*. The latter defines how to map uses of old legacy classes to their replacement classes. The authors rely on type constraint analysis to control the correctness of migration and to preserve the program behaviour.

The related technologies presented in this chapter require the availability of application sources. In addition, all of them are not transparent for the application user as they require recompilation of existing applications. These two limitations have been considered unacceptable for our project.

4. Conclusions

Although the presented tool does not eliminate the burden of preserving the *semantic* backward compatibility, it allows for a wide range of changes applicable to an API in a backward compatible manner. Moreover, in contrast to similar approaches our tool:

1. is transparent for existing applications, and

2. does not require access to their sources.

Acknowledgments. The ADE project is funded by the Sächsische Aufbaubank, project number 11072/1725.

References

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

[2] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[4] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[5] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.

A. Tool Presentation

Slides: Brief Introduction to API Adaptation

We will begin our presentation with a summary of our approach in order to sketch the general idea of API adaptation and to explain all the terms we will use in the demo.

Demo: Adapting an Application

In the demo we will apply our tool to the real interfaces of our ERP system.

1. We will present a very simple application that imports information from external sources of data into the system using its API.
2. We will present the outcome of the program - the imported information will be visible in our system.
3. We will adapt the interfaces and upgrade the system.
4. Once again we will run our importing application and show that the data has been imported into the new version of the system.

B. Screenshots