# A Refactoring-Based Approach to Support Binary Backward-Compatible Framework Upgrades

**Dissertation**

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

**M.Sc. Ilie Şavga**
geboren am 02.02.1975 in Chişinău, Republik Moldau

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann (Technische Universität Dresden)
Prof. Dr. Friedrich Steimann (Fernuniversität in Hagen)

Tag der Verteidigung: Dresden, den 21 April 2010

Dresden, den 16 Juni 2009

*To my parents and wife, for their endless love and support*

# Abstract

A successful *software framework*—a software component encompassing reusable design decisions for a family of related applications—usually has to evolve considerably due to new or changed requirements, or quality improvements. Although the Application Programming Interface (API) of a framework should stay stable, in practice it often changes during framework evolution. When upgrading the framework to a new, improved version, such changes may invalidate *plugins*—modules that instantiated one of the previous framework versions into a final application. Such application-breaking API changes are *backward-incompatible* with existing plugins: plugin sources cannot be recompiled, or plugin binaries cannot be linked and run with a new framework release. When upgrading the framework, its developers are forced to either manually adapt plugins or manually specify update patches. Both tasks are usually error-prone and expensive, the costs often becoming unacceptable in case of large and complex frameworks. Existing approaches to automate component adaptation demand cumbersome specifications, which the developers are reluctant to write and maintain. Finally, *intrusive* update of existing applications via modifying their sources or binaries may be impossible at all, in case their sources or binaries are unavailable, or software licenses forbid their change.

This thesis elaborates on an adaptation technology that supports *binary backward-compatible framework upgrade*—existing plugins link and run with the new framework release without recompiling. To protect existing plugins from API changes, we create binary adapters that constitute an adaptation layer placed between plugins of an old version and the upgraded framework. From the point of view of an existing plugin, the adaptation layer looks like the original framework (before the API changes), while from the point of view of the framework the adaptation layer looks like the updated plugin. Via the adaptation layer, plugins may invoke the improved functionality of the upgraded framework, while the framework may call back functionality of existing plugins.

Most adapters of the adaptation layer can be created automatically basing on the history of API changes. This automation is inspired by several case studies of API evolution, reporting that most application-breaking evolutionary API changes of software components are *refactorings*—behavior-preserving program transformations. Since refactorings have formally defined semantics, we know precisely how they change framework APIs and, in case of application-breaking API refactorings, can formally define adaptation actions. In particular, we rely on the semantics of application-breaking API refactorings to automatically derive compensating binary adapters in our adaptation tool ComeBack!. The tool evaluation shows that most application-breaking API refactorings can be addressed automatically, and that the performance overhead implied by adapters is acceptable in a large number of adaptation scenarios.

# Acknowledgments

First, I wish to thank my supervisor Prof. Uwe Aßmann for his support, guidance and patience while supervising my work. Our collaboration started in 2002 in Linköping, Sweden, where Uwe supervised my master thesis and the follow-up work in the EASYCOMP project. While moving back to Germany in 2004, Uwe offered me to follow him for a doctoral position at TU Dresden; I did not hesitate a second when accepting his offer and Uwe helped me a lot not to feel sorry about this decision later on. Thank you Uwe!

I am indebted to Michael Rudolf for his indispensable help in developing most (if not all) aspects of the refactoring-based framework adaptation. While supervising Michael's bachelor, internship and master theses, I learned from him at least as much as he learned from me. Michael co-authored all scientific publications related to my thesis and heavily influenced all major conceptual and technological decisions taken. Michael was also the main developer of the Comeback! tool, in the implementation of which he showed to be the greatest Java programmer I have ever met. Finally, he delivered extremely valuable comments on early, late, and the final drafts of this document. A big thanks to you Michael, without you this thesis would have been impossible!

I was fortunate to work with Sebastian Götz, who helped a lot in performing the thesis' case study, in implementing the Comeback! tool, and in co-authoring several related publications.

It was an honour for me to have Prof. Friedrich Steimann as the second reviewer of the thesis. I thank him very much for finding time in his overbooked calender to scientifically x-ray my thesis and to kindly suggest its improvements.

I want to thank our research group for their feedback on my ideas, papers, and the thesis document. Jakob Henriksson, whom I shared an office with, gave important insights on early drafts of the thesis and saved a lot of my time by providing guidance during the tricky bureaucratic process of thesis submission. Thanks to Andreas Bartho, Birgit Demuth, Jendrik Johannes, Mirko Seifert, and Christian Wende for their invaluable comments on the written text of the thesis and its related papers. Many thanks to Steffen Zschaler, who not only commented on our major papers published, but also provided important hints on how to improve the thesis structure and content. Special thanks to Florian Heidenreich and Sven Karol for tolerating me in the FeasiPle project during the last months of my thesis writing.

I want to mention Danny Dig, whom it always was a great pleasure to meet with. Danny has been a convinced proponent of the refactoring-based adaptation, and his feedback on my work as well as his own contributions to this research field have been of great value in forwarding my research.

Very personal thanks on Ulrich and Uta Bergk for their enormous help in solving many routine problems I would otherwise have had to solve and, especially, for their extremely encouraging friendly support.

I thank my friends Serghei "Куманёк" Ghelici, Alexei "Лёлик" Mihailov, Adrian Pop, Vladimir "Вовец" Sterpu, and the whole Stroikins family for visiting me and for inviting me to visit them, regardless of the country borders and distances between us. Although our meetings could hardly be considered scientific, the positive energy I accumulated during those visits helped me a lot in my further work.

I thank my parents Nicolae and Ala for always supporting me and inspiring me to bring out the best in myself. Specifically in hard times during my graduation, I appreciated very much their interminable confidence in my abilities to achieve my final goal.

I am especially thankful to my wife Natasha for being an extraordinary wife and mother. Without her love and support, I never would make it through. I thank her and my children Yuri and Ala for filling my life with a lot of reasons to live and to love.

Finally, but not lastly, I am thankful to God for granting me the skills to achieve this accomplishment.

# List of Publications

This thesis is partially based on the following peer-reviewed publications:

- Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 171–180, New York, NY, USA, 2008. ACM.
- Ilie Şavga and Michael Rudolf. Refactoring-based adaptation of adaptation specifications. In *SERA '08: Best Papers Selection Proceedings of the 6th Conference on Software Engineering Research, Management and Applications*, volume 150 of *Studies in Computational Intelligence*, pages 189–203. Springer Verlag, 2008.
- Ilie Şavga, Michael Rudolf, and Sebastian Götz. Comeback!: a refactoring-based tool for binary-compatible framework upgrade. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 941–942, New York, NY, USA, 2008. ACM.
- Ilie Şavga, Michael Rudolf, and Jan Lehmann. Controlled adaptation-oriented evolution of object-oriented components. In C. Pahl, editor, *IASTED SE '08: Proceedings of the IASTED International Conference on Software Engineering*. Acta Press, 2008.
- Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 175–184, New York, NY, USA, 2007. ACM.
- Ilie Şavga and Michael Rudolf. Refactoring-driven adaptation in evolving frameworks. In *WRT '07: Proceedings of the 1st Workshop on Refactorings Tools*, 2007. Printed as the TU Berlin Technical Report ISSN 1436-9915.
- Ilie Şavga, Michael Rudolf, Jacek Śliwerski, Jan Lehmann, and Harald Wendel. API changes—how far would you go? In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 329–330, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

# Contents

# List of Figures

# List of Tables

# Listings

*"[C]omponent producers should document the changes in each product release in terms of refactorings. Because refactorings carry rich semantics (besides the syntax of changes) they can serve as explicit documentation for both manual and automated upgrades. [...] migration tools should focus on support to integrate into applications those refactorings performed in the component."*

–Danny Dig and Ralph E. Johnson
*The role of refactorings in API evolution [DJ05]*

# 1 Introduction

As they decrease the development and maintenance costs of software systems, software components play an indispensable role in software reuse [Szy98]. Component developers design components for reuse by hiding irrelevant implementation details behind Application Programming Interfaces (APIs) of the components. Application developers use component APIs to assemble components into final applications. Thereby application developers are liberated not only from writing component code, but also from its maintenance.

However, in case a number of applications of a certain application domain have a similar architecture, code reuse may be not enough. In such cases, one may also want *design reuse* [GHJV95, p. 27] to share architectural decisions among similar applications. This is what software frameworks allow for. "A [software] framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes" [JF88]. It is a skeleton that prescribes the applications' overall structure, relation between their subparts, and the applications' thread of control. To instantiate an object-oriented framework, developers write *plugins*—modules that address particular customer requirements and constitute together with the framework a final application (Figure 1.1). By reusing the framework, application developers are liberated from implementing design decisions encapsulated in the framework and can focus on particularities of the application in question tailoring the framework to address application requirements [JF88].

As opposed to other software components, which are usually called from the code written by application developers, a framework often plays the main coordinating role in the final application by calling back plugin code. In such cases, the framework defines a generic algorithm, which is tailored by the application developers in their plugins. This *inversion of control* [JF88] enables high framework extensibility and fosters framework reuse.

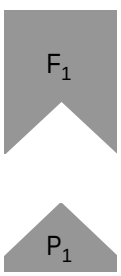



Figure 1.1: Framework-based application development. By implementing a plugin $P_1$, application developers tailor the framework $F_1$ to customer-specific needs.

In general, there are two types of object-oriented frameworks: white-box and black-box frameworks. A white-box framework is reused by inheritance: application developers subclass API classes and provide application-specific implementation in subclass methods. Therefore, to develop a new application, its developers need to understand how the framework is implemented, while to maintain an application they need to understand and maintain existing application-specific subclasses. By contrast, a black-box framework is reused by object composition: it is instantiated by a set of application-specific components, which obey a particular communication protocol and encapsulate their implementations behind external interfaces. Such components can be reused by plugging them together without worrying about their implementation [JF88].

Since black-box frameworks require less knowledge about the framework implementation, they are easier to reuse than white-box frameworks. In ideal cases, a black-box framework can be reused by domain experts (non-programmers) by applying a set of associated tools to interactively compose framework components [RJ97]. However, at early stages of framework development it is usually not clear how to appropriately define the communication protocol of a black-box framework and the interfaces of its components. Therefore, framework development usually starts with a white-box framework, which evolves gradually into a black-box framework along with an increasing understanding of the system design. As Johnson and Foote argue: "Ideally, each framework will evolve into a black-box framework. However, it is often hard to tell in advance how a white-box framework will evolve into a black-box framework, and many frameworks will not complete the journey from skeleton to black-box frameworks during their lifetimes" [JF88]. The main implication of this evolution lifecycle is that, in the same framework, some framework API types may be reused by object composition (in the following called black-box framework reuse) and others by inheritance (in the following called white-box framework reuse).[1]

As any software component, frameworks are software artifacts, which change considerably during maintenance, because, according to Lehman's first law, "a large program that is used undergoes continuing change or becomes progressively less useful" [LB85, p. 250]. According to Swanson [Swa76] and more recent data from Seacord et al. [SPL03, p. 5], the latter based on empirical investigations of maintenance activities [MM83, NP90, Vli00], most maintenance changes are *perfective*. Such changes improve the software products by addressing new requirements, enhancing performance, or usability [II06]. Comparing with other changes—corrective (e.g., fixing bugs), adaptive (e.g., migrating to a new implementation platform), and proactive (e.g., improving the future component reliability) [II06]—perfective changes account for about 60% of all maintenance changes in successful software components [Swa76, SPL03]. To summarize these findings, a successful component is always a candidate for evolution; citing Foote and Opdyke, "success sets the stage for evolution" [FO95]. A successful framework, as any successful component, is not an exception and is constantly evolved during its lifecycle.

Ideally, the API of a software framework should stay stable during framework evolution. In practice, however, the API often changes during the evolutionary lifecycle. As Gamma et al. point out [GHJV95, p. 27], "because applications are so dependent on the framework for their design, they are particularly sensitive to changes in framework interfaces. As a framework evolves, applications have to evolve with it." Otherwise, any change that affects the framework API may be *backward-incompatible* with existing plugins. When a new framework version is released replacing a previously existing version (*framework upgrade*), such changes invalidate existing applications; that is, plugin sources cannot be recompiled or plugin binaries cannot be linked and run with a new framework release. When upgrading to a new framework version, developers are forced to either manually adapt plugins or write update patches. Both tasks are usually disruptive in the middle of the development cycle, error-prone, and expensive—the costs often becoming unacceptable in case of large and complex frameworks.

In general, to reduce the cost of component upgrade, a number of authors propose to, at least partially, automate component adaptation [BTF05, CN96, RH02, YS97, IT03, KH98, BCP06]. These approaches rely on and require additional *adaptation specifications* that describe compensating changes in a machine-processable way. Developers have to write such specifications using a special language either as a separate

[1] As an important remark, in the rest of the thesis we consider implementing an API interface white-box framework reuse. Since the implemented methods are usually called back by the framework in a certain order and might be connected with different states of the application, such reuse requires much more knowledge of the framework internals than when just composing framework objects in black-box framework reuse. In such cases, although application developers do not see framework internals from the plugin classes they write, they still have to know about those internals.

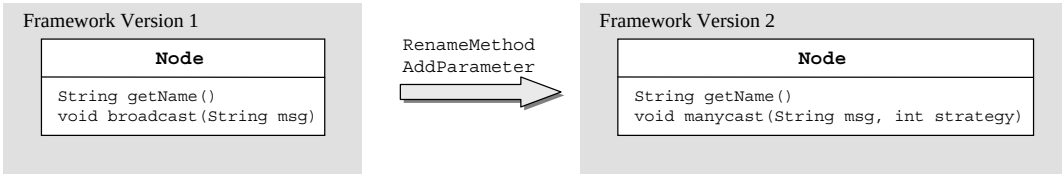



Figure 1.2: `RenameMethod` and `AddParameter` refactorings. To vary on communication strategies besides the default broadcasting, framework developers rename the method *broadcast()* to *manycast()* and provide an integer number indicating a communication strategy as the second method parameter.

document or as annotations in component source code. These specifications are then used by a transformation engine to either intrusively adapt the old application sources or binaries, or to generate adapters that translate between the upgraded components and existing application code.

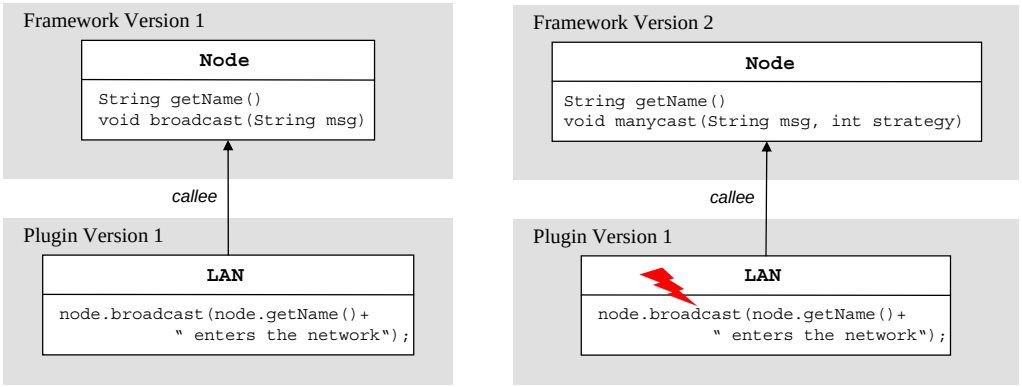
In reality, developers are reluctant to write adaptation specifications. Moreover, once such specifications are provided, component evolution unavoidably demands their maintenance, because evolutionary changes may alter component parts on which existing specifications rely, rendering the latter invalid. For instance, if a component API type that is referred in a specification is renamed after the specification was written, the specification is no longer valid, unless it is updated as well to reflect the renaming. In such cases, specifications must be updated along with the changes of the involved components, which raises the complexity and costs of component adaptation. As the main consequence, while developing and maintaining a software system based on a large number of components (e.g., frameworks), its developers accept a new component version only when it helps to fix an existing problem; otherwise, they prefer to stay with old, often obsolete component versions. This leads to aging of the software system, sometimes even before its first release [SPL03, p. 3].

## 1.1 API Refactorings May Break Existing Applications

To overcome the problem of providing and maintaining adaptation specifications, and to foster proper component upgrade, Dig and Johnson [DJ05] suggest reusing the information about API changes to automatically guide adaptation. They asked the questions: What kind of API changes occur commonly in practice? Could the information about these changes be used to perform adaptation, so that developers are not burdened with the task of creating and maintaining additional specifications? To answer these questions, they investigated the evolution of five big software components, such as Eclipse [Ecl] and Struts [Str]. They discovered that from the point of view of the component most (from 81% up to 97%) application-breaking API changes were *refactorings*.

In his PhD thesis Opdyke introduces refactorings as program transformations that "do not change the behavior of a program; that is, if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same" [Opd92, p. 2]. Refactorings play an indispensable role in software evolution, since, according to Lehman's second law, "as a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it" [LB85, p. 253]. The so-called "bad smells" in code [FBB+99, p. 75], such as duplicated code and large classes, decrease code readability, thereby hamper maintainability. Consequently, they impede software change increasing the efforts and costs required for fixing bugs and making software fulfill new or changed requirements. By eliminating "bad smells", refactorings improve the code structure and, in effect, its maintainability. Common examples of refactorings include renaming classes and class members to better reflect their meaning, moving members to decrease code coupling and increase code cohesion, adding new and removing unused members. Figure 1.2 shows the `RenameMethod` and `AddParameter` refactorings applied to a framework API class, resulting in a refactored signature of an API method.[2]

[2] All thesis examples are inspired by the LAN simulation lab [LAN, DRG+05] and are summarized in Appendix A.




(a) Before API refactorings.

(b) After API refactorings and the framework upgrade.

Figure 1.3: Application-breaking refactorings of a framework API method. While before the refactoring, the plugin class *LAN* could invoke methods *getName()* and *broadcast()* on an instance of the framework class *Node*, after the refactoring the method *manycast()* refactored from *broadcast()* via `RenameMethod` and `AddParameter` cannot be located from the old *LAN*.

If the refactorings of a component preserve its behavior, why are they breaking existing applications developed with an earlier version of the component? The reason is indicated by Dig: "There is a mismatch between the closed-world paradigm of refactoring engines and the open-world paradigm where components are developed at one site and reused all over the world" [Dig07, p. 2]. For refactorings, it is generally assumed that the whole source code can be accessed, analyzed and, if needed, modified. For instance, when renaming a method, all its call sites need to be found and updated to reflect the change. As another example, a method can be deleted only when it is not used. However, if applications developed with the component being refactored are not available for analysis and update, they may be broken by the component's refactorings.

In the context of framework-based applications, where plugins are developed typically by third parties, the mismatch pointed out by Dig is caused by the fact that plugins are usually not available at the time of framework refactoring. As a consequence, plugins are not analyzed and updated correspondingly. Figure 1.3 shows an application-breaking execution of the `RenameMethod` and `AddParameter` refactorings on a framework API method used in a plugin, the latter not available at the time of framework refactoring.

Concluding their case study, Dig and Johnson suggest to document, whenever possible, evolutionary API changes by refactorings and to build refactoring-aware migration tools supporting safe component upgrade [DJ05]. Since refactorings are formally defined, the refactoring history of a component can be treated as a formal specification of its evolution representing a kind of *maintenance delta* [Bax92] to be used for automatic software construction and maintenance.

## 1.2 Thesis Research Questions

Since API refactorings account for most application-breaking changes of framework APIs, a technology that automatically solves problems caused by API refactorings would considerably decrease the costs of adaptation required for a framework upgrade. In addition to saving costs, such technology would also improve the quality of framework upgrade by eliminating errors often made when adapting plugins manually. Furthermore, it would also relax the constraints on the permitted API changes, extending developers' options for appropriate framework evolution.

The main goal statement of this thesis is, therefore, *to use the information about API refactorings for rigorous and practical framework upgrade.*

First, we want to be rigorous by defining precisely the intended meaning of our adaptation. For an application-breaking API refactoring we should be able to define *how* the compensating adaptation action is performed and *what* the result of this action is. In doing that, we must consider all particularities of framework-based applications associated with black-box and white-box framework reuse. In addition to considering particular application-breaking API refactorings, we should also consider the adaptation of a number of such refactorings that occurred between two framework releases as a whole.

Second, we want to be practical by addressing the main technical requirements of both framework and plugin developers. Framework developers require a powerful and reliable adaptation technology that does not place additional strain on them, while compensating for most application-breaking API changes. Plugin developers want adaptation that permits easy framework upgrade and does not require plugin recompilation. An important consequence of the latter requirement is that we must support *binary backward compatibility* of the upgraded framework and existing plugins—the latter should link and run with the new framework release without recompiling.[3]

To support our main goal statement, we will answer the following research question:

*How can we use the information about framework API refactorings to foster binary backward-compatible framework upgrade?*

This main question can be detailed to the following four groups of questions:

- **[Group 1]** Can we expect that other frameworks evolve similarly to the ones investigated by Dig and Johnson [DJ05], making refactoring-based adaptation feasible in general? What are the circumstances affecting the applicability of refactoring-based adaptation? Which API refactorings can break existing applications and how?
- **[Group 2]** In the context of framework API evolution via refactorings, how should an adaptation technology supporting binary backward compatibility be defined for application-breaking API refactorings, and what is the intended result of such technology? What are the main requirements and limitations of the adaptation technology?
- **[Group 3]** What is the correlation between the semantics of refactorings and the adaptation required? How can this correlation be formally defined and used to perform the adaptation automatically?
- **[Group 4]** Given such a formal approach for refactoring-based adaptation, how can it be realized in a practical way?

In this thesis, we will answer these four groups of questions and, hence, our main research question.

## 1.3 A Refactoring-Based Approach to Support Binary Backward-Compatible Framework Upgrade

Each of our four groups of research questions is answered concisely in one of the following four subsections (Sections 1.3.1–1.3.4) and in detail in one of the four main thesis chapters (Chapters 2–5).

[3]The concept of release-to-release binary backward compatibility was introduced by Forman et al. [FCDR95] in the context of library evolution. They distinguish furthermore between API compatibility, meaning applications recompile successfully, and ABI (for Application Binary Interface) compatibility, meaning applications continue to run successfully [FCDR95]. Although we are primarily interested in the latter, in the rest of this thesis we stay with the more common concept of API backward compatibility, specifying precisely binary issues involved whenever necessary.

### 1.3.1 Re-Exploring the Frontiers of Refactoring-Based Adaptation

To answer the questions of Group 1, we learn more about application-breaking API changes in general and application-breaking API refactorings in particular. Considering application-breaking API changes, we analyze how their intent impact on the probability of refactorings to occur in evolving framework APIs. Knowing the intent of API changes and its relation to the ratio of API refactorings, we reason about the general applicability of refactoring-based adaptation for a large number of frameworks. Considering application-breaking API refactorings, we analyze how exactly they break existing applications and reason about the appropriate adaptation means.

To anticipate the feasibility of refactoring-based adaptation, using two frameworks we define a case study of API evolution, which is, according to the definition of Yin [Yin94, p. 38], *critical*—the study's unique conditions permit to generalize its results to a larger number of cases. In our case, the conditions are unfavorable for API refactorings to occur, because the developers of the two frameworks were not restricted in the way they evolved framework APIs and could apply API changes without preserving backward compatibility with existing applications. Arguably, in such cases framework developers may change framework APIs in a more drastic way than via refactorings. However, if the API evolution of such frameworks are similar to the findings of Dig and Johnson [DJ05], we can expect the broad applicability of refactoring-based adaptation.

For all API changes detected in our case study, we analyze why they were applied by framework developers, and attempt to model these changes as refactorings, whenever possible. Our main findings show that, even under such circumstances of unconstrained API evolution, when framework developers want to *improve* the framework, in the investigated frameworks more than 85% of application-breaking API changes can be modeled by refactorings. However, when framework developers *replace* the framework with a new framework version written from scratch, the ratio of refactorings is insignificant. In general, the results of our critical case study permit us to stipulate a general feasibility of refactoring-based adaptation for a large number of evolving framework APIs.

As a secondary contribution of our case study, we provide a discussion of the mechanics of the problems introduced by application-breaking API refactorings. In general, an API refactoring may lead to (1) Non-localizable Functionality (e.g., a renamed or moved API method cannot be located from a plugin), (2) Missing Functionality (e.g., a deleted API method is expected in a plugin), (3) Unintended Functionality (e.g., a plugin method is called when a framework method is expected), or (4) Type Capture (e.g., a plugin type is instantiated and used when a framework type is expected). This classification helps us to better understand the adaptation means required for each group, that is, to locate the non-localizable functionality, to insert the missing functionality, and to protect from the unintended functionality and from the inadvertently instantiated types.

Considering our findings, we decided for adapters (i.e., applications of the Adapter design pattern [GHJV95, p. 139]) as unintrusive adaptation means to compensate for application-breaking API refactorings (Figure 1.4). In practice, intrusive update of plugins via modifying their sources or binaries may not be possible at all, when plugin sources or binaries are unavailable, or software licenses forbid their change. By contrast, adapters are applicable in a large number of adaptation scenarios, also when intrusive adaptation cannot be accomplished.

### 1.3.2 Designing Adapters for Binary Backward Compatibility in Refactored Framework APIs

Addressing the questions of Group 2, we develop *binary* adapters that translate between existing plugins and the upgraded framework. For an existing plugin, its corresponding adapters are collected in an *adaptation layer* placed between the plugin and the framework (Figure 1.5(a)). The adaptation layer *reconstructs* the old framework API, with which the plugin was developed, and translates between old plugin types and API types of the upgraded framework. From the point of view of the plugin, the adaptation layer looks like the original framework (before the refactoring), while from the point of view of the framework the adaptation layer looks like the updated plugin. Adapted plugins of different versions are provided with their version-specific

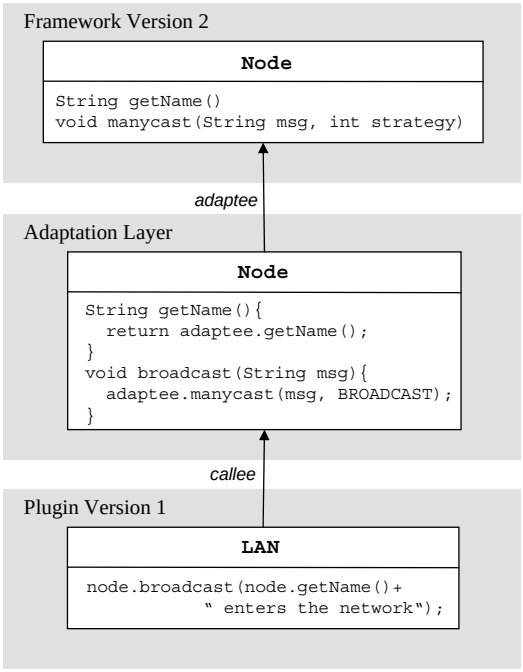
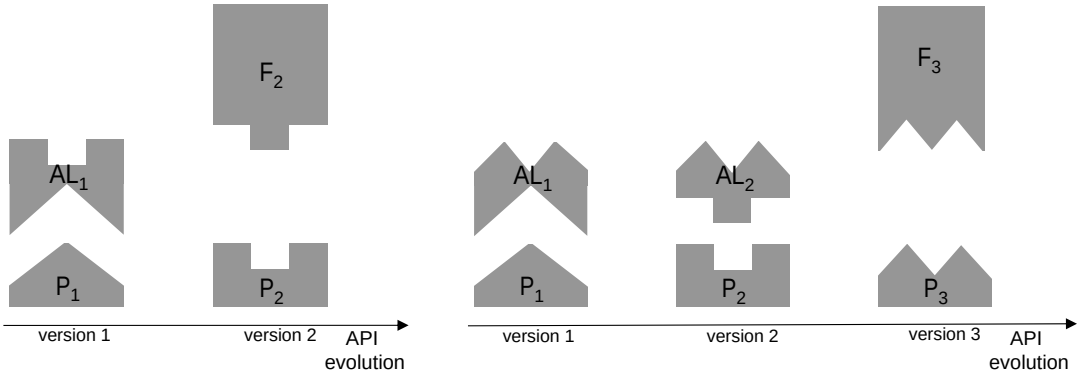



Figure 1.4: Adapter for `RenameMethod` and `AddParameter` API refactorings. The adapter *Node* of the adaptation layer placed between the plugin and the framework uses the API methods of the refactored API class *Node* to serve the calls from the plugin's *LAN*. The BROADCAST parameter used in the adapter is provided at the time of API refactoring as part of the `AddParameter` refactoring.




(a) First framework upgrade. Adaptation layer $AL_1$ reconstructs the API of the framework version $F_1$ and translates between the plugin $P_1$ and the upgraded framework $F_2$.

(b) Second framework upgrade. Adaptation layers $AL_1$ and $AL_2$ reconstruct the APIs of the framework versions $F_1$ and $F_2$, while translating between the upgraded framework $F_3$ and the plugins $P_1$ and $P_2$, respectively.

Figure 1.5: Adapters in refactored framework APIs. Adapters making up adaptation layers *AL* are placed between plugins *P* and the upgraded framework *F*. Since new plugins are always developed with the upgraded framework, they require no adaptation.

adaptation layers (Figure 1.5(b)). Adaptation layers support binary backward compatibility of the upgraded framework and existing plugins—the plugins link and run (via adapters) with the new framework release without recompiling. Moreover, as shown in Figure 1.5(b), adaptation layers work directly with the latest framework version avoiding performance degradation that could otherwise be caused by stacking adapters.

Considering the coupling of the implementation of plugins and the framework, and the possible mixture of black-box and white-box reuse of the same framework, we introduce four variants of the Adapter design pattern: Black-Box Class and Interface Adapter patterns, and White-Box Class and Interface Adapter patterns. To be applicable in refactored framework APIs and to support binary backward compatibility, the patterns distinguish adapting refactored API classes and API interfaces. Black-box and white-box adapters are classes connecting types of old plugins and the latest framework API. While black-box adapters help old plugins to locate the refactored API types of the framework, white-box adapters support the upgraded framework to call back functionality of old plugins. The most sophisticated pattern—the White-Box Class Adapter pattern—supports bidirectional message exchange between old plugins calling the upgraded framework and the upgraded framework calling back old plugins. While they serve as a direct solution for application-breaking API refactorings, the patterns are also extensible to cope with other API changes, which can be adapted by integrating (manually or by reusing other adaptation technologies) the compensating code into black-box and white-box adapters.

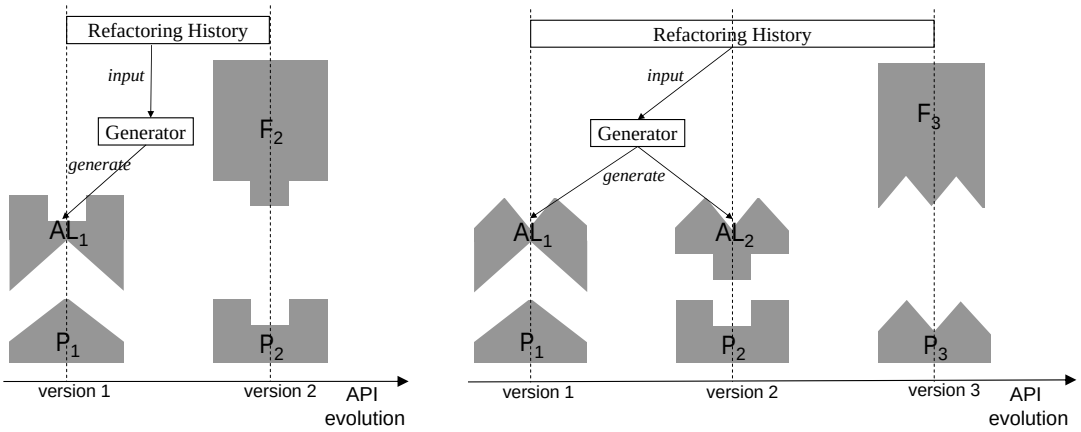
Considering the adaptation of the framework API as a whole, we introduce *exhaustive API adaptation*: both statically and at run time, the possibly refactored types and objects of the framework API, and old types and objects of an existing plugin are completely (exhaustively) separated by a corresponding adaptation layer. Statically, the adaptation layer contains black-box and white-box adapters for all API types, possibly affected by application-breaking API refactorings. At application execution time, depending on the run-time type of the object requiring adaptation, the proper (black-box or white-box) adapters are instantiated. If at run time an object of a type defined in one site (the framework or the plugin) is sent to the other site (e.g., as a method argument), the object is adapted (*wrapped*) by a *wrapper*—an adapter object of the adaptation layer. If the wrapped object is later sent back to the site it originates from, it is obtained (*unwrapped*) from the wrapper.

Considering the encountered challenges, adopted solutions, and limitations, we delimit precisely our adaptation design. In particular, we discuss how we preserve object identity in the presence of adapters, and prescribe how to properly use reflective calls and object serialization to avoid invalidating our adaptation. We also introduce several optimizations to decrease the performance overhead implied by adapters. Finally, we discuss refactorings, which the Adapter design pattern (and, therefore, our variants of the pattern) can or cannot compensate. Examples of the latter are API refactorings operating at an implementation level below method signatures, and refactorings of instance variables declared as public in framework APIs. We explain how to model complex refactorings using a sequence of more primitive adaptable refactorings, so that the complex refactorings can be adapted as well.

### 1.3.3 Formalizing Refactoring-Based Adapter Derivation

Addressing the questions of Group 3, we treat API refactorings as formal specifications of syntactic API changes and use their trace to automatically derive adapters before upgrading a framework (Figure 1.6(a)). The refactoring information required can be logged automatically, for example, in Eclipse [Ecl] or JBuilder [JBu], or harvested semi-automatically [DCMJ06, WD06], and thus does not require additional specifications for most application-breaking API changes [DJ05] from developers. Our adaptation technology is not limited to consecutive framework versions: adapters can be generated for any of the previous framework versions (as shown, for instance, in Figure 1.6(b)).

The main idea in automating our adaptation is to invert all syntactic changes caused by application-breaking API refactorings, as seen from the point of view of a plugin being adapted. However, we do not want to invert API refactorings directly on the refactored framework API, because we want the plugin to use the refactored, improved framework version. Instead, we want to invert the original application-breaking API refactorings on adapters, automatically constructing the latter. However, if we defined for an application-breaking API refactoring its (real) inverse transformation, the latter would be defined in terms of the structure of the API type affected by the original API refactoring. Since adapters have specific members (e.g., adapter constructors and adapter methods), their structure differs from the structure of the API types. As a consequence, in general a real refactoring inverse does not qualify to be executed on adapters.




(a) First framework upgrade. Adaptation layer $AL_1$ is generated for the plugin $P_1$ basing on the trace of application-breaking API refactorings that occurred between the framework versions $F_1$ and $F_2$.

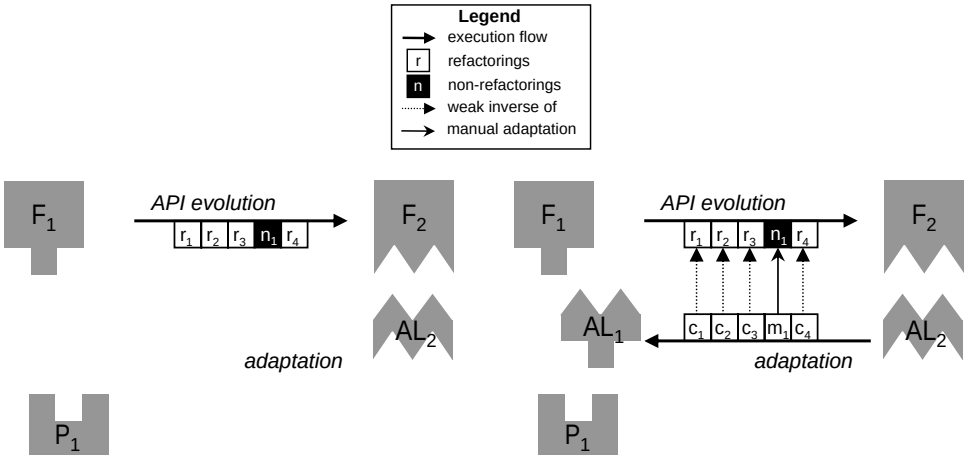
(b) Second framework upgrade. Adaptation layer $AL_1$ is regenerated for the plugin $P_1$ considering all application-breaking API refactorings in the history, while adaptation layer $AL_2$ is generated for the plugin $P_2$ considering only such refactorings occurred between the framework versions $F_2$ and $F_3$.

Figure 1.6: Refactoring-based adaptation (two framework upgrades). Given the latest framework API and the trace of API refactorings that occurred between previous and the last framework versions, adaptation layers are generated, which translate between existing plugins and the upgraded framework. For plugins developed with the upgraded framework, no adaptation layer is required.

Because in fact we are interested in inverting only *syntactic* API changes visible from the old plugin, for each application-breaking API refactoring we formally define its *weak inverse*—a transformation that effectively rolls back the syntactic changes introduced by the refactoring. We call it weak, because its definition considers only certain (i.e., syntactic) program properties, in contrast to all affected program properties as considered by the definition of a real ("strong") inverse. Moreover, not every weak refactoring inverse is applicable on adapters, due to their structural particularities. For example, although one can specify a weak refactoring inverse for a refactored public API field, this weak refactoring inverse would not be applicable on adapters, because the latter do not support public fields. To denote the intrinsic property of certain weak refactoring inverses as being executable on adapters (and used to derive adapters), we call them *comebacks*.

Given the latest framework and the history of API refactorings sorted in the temporal order of their application to the API, we use comebacks to semi-automatically construct adapters before upgrading the framework to its latest version (Figure 1.7). As the first step of adapter generation, we automatically create a *syntactic replica* of the latest framework API—an adaptation layer that mimics the API of the latest framework version and uses that version's API types as adaptees (Figure 1.7(a)). At this stage, the adaptation layer corresponds to the latest API of the framework (i.e., represents exactly the same set of API types), and is not yet useful for old plugins. As the second step of adapter generation, we invert the syntactic changes introduced by API refactorings by executing corresponding comebacks on the adapters of the adaptation layer (Figure 1.7(b)). The execution of comebacks yields the adaptation layer that reconstructs the API of the old framework version and translates between plugins of the old version and the upgraded framework. API changes other than refactorings are addressed manually (as indicated in Figure 1.7), or using other adaptation technologies (e.g., *protocol adaptation* [YS97]) by inserting the corresponding adaptation code into adapters of the adaptation layer.

Although Figure 1.7 uses consecutive framework versions, in general framework versions need not be consecutive: what is important is the difference between the framework's API versions expressed in terms of refactorings. For instance, given the API of the upgraded framework, now in version $F_3$, and the refactoring history between $F_1$ and $F_3$, the adaptation layer $AL_1$ translating between the plugin $P_1$ and the upgraded framework $F_3$ will be derived in the same manner as for consecutive framework versions.




(a) First step. Adaptation layer $AL_2$ is generated as a syntactic replica of the latest framework API of version $F_2$.

(b) Second step. Execution of comebacks yields the adaptation layer $AL_1$.

Figure 1.7: Comeback-based adapter derivation. To a set of refactorings ($r_1$–$r_4$) between two framework versions ($F_1$, $F_2$) correspond comebacks ($c_1$–$c_4$) executed on the adapters of the adaptation layer. The execution order of comebacks is the inverse of the order of the original API refactorings in the history. Changes other than refactorings are adapted manually ($m_1$). The resulting adaptation layer $AL_1$ reconciles the upgraded framework $F_2$ and the old plugin $P_1$.

### 1.3.4 Realizing, Evaluating, and Empowering a Refactoring-Based Adaptation Technology

Considering the questions of Group 4, we develop and evaluate a logic-based tool ComeBack! that generates Java binary adapters using a history of API refactorings and a library of comebacks. The library is developed by encoding comeback definitions as Prolog rules, which operate on the facts representing adapters. The input of the tool is a set of facts about the latest framework API parsed from the framework binaries, and the history of API refactorings. The output of the tool is a set of facts about the adapters derived by comebacks, serialized into executable binary adapters.

The framework upgrade is transparent for a plugin, because the corresponding adapters are generated on the framework site and are deployed together with the latest framework version. To assemble the plugin and the upgraded framework via adapters, while avoiding accidental collisions of existing plugin types and refactored framework types, we separate the name spaces of the framework and the plugin (in Java, by loading framework and plugin types by separate class loaders). In case an application consists of plugins of different versions, we support their *side-by-side execution* [Ang02]: adapted plugins of different versions may co-exist and simultaneously use, together with newly developed plugins, the latest framework version (e.g., assuming that the framework and all plugins of Figure 1.6(b) constitute a single application).

The functionality of ComeBack! permits us to adapt all application-breaking API refactorings detected in our case study and most of such refactorings reported by Dig and Johnson [DJ05] (except for refactorings of public API fields). We used our case study also to test the performance of the adapters generated by ComeBack!. Our performance benchmarking shows an increase of execution time ranging from 1% (in the best case) up to 6.5% (in the worst case) in the adapted applications compared to the same applications before upgrading the refactored framework. We consider such adapter performance overhead acceptable in most adaptation scenarios.

ComeBack! is designed to be independent from a particular Integrated Development Environment (IDE) and is easily combinable with other adaptation tools. Its declarativeness, intrinsic to a logic-based implementation, allows us to combine the tool with the Eclipse refactoring log to reuse the refactoring history automatically recorded in Eclipse. In addition, ComeBack! is flexible enough to allow combination of refactoring-based

adaptation with other adaptation technologies. In particular, we show that it is feasible to integrate *protocol adaptation* [YS97] in ComeBack!.

## 1.4 Thesis Context, Contributions, and Main Limitations

In this thesis we focus on Java frameworks.[4] As a consequence, we limit our discussion to the object-oriented frameworks with single inheritance and single method dispatch. In our adaptation, we support classes, interfaces, and their members that belong to the API (i.e., declared public or protected, and placed in dedicated directories accessible by application developers). Among the Java language features we support are inner classes, enumerations, and static methods. We also support collections and arrays. We do not support public fields in APIs; this should not be a serious limitation in practice, however, since all instance variables ought to be encapsulated by accessor methods to implement information hiding [Par72].

In our adaptation scenarios, we assume that plugins are available neither for analysis nor upgrade. Furthermore, we expect a non-distributed context of application execution. Finally, as one of the main requirements of our technology, our adaptation works in presence of black-box framework reuse (i.e., reuse by object composition), white-box framework reuse (i.e., reuse by inheritance), and the combination of both.

In this context, the main thesis contributions (indexed, to be referred to in the following chapters) are:

**C1: Empirical estimation of the general feasibility of refactoring-based adaptation.** By reconsidering the way framework APIs evolve, we go beyond the case study of Dig and Johnson [DJ05] to estimate the general feasibility of a refactoring-based adaptation technology. The results of our critical case study are affirmative—we stipulate that refactoring-based adaptation should be useful in a large number of evolving framework APIs.

**C2: Insights into the mechanics of application-breaking API refactorings.** By investigating application-breaking API refactorings, we improve the knowledge about their technical aspects that make existing applications break. Based on this understanding, we systematically discuss the means of an adaptation technology required to compensate for application-breaking API refactorings.

**C3: Adaptation design compensating for a refactored API type.** By introducing the Black-Box and White-Box Class and Interface Adapter patterns, we carefully define the structure of adapters that provide binary backward compatibility compensating for application-breaking API refactorings in the presence of black-box and white-box framework reuse, and inversion of control. We show how problems introduced by API refactorings (including the cases, when plugins are not available for analysis) can be solved by our adapters, without recompiling existing applications.

**C4: Adaptation design compensating for a refactored framework API as a whole.** By considering the adaptation of the API as a whole, we introduce exhaustive API adaptation as a set of static and run-time decisions to preserve type safeness and object identity in the presence of adapters.

**C5: Description of challenges, adopted solutions, and limitations in using adapters for refactored framework APIs.** We report on how we solve object schizophrenia, support reflective calls and Java object serialization, and minimize performance degradation in the presence of adapters. Moreover, we discuss why we could or could not compensate for certain application-breaking API refactorings using the Adapter design pattern.

[4] Although we are addressing, besides Java frameworks, different aspects of refactoring-based adaptation of .NET frameworks [ŞR07a, ŞRGA08, ŞRL08], our tool support for .NET is currently much more restricted than for Java. Moreover, the frameworks considered in our case study are also Java-based. Although Java and .NET have similarities in static and run-time type resolution, which we describe to explain the mechanics of application-breaking API changes and the structure of our adapters, we prefer to focus on Java specifics for simplicity. In this thesis, we will not discuss details of the on-going work in adapting .NET frameworks, which can be considered a follow-up work.

**C6: Rigorous refactoring-based adapter derivation.** We formalize the concept of a weak refactoring inverse as a program transformation inverting syntactic changes introduced by refactorings, and formalize comebacks as a special kind of weak refactoring inverses that can be executed on adapters. We show how comebacks can be used to systematically construct adapters, basing on the history of application-breaking API refactorings. For several common refactorings we define and prove corresponding weak refactoring inverses and comebacks.

**C7: Practical tool for refactoring-based API adaptation.** ComeBack! supports the refactoring-based binary backward-compatible upgrade of Java frameworks. Its evaluation shows that the tool compensates for a number of common application-breaking API refactorings, with a fairly acceptable performance overhead implied by the generated adapters. Moreover, its functionality can be enriched to reuse the refactoring logs of development environments, and to integrate adaptation means coping with changes beyond refactorings. Therefore, we believe ComeBack! is practical and can be used in the future in large and complex adaptation scenarios.

It is important to emphasize, that although indeed a number of important application-breaking API refactorings have been identified during the case study and covered by our adaptation technology, completeness of the latter is hard (or, most probably, is impossible) to achieve. Since our technology is limited to adapting only a subset of application-breaking API changes—API refactorings—, remaining changes need to be addressed using other adaptation technologies and tools. Moreover, for certain application-breaking API refactorings it is difficult or impossible to construct the corresponding adapters. For instance, refactorings of public API fields cannot be compensated for by the adapters presented in the thesis.

Furthermore, applying refactorings may subtly change framework behavior, which cannot be observed unless dependent application code is executed. For example, if two API methods of an API class' instance are coupled by a shared state, moving one of the methods to another API class may impact the state sharing. If the framework does not use the method in a way that the shared state influences behavior, moving the method will be considered a refactoring (by the framework developers), although it may indeed change the framework behavior observable from existing application code. In such cases, simply compensating for the syntactic modifications using our adaptation technology may not suffice to preserve dependent application code from malfunctioning. To detect such subtle behavior deviations associated with refactorings requires thorough testing—an indispensable part of any code restructuring activity. Citing Fowler et al.: "If you want to refactor, the essential precondition is having solid tests. Even if you are fortunate enough to have a tool that can automate the refactorings, you still need tests" [FBB+99, p. 89].

Another limitation of our technology is intrinsic to the current state of the art in formalizing and implementing refactorings for commercial programming languages, such as Java. In a number of recent publications [SEM08, SVEM09, ST09], it has been shown that most (if not all) common refactorings are insufficiently formalized, while major industrial refactoring tools are flawed. As a consequence, applying refactorings may introduce errors into programs. Since our adaptation technology is focused on adapting API refactorings and, moreover, is itself based on refactorings (i.e., comebacks), it may inherit any flaws in formalization and implementation of refactorings. The practical application of the technology strongly depends on the quality of formalization and implementation of refactorings for commercial object-oriented languages, such as Java.

## 1.5 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 reports our findings in a case study of unrestricted API evolution. It shows, that even under such unfavorable conditions for API refactorings to occur, when framework developers want to maintain the framework, application-breaking API refactorings detected account for about 85% of all application-breaking API changes. The chapter also presents a taxonomy of API refactorings classified according to the mechanics of problems introduced by API refactorings, and argues for adapters as a general adaptation solution.

Chapter 3 presents in depth the design of our adaptation. Coping with refactorings of particular API types, the chapter introduces the Black-Box and White-Box Class and Interface Adapter patterns as variants of the conventional Adapter design pattern that aim for binary backward compatibility of the refactored framework and existing plugins. Coping with the adaptation of a framework API as a whole, the chapter continues with static and run-time aspects of exhaustive API adaptation. In addition, the chapter describes challenges, adopted solutions, and limitations of our adaptation design.

Chapter 4 discusses how to automate adapter derivation based on the formal semantics of API refactorings. After providing the necessary background information on refactoring formalization, the chapter introduces the concepts of weak refactoring inverse and comeback, the latter as a formal means for refactoring-based adapter derivation. The definitions are exemplified by proofs of weak inverses and comebacks for two common refactorings, while proofs for several other refactorings are presented in Appendix D.

Chapter 5 discusses ComeBack!—a logic-based tool for refactoring-based adapter derivation. Besides highlighting the general tool architecture, the chapter reports on the refactorings currently supported and the performance overhead as measured during the tool's evaluation. In addition, the chapter shows how ComeBack! can be combined with tools for acquiring refactoring history necessary for refactoring-based adaptation as well as extended with technologies for adaptation of changes other than refactorings.

Chapter 6 surveys related work and Chapter 7 concludes and discusses possible directions of future work.

*"By three methods we may learn wisdom: First, by reflection, which is noblest; Second, by imitation, which is easiest; and Third, by experience, which is the bitterest."*

–Confucius (551 BC–479 BC), Chinese Philosopher

# 2

# Re-Exploring the Frontiers of Refactoring-Based Adaptation

The applicability and benefit of refactoring-based adaptation depends directly on the ratio of refactorings in the evolution of an Application Programming Interface (API). To estimate the impact of API changes on existing applications, Dig and Johnson [DJ05] investigated the evolution of five big software components (four frameworks: open-source Eclipse [Ecl], Struts [Str], JHotDraw [JHo], and proprietary e-Mortgage; and the software library log4j [Log]). They discovered that 84%, 90%, 94%, 81% and 97%, respectively, of API changes that could potentially break existing applications could be seen as refactorings from the point of view of the components being changed. In particular, framework-based applications are broken by API refactorings, because plugins are usually developed by third-party companies, and are usually not available for analysis and update at the time of framework refactoring.

Could one expect the change pattern discovered by Dig and Johnson to be similar in other cases of evolving components, and thus anticipate the general feasibility of refactoring-based adaptation? To an extent, in his PhD thesis Dig answers this question, stipulating that "most API breaking changes are small structural changes [...] because large scale changes lead to clients abandoning the component. For a component to stay alive, it should change through a series of rather small steps, mostly refactorings" [Dig07, pp. 33–34]. In other words, if component developers are concerned about backward compatibility with existing applications (and usually they are), they are likely to evolve component APIs through small changes, especially refactorings.

However, as an evolutionary activity,[1] maintenance often conflicts with preserving backward compatibility. Component developers usually face a tradeoff between updating component APIs and not breaking existing applications. The degree to which backward compatibility must be preserved, dictates the range of allowed API changes and varies depending on the development policies. For instance, informal interviews with developers of our industrial partner Comarch [Comb] show that backward compatibility has the highest priority when evolving their commercial framework CDN XL [CDN]. As a consequence, its API has been evolved mostly by additive changes and renamings, for which update patches can be relatively easily specified manually. As another example, although Eclipse is evolved "while remaining compatible with previous versions to the greatest extent possible [...] there are a few areas of incompatibility"[2] reported also by Dig and Johnson [DJ05].

To which extent can maintenance changes, which are not restricted by compatibility preservation, be modeled as refactorings? By answering this question we should be able to reason about the general feasibility of

[1] According to Lehman [Leh96], we consider software maintenance a form of software evolution.

[2] Eclipse help system, Section "Eclipse 3.0 Plugin Migration Guide", Subsection "Incompatibilities between Eclipse 2.1 and 3.0"

refactoring-based adaptation. Should unrestricted maintenance resemble the change pattern reported by Dig and Johnson, we could stipulate its occurrence in a large number of software components, in our case, software frameworks. To foresee the frontiers of refactoring-based adaptation, we address this question by performing a *critical* case study.

According to Yin [Yin94, p. 38], a case study is critical, if its unique conditions permit its results to be generalized (with a certain degree of probability) to a larger number of cases. The conditions are either in favor of, or against, the occurrence of a certain phenomenon under investigation. In the former case, the study results may be used to disprove the stipulated research statement: if it does not hold in favorable circumstances, it will not hold in general. In the latter case, the results are used to validate the statement: if it holds in unfavorable circumstances, it will hold in general [Yin94].

Arguably, framework maintenance that ignores backward compatibility implies unfavorable circumstances for API refactorings to occur. Not being afraid of "clients abandoning the component" [Dig07, pp. 33–34], framework developers might not refrain from changing framework APIs in a more drastic way than possible with refactorings. If the results of such a case study are similar to the ones reported by Dig and Johnson [DJ05], we could expect the broad applicability of refactoring-based adaptation. Otherwise, we should at least be able to reason about circumstances, in which API refactorings are less probable.

Basing on this argument, we defined and performed a critical case study using two open-source Java frameworks. Both frameworks are mature, being in development for more than five years with several major versions released. Their developers were never restricted in changing the framework APIs and performed all intended changes. This chapter brings the following two thesis contributions resulting from our case study:

**C1: Empirical estimation of the general feasibility of refactoring-based adaptation.** In our case study, for each application-breaking API change discovered we analyzed its intent (why developers applied it) and whether this change can be seen as a refactoring, or a series thereof. We show that, when developers aimed to *maintain* the frameworks, our change pattern discovered (refactorings accounting for about 85% of all detected application-breaking API changes) resembles the pattern of Dig and Johnson for more conventional framework evolution [DJ05]. However, when developers wanted to *replace* the framework by implementing its new version completely from scratch, the refactoring ratio was insignificant. In general, the results of our critical case study are affirmative permitting us to stipulate a general feasibility of refactoring-based adaptation for a large number of evolving frameworks.

**C2: Insights into the mechanics of application-breaking API refactorings.** As another result of the case study, we classify application-breaking API refactorings we discovered by the mechanics of the problems introduced and show examples for each problem group. Using this classification, we reason about the adaptation means required to adapt each problem group and argue for adapters as a generally applicable adaptation solution. The classification and the examples introduced will be used consistently throughout the rest of the thesis to explain exactly how our adaptation technology compensates for application-breaking API refactorings.

The rest of this chapter is organized as follows: Section 2.1 postulates the feasibility of refactoring-based adaptation by defining a critical case study of API evolution (Section 2.1.1), reporting the intent of the application-breaking API changes and the refactoring ratio discovered in the case study (Section 2.1.2), and discussing the feasibility of the refactoring-based adaptation for a large number of frameworks (Section 2.1.3). Section 2.2 continues with details on how exactly API refactorings of Java-based frameworks may break existing applications by explaining the necessary basics of Java dynamic linking and method dispatch (Section 2.2.1), introducing running thesis examples of black-box and white-box framework reuse in Section 2.2.1, presenting the taxonomy of application-breaking API refactorings by the mechanics of breaking changes (Section 2.2.3), and discussing adaptation means required to compensate for such refactorings (Section 2.2.4). Section 2.3 summarizes and concludes our empirical investigations.

## 2.1 Empirical Estimation of the General Feasibility of Refactoring-Based Adaptation

To foresee the general feasibility of refactoring-based adaptation, we define a critical case study, in which we not only search and count application-breaking API changes (including refactorings), but also evaluate developers' goals for performing such changes. As a consequence, our case study is, in fact, complementary to the study of Dig and Johnson [DJ05]: their results are more quantitative, while ours are more qualitative.

### 2.1.1 Setup of a Critical Case Study

In their case study, Dig and Johnson collected data in a *document-driven* way using the framework change logs or release notes and, if the documentation was not precise enough, manually comparing old and new framework sources [DJ05]. This way of collecting data is hardly possible for unrestricted framework evolution, when developers typically do not specify detailed release notes or change logs for new framework releases. More appropriate to retrospectively detect changes in such frameworks is *application-driven* change detection, by making plugins of existing applications compile and run with new framework releases. All changes applied to plugins making an application executable again represent the framework change history from that application's point of view.

Since to perform the application-driven change detection we needed old plugin sources, for our case study we required at least two applications developed with one of the older framework releases. Furthermore, we wanted to adapt each application to at least two subsequent framework releases. Therefore, we required at least three major framework releases to be available: the old release, with which the chosen applications were developed, and two subsequent releases.

#### Selected frameworks

It is rarely the case that a mature framework is evolved without, at least partially, preserving backward compatibility with existing applications. One of the most likely exceptions are frameworks used for teaching software technologies. Their existing applications are mainly student exercises that usually do not need to be preserved, so framework developers are unrestricted in evolving the framework APIs. The two frameworks used in our case study (summarized in Table 2.1, where **LoC** stands for lines of code) are of this nature.

**JHotDraw.** As the first framework we used JHotDraw [JHo]—a well-known GUI framework freely available since 2001, used for developing 2D structured editors. As with its predecessor HotDraw originally developed in SmallTalk, JHotDraw's API is extensively documented using design patterns [Joh92], which helped us considerably to comprehend the framework when performing our case study.

Although the API evolution of JHotDraw was also investigated by Dig and Johnson [DJ05], they investigated one big version step from 4.0 to 5.0 using the change log. At the time we were performing our case study, neither JHotDraw 4.0, nor any applications developed with it, were freely available anymore. To perform application-driven change detection, we selected the earliest framework version (version 5.2), for which

| Framework Versions | | API Classes (Methods) | LoC | Applications | API Coverage |
|---|---|---|---|---|---|
| JHotDraw | 5.2 | 129 (1105) | 8294 | 4 (sample) | 36% (15%) |
| | 6.0 | 285 (2737) | 20463 | | |
| | 7.0 | 372 (3201) | 38879 | | |
| SalesPoint | 1.0 | 49 (328) | 3350 | 3 (student exercises) | 18% (21%) |
| | 2.0 | 172 (1509) | 17237 | | |
| | 3.1 | 187 (1651) | 18732 | | |

Table 2.1: Frameworks selected for the critical case study. API coverage reflects the use of API classes (in parentheses, of API methods) by the most advanced application used to evaluate each of the two frameworks.

framework and plugin sources were available. We used four sample clients (JavaDraw, Pert, Net, and Nothing) delivered with the version 5.2. We also used the two major subsequent framework versions 6.0 and 7.0. JHotDraw developers confirmed that all framework versions used in the case study were evolved without preserving backward compatibility.[3]

**SalesPoint.** As the second framework we used SalesPoint [Sal] developed at our department for teaching framework technologies. The framework models the skeleton of a purchase-selling application with such associated concepts as customer, shop, catalog, and data item. To realize application-specific requirements (e.g, to implement a ticket shop), students specialize framework types. The framework itself has been developed by students and the developer team changed often during the development process.

We used SalesPoint versions 1.0, 2.0 and 3.1, as well as two applications (student exercises) developed with the framework version 1.0. During the case study we realized that, because the framework changed considerably between versions 1.0 and 2.0, the two initially chosen applications did not use the part of the API, that was new in version 2.0 and could have changed then in version 3.1. To analyze that API part, we added another application initially developed with SalesPoint 2.0, to the case study set. Because one of the investigators of the case study has been directly involved in the development of SalesPoint, he could affirm that backward compatibility was never considered in the API evolution of the framework.

### Collecting and interpreting the data

Because the change log of JHotDraw was too coarse-grained (most of the changes were undocumented) and there was no change log for SalesPoint, application-driven change discovery was performed manually. Both study investigators had Java programming experience of more than 7 years and good knowledge of refactorings. Moreover, one of the investigators was previously involved in the development and teaching of SalesPoint, while the other was knowledgeable about JHotDraw.

We gradually reconstructed the change history of the framework by manually adapting the plugins of applications developed with the first major framework version to compile and run with the second and third framework versions. Whenever possible, such adaptations were performed by refactorings. In such cases, preserving application behavior was confirmed by comparing the functionality in question of the application running with the first framework release and of the adapted application. In addition, for each application-breaking API change detected we analyzed why it was applied by framework developers. We did not explicitly design a list of change intents from the beginning of the study, but rather evolved it as we learned evolutionary API changes. In such a way, in our manual application-driven change detection we discovered exactly the backward-incompatible API changes affecting the chosen applications, interpreted change intent, and, whenever possible, modeled those changes as refactorings.

The change information obtained was recorded in textual form.[4] Sometimes it was difficult to infer how to model a particular detected change and how to interpret its intent. In these cases, we backed up our interpretation of findings by checking documentation written for application developers and investigating source comments. In some cases, initial interpretation had to be revised and updated after detecting other related changes. In other cases, we needed to separate changes actually breaking applications (e.g., moving an API class) from related preparation changes (e.g., creating a new package for the class being moved). In such cases, we did not count preparatory changes as application-breaking, although considered them to better understand the intent of application-breaking changes. We also did not count changes other than code changes (e.g., changes of XML configuration files in SalesPoint).

It is important to emphasize, that such manual application-driven change detection makes it hard to impossible to reconstruct exactly the same change history twice for the same application. First, adaptation decisions highly depend on the experience of the investigators, in particular, on their ability to model changes as refactorings. Second, some changes can be modeled in different ways, and the decisions often depend on the investigators' modeling taste. Third, adaptation changes applied may depend on previously made

[3] JHotDraw Developer Forum: `http://sourceforge.net/forum/forum.php?thread_id=2121342&forum_id=39886`

[4] The protocol of the case study including detailed metrics of the frameworks and applications used, as well as the log of the recorded changes, is available at `http://comeback.sourceforge.net/protocol.pdf`

adaptation decisions, and the relative order of adaptation changes impacts the discovered change history. Fourth, the knowledge of the frameworks being investigated is important to better understand the overall impact of API changes: we could probably not manage to model many changes as refactorings without a deep knowledge of the two frameworks under investigation.

We did not complement our change detection technique with any tool for refactoring detection (e.g., [Refb, WD06]). Each such tool supports only a certain range of refactorings and we could not anticipate, which changes would be detected in our case study. Moreover, using a tool for refactoring detection would find API refactorings *possibly* affecting applications, while manually detecting the rest of changes would find the API changes *actually* affecting applications. Combining the two would most likely bias our investigation toward a higher ratio of detected application-breaking API refactorings.

Admittedly, by application-driven detection we could not find all application-breaking API changes applied in the two investigated frameworks, because each chosen application used only a subset of the API types of the corresponding framework. Moreover, the same application-breaking API change could repeatedly be detected via different applications. However, our aim was not to find the total number of changes, but rather to understand why and how application-breaking API changes occurred, and to which extent they could be modeled as refactorings.

### 2.1.2 Case Study Results

We were not able to adapt any JHotDraw sample application to use the framework version 7.0. According to the documentation of JHotDraw (packed together with release 7.0.9), this version "is a major departure from previous versions of JHotDraw—only the cornerstones of the original architecture remain. The API and almost every part of the implementation have been reworked to take advantage of the Java SE 5.0 platform." Even for the smallest sample application of 63 lines of code (LoC), its recompilation produced more than 60 errors, most of which could not be adapted manually.

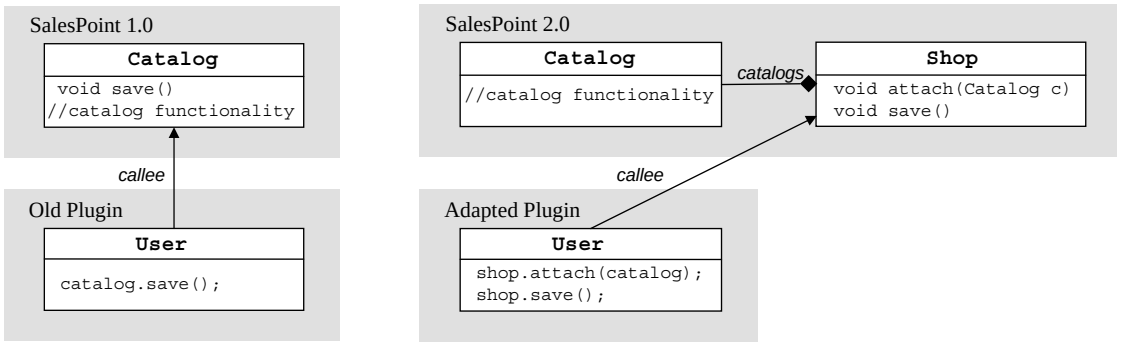
The results for the other JHotDraw and two SalesPoint versions classified by the intent of API changes are summarized in Table 2.2 and discussed in the rest of this section. Our main goal of grouping the detected application-breaking API changes by their intent was to better understand the driving forces of the changes—what caused framework developers to apply them to the framework APIs. For each intent, we present its short description followed by corresponding examples of application-breaking API changes detected and adapted in the case study.

#### Shift of responsibility

A change of this group retracted certain functionality from API classes and assigned it to other API classes. While after the change the functionality was still present in the API, other classes were responsible for providing the functionality than before the change.

| **Change intent** | **JHotDraw** | **SalesPoint** | |
|---|---|---|---|
| | **5.2 → 6.0** | **1.0 → 2.0** | **2.0 → 3.1** |
| Shift of responsibility | 0 | 34 (30) | 1(0) |
| Concept addition | 0 | 23 (23) | 1 (1) |
| Concept refinement | 8 (4) | 94 (74) | 3 (3) |
| Eliminating duplicated functionality | 0 | 6 (6) | 14 (11) |
| Refactoring to patterns | 29 (29) | 5 (5) | 0 |
| Language evolution | 0 | 3 (2) | 0 |

Table 2.2: Types of application-breaking API changes by their intent. For each type, the overall number of detected changes (in parentheses, of refactorings) is specified. In total, API refactorings account for about 85% of all application-breaking API changes.




(a) Before the shift.

(b) After the shift.

Figure 2.1: Shift of responsibility. The method *save()* is moved from *Catalog* in SalesPoint 1.0 to *Shop* in SalesPoint 2.0. To be persisted, in the upgraded framework a catalog has to be attached to the shop.

For example, in SalesPoint 1.0 an instance of *Catalog* (containing purchase items) is saved separately; in SalesPoint 2.0, all catalogs must be attached to an instance of *Shop* (a singleton class representing the shop) that is responsible for catalog persistence (Figure 2.1). Such responsibility reallocation could usually be modeled as refactorings (e.g., moving the method *save()* from *Catalog* to *Shop*, as shown in the transition from Figure 2.1(a) to Figure 2.1(b)) combined with changes of *protocols* (i.e., message exchange [AG97]) between the framework and plugins. Although typically not affecting the actual framework functionality (e.g., the ability to store catalogs), protocol changes introduce permuted message sequences, surplus, or absence of messages between communication parties [BBG+04]. For the example of SalesPoint, the protocol change can be seen as the absence in the existing plugin code (e.g., class *User* in Figure 2.1(a)) of the method call to *Shop*'s *attach()*, to attach a catalog before persisting the whole shop. To adapt plugins, we inserted such method calls manually.

### Concept addition

A change of this group added a new API concept not present in the old API.

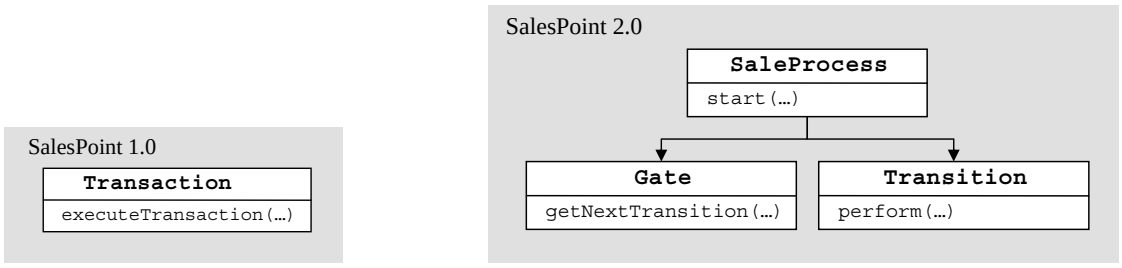
Adding a new API concept usually implied adding new API types and methods, and new parameters to existing API methods. For example, in SalesPoint 2.0, introducing the concept of a shopping cart involved creating an API class *DataBasket*, adding it as a formal parameter to six existing methods, creating an exception type to control that a cart is named correctly, and adding a new method to handle that exception. All these changes could be modeled as refactorings.

### Concept refinement

This group consisted of changes that refined previously existing API concepts to better reflect their semantics and extend their specific functionality.

In some cases, concept refinement involved renaming methods to better reflect their meaning and extracting interfaces to separate them from implementing classes. In other cases, refinement implied changing from Java language types to framework-specific user-defined API types. For example, in SalesPoint 2.0 a new class *NumberValue*, modeling all kinds of numerical values, replaced method parameters and return values of library types *String* and *Integer* in all API methods dealing with such numerical values in SalesPoint 1.0.[5] As another example, in JHotDraw 6.0 the Java *Enumeration* class was replaced in the API by the framework-specific *FigureEnumeration*, that enabled further addition of concept-specific methods, such as

[5] Refactoring called `ReplaceDataValueWithObject` by Fowler et al. [FBB+99, p. 141]




(a) Before the refinement.

(b) After the refinement.

Figure 2.2: Concept refinement. While in SalesPoint 1.0 transactions are supported by a thread-based *Transaction* class, in SalesPoint 2.0 they are supported by a state machine implemented in classes *SalesProcess*, *Gate*, and *Transition*. For simplicity, method parameters are omitted.

*hasNextFigure()*. Consistently with Dig and Johnson [DJ05] we considered such changes refactorings only when the corresponding mappings from old to new types could be provided.[6]

An advanced case of changes, partially going beyond refactorings, was replacing one SalesPoint concept by a different concept with richer functionality (Figure 2.2). In SalesPoint 1.0, the concept of transaction was modeled by *Transaction*—a thread that could be suspended and resumed (Figure 2.2(a)). In SalesPoint 2.0, it was replaced by *SaleProcess* to explicitly model a state machine with states (*Gate*) and transitions (*Transition*) (Figure 2.2(b)). The explicit state machine allowed for a more fine-grained control over the purchase-selling workflow. For instance, one could attach to transitions explicit preconditions controlling the execution of transitions, and attach to gates specific GUI elements displaying the current state of the workflow to the user. Adaptation required "dissecting" *Transaction* to make its states and transitions explicit for the new state machine, and involved a number of refactorings and non-trivial protocol changes.

#### Eliminating duplicated functionality

Changes of this group eliminated API functionality discovered to be duplicated, or very similar, in two or more API classes and methods.

In SalesPoint 2.0, the class *User* serves for authentication and authorization, while its subclass *Customer* mimics the real presence of a customer in a shop at a given point of sale. In the next framework release, since its developers realized that almost all functionality implemented in *Customer* exists in class *SalesPoint*, they removed *Customer* and used methods of *SalesPoint* instead. All but one change involved in adapting plugins could be modeled as refactorings to call the semantically equivalent methods in *SalesPoint*; the exception was a protocol change to call a sequence of three methods in the refactored framework instead of a single initial method in the previous framework version.

#### Refactoring to patterns

Refactoring to patterns [Ker04] is the processes of restructuring existing code to make it follow certain design patterns [GHJV95]. By definition, all changes of this group are refactorings.

For example, class *MenuSheet* of SalesPoint 2.0 was restructured to follow the Composite design pattern [GHJV95, p. 163]—a change that involved a set of refactorings to participating types, such as class and method addition, method renaming, and generalization of argument types.

As another example, in JHotDraw 5.2, *Editor* mediated between the classes *Drawing*, *View* and *Tools* in a realization of the Mediator design pattern [GHJV95, p. 273]. In JHotDraw 6.0, the class *View* does not participate in this pattern anymore. Instead, it is Subject in a newly introduced application of the Observer

[6] Although there is an ongoing research of adaptation in the presence of language evolution [DKTE04, BTF05, KETF07], transformations replacing library types with user-defined types are not yet formally defined.

design pattern [GHJV95, p. 293], with *Editor* observing *View*. Adapting plugins implied their refactoring to call semantically equivalent methods, now located in other classes than in the previous framework release.

#### Language evolution

Changes of this group discovered in our case study were driven by the evolution of Java, when old Java language types were replaced by new, improved Java types.

In SalesPoint 2.0, the parameter type of the *FormSheet* class constructor changed from *Panel* of AWT to *JComponent* of Swing. This change was visible in the API and could not be considered a refactoring. Contrary, changing *Catalog* to collect its items in a *Map* (JDK 1.2), instead of a *Dictionary* (JDK 1.0), was encapsulated and reflected in the API as two refactorings of *Catalog*'s methods (e.g., from *keys()* to *keySet()*). In this case, the changes could be modeled as refactorings mainly because "the functionality of [Enumeration] interface is duplicated by the Iterator interface."[7]

### 2.1.3 Discussion of the Feasibility of Refactoring-Based Adaptation

Although backward compatibility was not considered by the developers of the two investigated frameworks, our pattern of discovered changes (about 85% consisting of refactorings) repeated the results reported by Dig and Johnson for more conventional framework evolution [DJ05]. Maintaining a framework means restructuring it in the first place, and it is this restructuring that often affects existing applications. Most of the other changes are usually additive and do not break existing plugins. Even if considerably changing a framework API (e.g., replacing *Transaction* by *SaleProcess* in SalesPoint, as we discussed in Figure 2.2), developers first analyze the existing framework implementation. Consciously or unconsciously, they perform a mental restructuring of the old design and reuse existing design decisions. Whether the new implementation partially reuses existing code or is written from scratch, this mental restructuring boils down to code that can often be modeled as the refactored old implementation. In other words, because software maintenance requires a good understanding of existing systems, developers think in terms of restructuring and extending existing concepts, even when applying complex changes.

The transition of JHotDraw from 6.0 to 7.0 does not contradict our statement, because it is a case of system *replacement* and not of system *maintenance* [SPL03, pp. 6–7]. Arguably, although developers possess knowledge about the framework being replaced, they develop the concepts and the whole architecture of the replacement system from scratch without mentally restructuring existing design. For such situations, refactoring-based adaptation is probably not feasible.

One could argue that the ratio of refactorings applied depends on how the system architecture is affected: the more of it is changed, the less is the probability of refactorings. However, Tokuda and Batory [TB01] show that even large architectural changes can be achieved by applying a sequence of refactorings. More important is the driving force of the change: if developers want to improve the system (and not replace it, for example), their changes are likely to be refactorings.

Interestingly, our results are echoed by recent investigations of framework evolution [SJM08, DR08]. Similar to us, Schäfer et al. [SJM08] use applications to detect framework API changes; in their case, by automatically comparing old and ported applications. In interpreting the data, they distinguish between a basic refactoring, meaning an operand as a single unit of transformation, and a *conceptual change* with a bigger impact on the system architecture (such as concept addition). For three frameworks and one major version step selected (Eclipse UI framework 2.1.3 to 3.0 [Ecl], JHotDraw 5.3 to 5.4 [JHo], and Struts 1.1 to 1.2.4 [Str]), they show that more than 75% of application changes that accomodate changes in evolving framework APIs are basic refactorings, while others are classified as conceptual changes. However, from our point of view, most of the conceptual changes can also be seen as a sequence of refactorings, as has been shown by Tokuda and Batory [TB01] and discussed by us in Section 2.1.2. In fact, Schäfer et al. themselves use refactorings to explain conceptual changes, for example, replacing method calls to call semantically equivalent new API methods [SJM08].

[7] http://java.sun.com/j2se/1.4.2/docs/api/java/util/Enumeration.html

Instead of analyzing application changes, Dagenais and Robillard [DR08] consider how the framework itself accommodates to the changes of its API. For all old API methods not present in a new framework version, they discover methods that are used internally in the framework as a replacement for old functionality, and interactively suggest choosing these methods to adapt applications. In a case study of the Eclipse JDT [JDT] they show that about 89% of the functionality not accessible through the API of the upgraded framework is still present in the framework. Similar to Schäfer et al. [SJM08] they distinguish between basic refactorings and more complex architectural changes. However, their findings show, that in most cases the functionality required is still provided by the framework, and repeat, in fact, the pattern discovered by Dig and Johnson [DJ05].

Concluding our feasibility study, our findings show that design (and code) restructuring is the intrinsic property of the process of software maintenance. As a consequence, this allows for great optimism that refactoring-based adaptation should be useful in a large number of evolving frameworks.

## 2.2 Insights into the Mechanics of Application-Breaking API Refactorings

Besides exploring the general feasibility of refactoring-based adaptation, the results of our critical case study made us understand precisely how certain framework API refactorings may break existing applications. As a consequence, we could exactly identify the mechanics of the problems introduced by API refactorings and reason about how to adapt such refactorings. In this section, we present a taxonomy of application-breaking API refactorings, discuss their mechanics and show examples of application-breaking API refactorings for each identified problem group. The examples introduced in this section will be used in the rest of the thesis, when discussing appropriate adaptation actions.

### 2.2.1 Background on Java Loading, Linking, and Method Dispatch

To start with, we shall shortly describe relevant issues of the Java Virtual Machine (JVM) [LY99]. Readers aware of the Java dynamic loading, linking, and method dispatch can safely skip this description and continue with Section 2.2.2.

In Java, a type refers to the types and type members it uses by symbolic information embedded in the type's binaries (stored in the type's .class file). For example, if a method of a class (called, in the following, *callee*) is to be invoked from within some class (called, in the following, *caller*), the information about this method is represented in the caller's binaries by the method signature[8] and return value, together with the callee's fully qualified name. The JVM uses this information to *load* the type of the callee—find the type's binary representation and create from it an executable type. In most of the cases, loading of a type is triggered by the type being referenced from some other type, which loading started at some earlier point in time. Each type is loaded by a *class loader* (either the Java default system class loader or a user-defined class loader), with which the loaded type is registered. Thereafter, the type is inserted by the JVM *linker* into the virtual machine so that the type's code can be executed. The main advantage of using symbolic information in the binaries is that the callers do not depend on the physical layout of their callees. If, for instance, a callee's members are reordered and the type is then recompiled, its callers will still be able to find the required methods without being recompiled. Because most of the work of (loading and) linking is done at (load time and) run time, it is called *dynamic linking*.

Besides dynamic linking, similar to other object-oriented languages, Java provides (single) method dispatch: when a method is called on an object (the method receiver), the actual method executed is chosen based on the class of the method receiver.[9] In case the method is not defined in the class, the run-time method lookup finds and invokes the most specific method with the same signature defined by a class' ancestor in the class hierarchy.

[8] In Java, a method signature consists of the name of the method and the set of its formal parameters.

[9] Multiple method dispatch that considers in addition run-time types of method arguments is not supported in Java.

**Note on Thesis Visual Notation**

The visual notation used in most thesis figures is similar to UML class diagrams [Obj05]. However, since we are not concerned about depicting public fields, the field parts of the class boxes are omitted. Moreover, since we operate only with API methods, visibility modifiers are omitted. For the sake of compactness, in case we want to show method implementations, they are specified directly in the bodies of the corresponding methods, and not on separate UML notes. For better visual disambiguation of API classes and interfaces (which is important, when discussing adapter design), we make *italic* the names of API interfaces, but not of abstract API classes. Arrows between class boxes point from the calling site to the callee. Finally, to demarcate the framework, plugin, and (if present in the figure) adapter types, we place the types in separate layers called Framework, Plugin, and Adaptation Layer, respectively.

### 2.2.2 Thesis Running Examples of Black-Box and White-Box Framework Reuse

Now that we briefly introduced necessary Java concepts, we can continue with examples of black-box and white-box framework reuse,[10] which we will use in Section 2.2.3 to show application-breaking API refactorings. The following examples are taken from RefactoringLAN—a collection of examples of framework reuse and API refactorings we define and use throughout the rest of the thesis. For the figures of RefactoringLAN, we use a visual notation similar to UML class diagrams [Obj05] with minor deviations made for the sake of presentation specific to our context. RefactoringLAN is inspired by the LAN simulation lab used at the University of Antwerp for teaching framework technologies [LAN, DRG+05], and is summarized in Appendix A.

#### Examples of black-box framework reuse

In black-box framework reuse, plugin types may act as callers.

Figure 2.3 shows a plugin class *LAN* calling the method *createPacket()* on an instance of a framework class *PacketFactory*. Thereafter, *LAN* asks the obtained packet (of type *IPacket*) about the packet creator node by calling the method *whoCreated()*. Plugin binaries contain symbolic information that refers to the fully qualified names of the framework class and interface, and the signatures and return values of the two framework methods to be called. This information is used in dynamic linking to find and prepare the framework types and methods for execution.

#### Examples of white-box framework reuse

In white-box framework reuse, both the plugin and the framework types may act as callers.

**Plugin caller.** A plugin type may call inherited API methods on an instance of a plugin subclass via a usual method call or a supercall. Figure 2.4 shows a framework class *Node* subclassed by the plugin's *SecureNode*. The framework's *Node* defines a method *getIdentifier()*, the implementation of which is inherited by *SecureNode*. The run-time method lookup will find and invoke the framework implementation of the method in the calls from the plugin's *LAN* and *SecureNode*.[11]

**Framework caller.** A key property of a framework that enables the intrinsic framework ability of inversion of control [JF88] is the ability to call back plugin functionality. To enable tailoring a generic framework algorithm to application-specific requirements, framework developers may declare certain API methods

[10] Recalling our footnote from the Introduction on page 2, we consider implementing an API interface a white-box reuse.

[11] In fact, using the keyword *super* in the example of Figure 2.4 is optional, since the method *getIdentifier()* is not redefined in *SecureNode*. However, we will use this example to discuss particularities of the adapter design, which should take into account possible supercalls from plugin classes to framework classes.

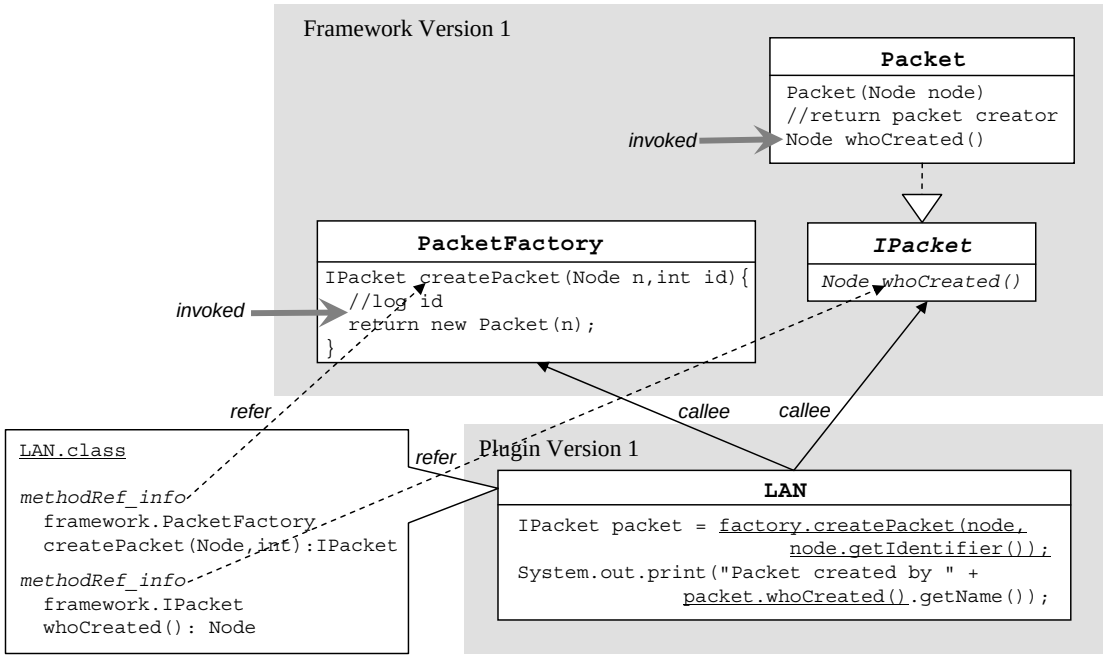



Figure 2.3: Black-box framework reuse: Plugin as caller. *LAN*'s binaries contain symbolic references to the framework's *PacketFactory* and its method *createPacket()*, as well as to the framework's *IPacket* and its method *whoCreated()*.

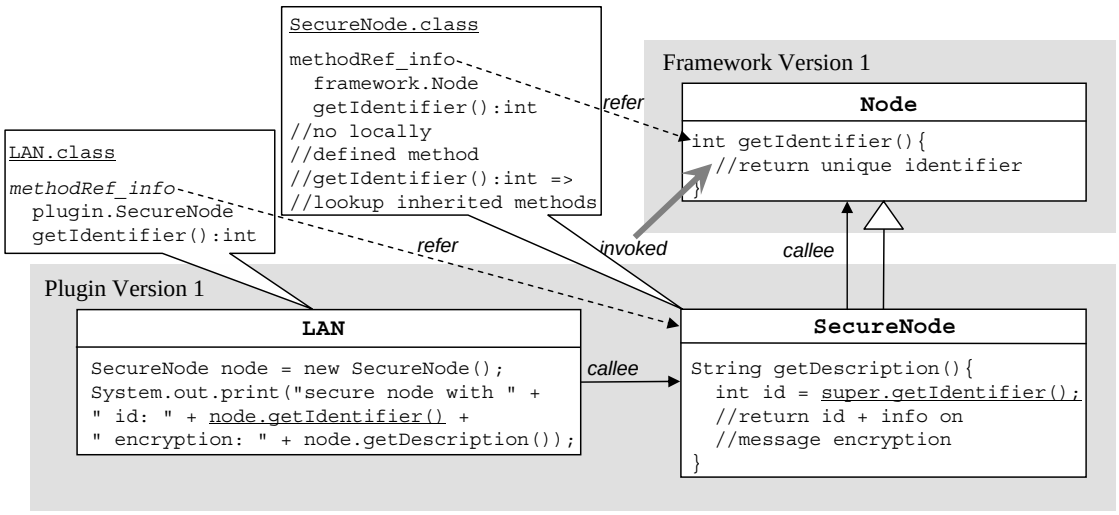
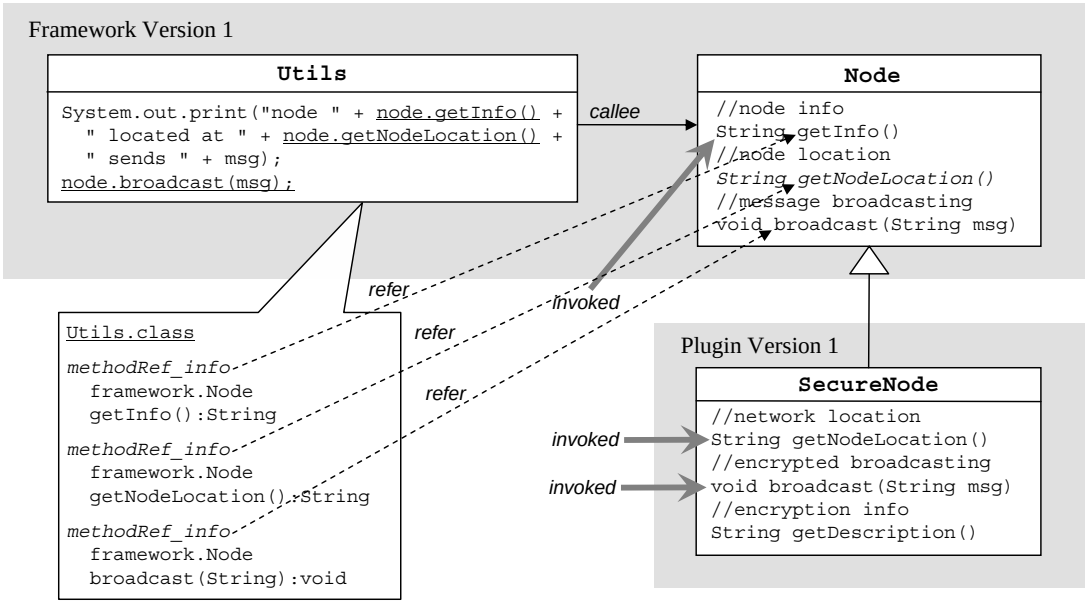


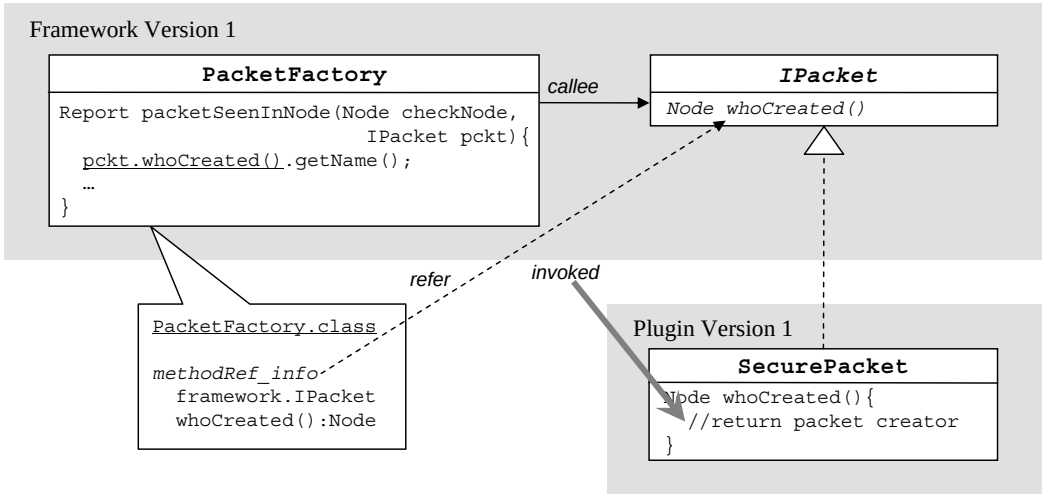

Figure 2.4: White-box framework reuse: Plugin as caller. The plugin's *SecureNode* inherits the method *getIdentifier()* from the framework's *Node*. The inherited method is invoked by the call from the plugin's *LAN* and (using the keyword *super*) by the supercall from the plugin's *SecureNode*.

to be implemented in plugins (so-called *hook* methods) and then call these methods on objects of plugin types.[12] In general, a hook method may be declared abstract (i.e., plugins must define its implementation) or concrete (i.e., with a default implementation plugins may redefine, or override). While for defined or redefined API methods their plugin implementation is invoked by the framework at run time, for other API methods the framework implementation is executed (Figure 2.5).

[12]For simplicity, in our examples we omit details of generic framework algorithms.




(a) Calling API class methods. The framework call to *getInfo()* will execute the framework method implementation, while the calls to the methods *getNodeLocation()* and *broadcast()* will invoke the implementations provided by *SecureNode*.




(b) Calling API interface methods. The framework call to *whoCreated()* will invoke the method implementation of the plugin class *SecurePacket* implementing the *IPacket* interface.

Figure 2.5: White-box framework reuse: Framework as caller. Framework binaries contain symbolic information about the API methods, including those defined or possibly redefined in plugins. At run time, the calls to the API methods are dispatched according to the run-time type of the message receiver, leading to the execution of the framework or plugin method implementations.

In the example of Figure 2.5(a), the framework class *Node* defines three methods: a concrete method *getInfo()* for obtaining node descriptions, an abstract method *getNodeLocation()* for identifying node locations in the network, and a concrete method *broadcast()* for message sending. The framework class is subclassed in a plugin by *SecureNode* that implements *getNodeLocation()* to return the node location and overrides *broadcast()* to encrypt a message before sending. In case at run time an instance of *SecureNode* is sent to the framework's *Utils*, which invokes the three methods on that instance, Java method dispatch will invoke the framework method implementation of *getInfo()*, and the plugin method implementations of *getNodeLocation()* and *broadcast()*.

| Problem | Description | PR | Examples |
|---|---|---|---|
| Non-localizable Functionality | Although functionality required exists, it cannot be located by the caller | yes | Changing API method signatures, moving API methods, renaming and moving API types |
| Missing Functionality | Functionality required by the caller does not exist | yes | Removing API methods, adding abstract API methods |
| Unintended Functionality | Functionality found and invoked by the caller is not the one intended | no | As for Non-localizable Functionality + adding API methods |
| Type Capture | Although the type defining required functionality exists, it is taken over by other type with different functionality | yes | Adding, renaming and moving API types |

Table 2.3: Types of problems caused by API refactorings. The **PR** column indicates, whether plugin recompilation would detect the problems introduced by framework API refactorings.

In the example of Figure 2.5(b), the framework API interface *IPacket* is implemented by the plugin's *SecurePacket*. In case at run time an instance of *SecurePacket* is sent to the framework's *PacketFactory*, invoking the *whoCreated()* method on the instance will lead to the execution of the plugin's method implementation.

### 2.2.3 Classifying Application-Breaking API Refactorings by the Mechanics of Breaking Changes

In general, API refactorings may lead to the caller not obtaining the functionality it requires, which could be obtained before the refactoring. Considering particularities of application-breaking API refactorings observed in our case study (reported in Section 2.1.2) and of Java language dynamic linking and method dispatch (sketched in Section 2.2.1), we identify four groups of problems caused by API refactorings: *Non-localizable Functionality*, *Missing Functionality*, *Unintended Functionality*, and *Type Capture* (summarized in Table 2.3). In the following, we discuss each problem group from both perspectives: as seen from plugin callers and from framework callers.

Some of the problems we discuss were previously reported in the area of component-based software engineering with regard to general API changes of components used in component-based applications [SLMD96, MS98, KL92]. However, we decided to introduce our terminology to precisely delimit the problems specific for the context of framework API refactorings, as opposed to arbitrary changes of component APIs. In our discussion, we refer to existing terminology [SLMD96, MS98, KL92] whenever appropriate.

#### Non-localizable Functionality

"Although functionality required exists, it cannot be located by the caller. Application execution results in linkage errors, while plugin recompilation results in compilation errors."

**Non-localizable Framework Functionality.** This problem may occur, when the plugin is acting as a caller, either in black-box or in white-box framework reuse. In case a framework refactoring is affecting the signature or location of an API method called in a plugin, the dynamic linking will not localize the refactored API method.

For black-box framework reuse introduced by Figure 2.3 on page 25, Figure 2.6 shows the consequences of two API refactorings: (1) `RemoveParameter` applied to the second parameter of the *createPacket()* method defined by the API class *PacketFactory*, and (2) `RenameMethod` renaming the method *whoCreated()* of the API interface *IPacket* to *getCreator()*. For white-box framework reuse introduced by Figure 2.4 on page 25, Figure 2.7 shows the problem caused by the `RenameMethod` refactoring applied to the method *getIdentifier()* of the API class *Node*. Similarly, other refactorings of API method signatures (e.g., adding parameters,

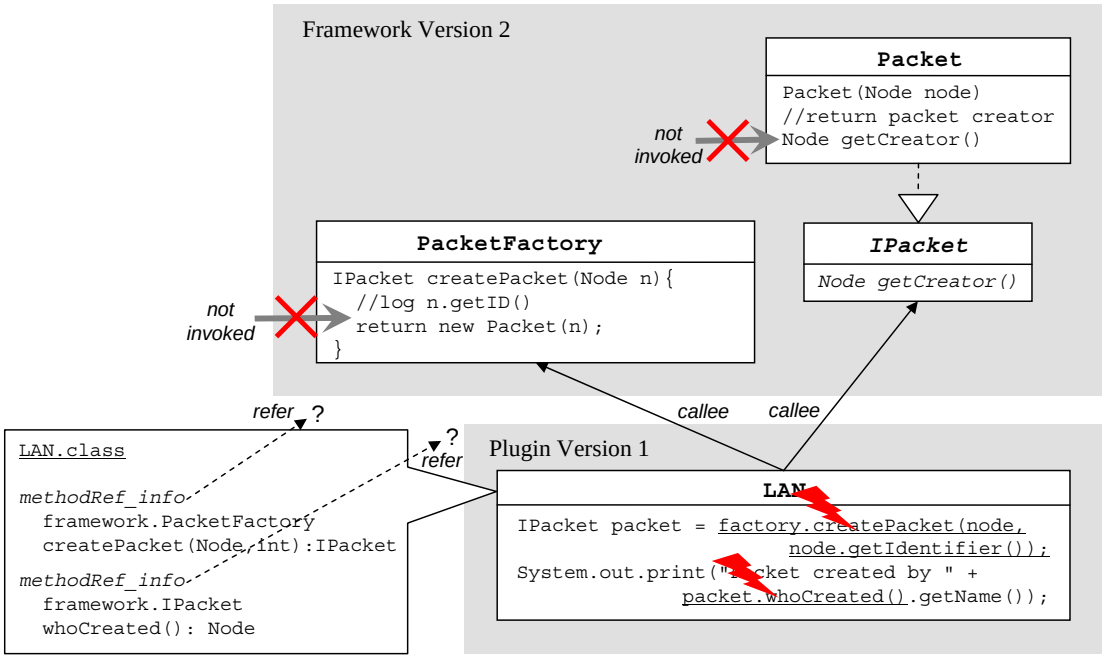



Figure 2.6: Non-localizable Framework Functionality (black-box framework reuse). Since framework developers realized, that the second method parameter of the *PacketFactory*'s *createPacket()* is redundant (the node's identifier can be obtained from the node passed as the first method argument), they removed the redundant parameter. In addition, they renamed *IPacket*'s method *whoCreated()* to *getCreator()*. As a consequence, symbolic method references in *LAN* became obsolete.

changing their types or order) will make dynamic linking unable to locate the refactored API method using old method references in plugin binaries.

Another common example of this problem group, which can be seen as a method signature refactoring from the point of view of existing plugins, is the refactoring that replaces calls to an old API method to call another API method with similar, yet improved functionality. The old method is usually marked as deprecated and deleted in upcoming framework releases. This so-called *deprecate-replace-remove lifecycle* is very common in evolving component APIs, in particular framework APIs. After such refactorings, binaries of the plugins unavailable for analysis and update at the time of framework refactorings will still contain the references to the old API methods, and not the API methods to be called after the refactoring.

Yet another example is the `ExtractInterface` refactoring [FBB+99, p. 277]: a previously existing class is split into an interface (named usually as the class before the refactoring) and a class implementing that interface. In such cases, dynamic linking may be able to find the type's binary representation, but because it finds the interface and not the class (as it expects), dynamic linking cannot locate the implementations of the methods required by the caller.[13]

As a concluding example of this problem group, refactorings changing the method location (e.g., moving methods among classes) lead to dynamic linking not being able to localize the moved methods. Similarly, one can consider refactorings renaming API types and packages as also changing method locations, because such refactorings, from the point of view of existing plugin callers, make methods appear in different (i.e., renamed) types.

In most of the cases, plugin recompilation does not solve the problems of this group and reports compilation errors, because plugin sources still contain obsolete references and the compiler is not able to locate changed types and methods. A notable exception is specializing return type or generalizing parameter type of a method, in which case the compiler is able to properly update plugins (an example of a *binary-incompatible*

[13] In Java, the exact reason of this problem is the different binary code instructions used for calling class and interface methods.

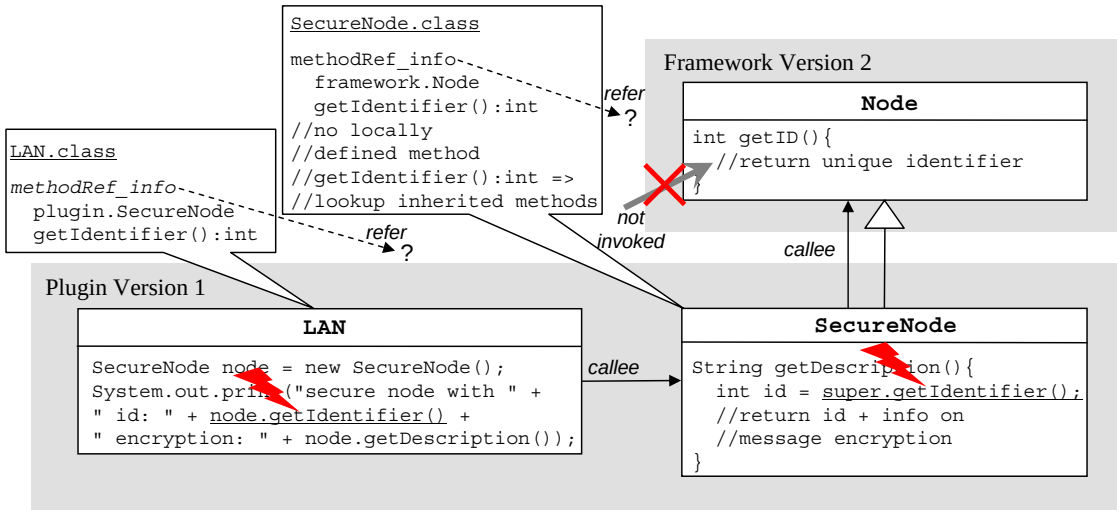



Figure 2.7: Non-localizable Framework Functionality (white-box framework reuse). *Node*'s *getIdentifier()* is renamed to *getID()* and can be located neither from *LAN* nor from *SecureNode*, due to obsolete symbolic references in plugin binaries.

refactoring as opposed to *source-incompatible* refactorings). The aforementioned `ExtractInterface` refactoring can also be solved by plugin recompilation, if plugins do not invoke any constructor of the class being refactored.

**Non-localizable Plugin Functionality.** This problem may occur in white-box framework reuse, when a framework type is acting as a caller. In case an API refactoring affects the signature or location of an abstract hook method, dynamic linking will not find the plugin implementation of the hook method. As a consequence, the callback mechanism will not function properly.

For an example, let us reconsider the framework calling back abstract API methods implemented in a plugin, as we introduced by Figure 2.5(a) for an API class and by Figure 2.5(b) for an API interface. Figure 2.8 shows application-breaking consequences of API refactorings applied to an abstract method of an API class (Figure 2.8(a)) and to a method of an API interface (Figure 2.8(b)). In both cases, the plugin functionality that should be accessible from the framework via the callback mechanism will not be located from the refactored framework, and dynamic linking will fail with a linkage error.

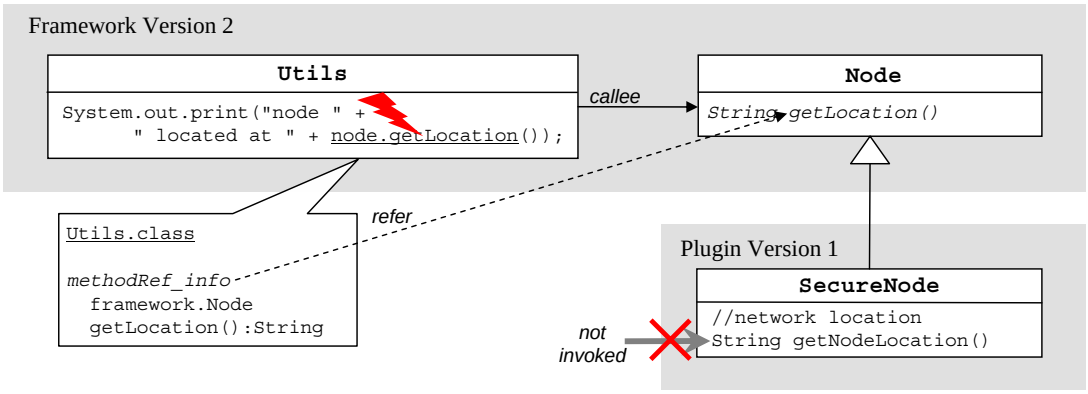
In the context of component API evolution, Steyaert et al. call the problem of invalidating existing subclasses by newly introduced abstract API methods the Unimplemented Method problem [SLMD96]. Analogously to their terminology, by Non-localizable Plugin Functionality: Unimplemented Method we call the problems caused by framework refactorings of existing abstract API methods.

**Effect.** Most application-breaking API refactorings lead to the problems of this group. In general, such refactorings lead to *syntactic mismatch* [BBG+04] between the framework and plugins: although the functionality is fully available, the framework and its plugins do not communicate as intended by developers, due to the different syntactic type representation.

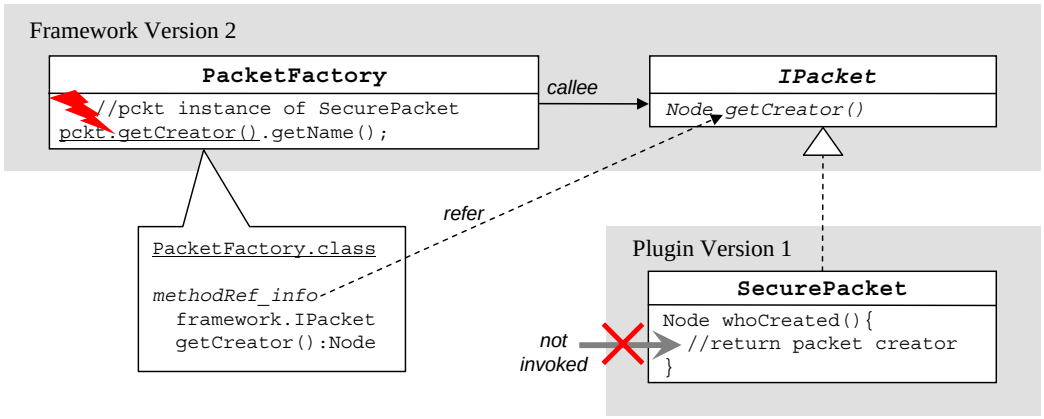
### Missing Functionality

"Functionality required by the caller does not exist. Application execution results in linkage errors, while plugin recompilation results in compilation errors."

**Missing Framework Functionality.** Whereas some authors (e.g., [DMJN07, DNMJ08]) use the term *edit* to denote pure API method removal, we stay with the well-known terminology of Opdyke [Opd92] and Roberts [Rob99], and consider `DeleteMethod` applied to an API method *unused in the framework* also a refactoring. Although the precondition of the `DeleteMethod` refactoring checks that the method is not in




(a) API class refactoring. The signature of the plugin method *getNodeLocation()* does not match the signature of the renamed abstract API method *getLocation()* in the framework's *Node*.




(b) API interface refactoring. The signature of the method *whoCreated()* implemented in the plugin's *SecurePacket* does not match the signature of the renamed API interface method *getCreator()* in the framework's *IPacket*.

Figure 2.8: Non-localizable Plugin Functionality: Unimplemented Method. Framework binaries contain symbolic information about the abstract API methods, the implementations of which are provided in plugins. After upgrading the refactored framework, since the signatures of methods defined in the plugin do not match the signatures of refactored abstract API methods, the plugin functionality will not be located from the framework.

use, in case plugins are not available for analysis an API method still in use may get deleted. The same holds for deleting an API type (a class or an interface). Other refactorings of this group reduce the type and method visibility, although not actually deleting the framework functionality (e.g., the `HideMethod` refactoring [FBB+99, p. 245]).

To emphasize, pure API method deletion is rarely the case in framework APIs; instead, the method to be removed from an API follows a deprecate-replace-remove lifecycle. In these cases, we consider API refactorings that realize the deprecate-replace-remove lifecycle as refactorings leading to Non-localizable Functionality, because the functionality required by plugins is still provided by refactored API methods of the framework, and is not deleted altogether.

**Missing Plugin Functionality.** Introducing a new abstract hook method is application-breaking, because no implementation in plugins exists. This is exactly the example of the Unimplemented Method problem as described by Steyaert et al. [SLMD96]. In contrast to Non-localizable Functionality: Unimplemented Method, in this case the plugin functionality required does not exist at all. If, however, the hook method being added is concrete (i.e., the framework provides a default method implementation), adding the method will usually not break plugins (except for Unintended Plugin Functionality: Method Capture described

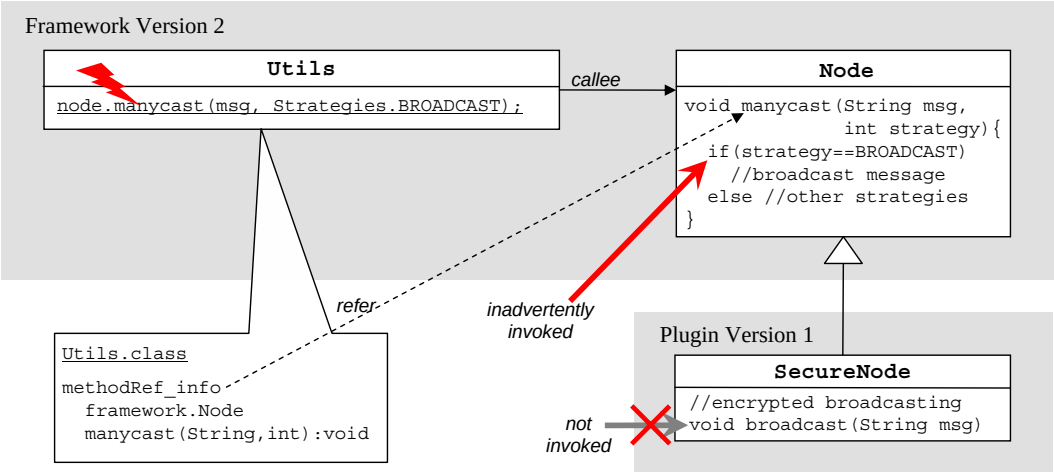



Figure 2.9: Unintended Framework Functionality: Inconsistent Method. In the upgraded framework, the overriding plugin functionality of the *SecureNode*'s *broadcast()* responsible for sending encrypted messages is not located by dynamic linking using the signature of the refactored *Node*'s *manycast()*. As a consequence, the framework implementation of the method is executed in *LAN*, as opposed to the plugin implementation invoked before the framework upgrade.

shortly, on page 33), because existing plugins are not supposed to override the newly introduced hook method.

**Effect.** From the point of view of dynamic linking, Missing Functionality is similar to Non-localizable Functionality, because in both cases dynamic linking is unable to localize functionality expected by a caller. Moreover, in most cases plugin recompilation reports similar problems, because the compiler is not able to figure out, whether some types or methods are simply modified, or are deleted altogether. However, it is important to distinguish the two groups, because they require different adaptation means (locating existing functionality versus defining missing functionality, as discussed in Section 2.2.4).

### Unintended Functionality

"Functionality found and invoked by the caller is not the one intended, possibly resulting in erroneous application behavior. Neither application execution nor plugin recompilation result in errors." This problem group is possible in white-box framework reuse, in the presence of dynamic linking and method dispatch as realized in Java.

**Unintended Framework Functionality.** This problem may occur in white-box framework reuse, when a plugin type acts as a caller. In case an API refactoring affects the signature or location of a concrete hook method, for which an overriding plugin implementation exists, dynamic linking will not locate the overriding plugin functionality. As a consequence, the callback mechanism will invoke the framework functionality, instead of the overriding plugin functionality invoked before the refactoring.

Based on the example from Figure 2.5 on page 26, Figure 2.9 shows a refactored API method: to vary on communication strategies besides the default broadcasting, the method *broadcast()* of the framework's *Node* is renamed to *manycast()* and now accepts an integer indicating a communication strategy, as its second parameter. The existing broadcast call from the framework's *LAN* is updated to call the refactored method and pass the BROADCAST constant as the second method argument indicating broadcasting strategy. After upgrading the framework, since the signature of the overriding plugin method in *SecureNode* remains unchanged, on an instance of *SecureNode* dynamic linking locates the framework method implementation and not the overriding plugin implementation. As a consequence, the method invocation in *Utils* will result in sending an unencrypted message.

Discussing changes of a base class' method leading to certain methods of existing subclasses being invoked less than before the changes, Kiczales and Lamping call the resulting problem of inconsistent application behavior Inconsistent Method [KL92]. Consistently to their terminology, we are using the name Unintended Framework Functionality: Inconsistent Method to denote similar problems caused by refactorings of API methods overridden in old plugins.

**Unintended Plugin Functionality.** This problem may occur in white-box framework reuse, when a framework type acts as a caller. More specifically, the problem occurs in case an API refactoring adds a new API method, or affects the signature or location of an existing API method, while the refactored API method matches the signature of an existing plugin method. As a consequence, dynamic linking will locate and the callback mechanism will invoke the plugin method with possibly different semantics than those of the (intended) refactored API method.

For an example, let us reconsider the API class *Node* subclassed by the plugin's *SecureNode*, as introduced by Figure 2.5 on page 26. Among its methods, *Node* defines the method *getInfo()* returning a node description. In the second framework release, framework developers add a hook method to *Node* enabling its subclasses to include subclass-specific information in node descriptions. By default, this hook method, called *getDescription()*, forwards to *getInfo()*; that is, the two methods are semantically equivalent (Figure 2.10, top right part). In addition, all existing framework method calls to *getInfo()* are updated to use the newly introduced method *getDescription()*.

However, unaware of the plugin implementation, framework developers introduced an accidental match of the method signature of the newly introduced framework method with the existing plugin method *getDescription()*, the latter with different semantics (i.e., returning the encryption used in *SecureNode*). As a consequence, if *Utils* of the upgraded framework invokes the method *getDescription()* on an instance of *SecureNode*, the method invocation will produce an unexpected behavior, obtaining the description of the encryption algorithm as defined in *SecureNode*, and not the description of the node as defined in *Node* (Figure 2.10).

In case a newly introduced method of a base class is accidentally overridden in a subclass, Steyaert et al. call the resulting problems of application misbehavior Method Capture [SLMD96].[14] In the context of framework refactorings, besides the addition of API methods, such problems may also be caused by refactoring existing API methods. In our example, this could also occur, if *getInfo()* was renamed to *getDescription()* (without adding a new API method to *Node*). Analogously to the terminology of Steyaert et al. [SLMD96], we use Unintended Plugin Functionality: Method Capture to denote similar problems caused by refactorings introducing new or modifying existing API methods, accidentally overridden in old plugins after upgrading to the refactored framework version.

**Effect.** In general, any refactoring introducing new, or changing existing, API classes and methods (e.g., adding and extracting methods, changing method signatures, moving methods) may lead to the well-known *fragile base class problem* [MS98], in which modifications to a component's base API class reused by inheritance may cause derived classes to malfunction.[15] In such cases, the syntactic mismatch caused by refactorings cannot be detected and solved by subsequent plugin recompilation. If the plugins were available for analysis and update, a refactoring engine could warn about the accidental method overriding, or could update signatures of accidentally overriding plugin methods. However, because plugins are usually not available at the time of framework refactoring, the problems of this group remain undetected and may lead to malfunction at application execution time.

[14] As a notable difference to Java, Method Capture is not possible in C#, in which hook methods must be declared (using *virtual*) and overridden (using *override*) explicitly. As a consequence, the language execution environment is able to avoid accidental overriding. For an appealing and concise discussion of these features of C#, we refer to an on-line conversation with Anders Hejlsberg by Bill Venners with Bruce Eckel at `http://www.artima.com/intv/nonvirtualP.html`. In Java, a good programming practice is to make non-hook methods *final*, and concrete hook method *protected*, while marking overriding methods with the attribute `@override`. However, these guidelines can be neither enforced nor checked by the language itself, and are often neglected in practice.

[15] Originally, the fragile base class problem was described in the context of C++ programs. In this thesis, in the line of Mikhajlov and Sekerinski, we use "the term fragile base class problem in a wider sense, to describe the problems arising in separately developed systems using inheritance for code reuse" [MS98]. While in their seminal paper [MS98] Mikhajlov and Sekerinski focus on *protocol changes* [YS97] in a base class, in this thesis we focus on refactorings.

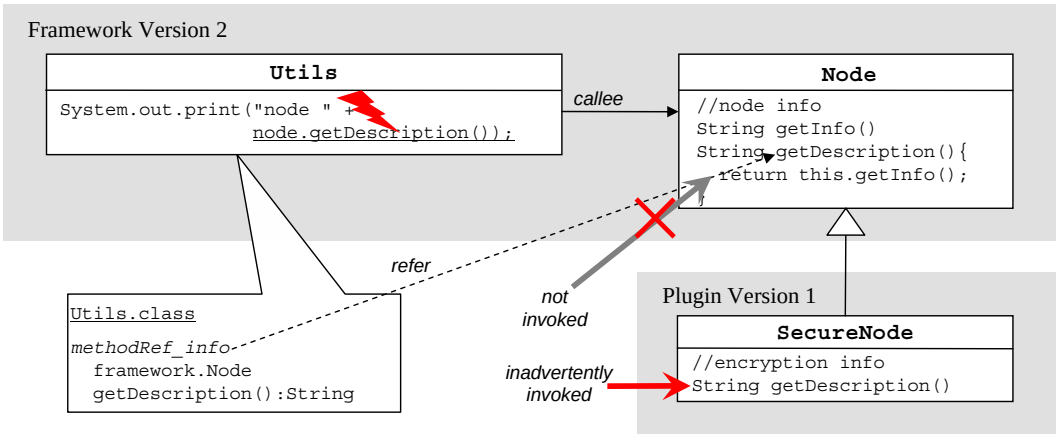



Figure 2.10: Unintended Plugin Functionality: Method Capture. The newly introduced method *getDescription()* in *Node* is semantically equivalent to *getInfo()*. However, the new method is accidentally named exactly as the *getDescription()* method of *SecureNode*, whereas the two methods have different semantics. As a consequence, instead of the node name as obtained before the refactoring, for an instance of *SecureNode* its encryption description is indeliberately retrieved by *Utils* of the refactored framework.

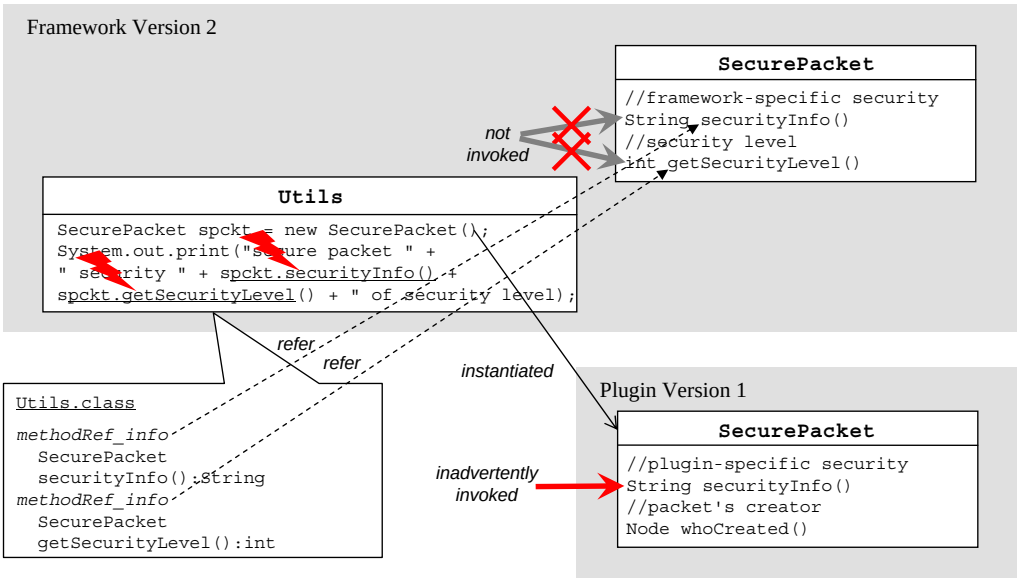
### Type Capture

"Although the type defining required functionality exists, it is taken over by other type with different functionality. Application execution may result in linkage errors or erroneous application behavior, while plugin recompilation results in compilation errors." This problem group may occur in both black-box and white-box framework reuse in the presence of dynamic linking. Analogously to Method Capture [SLMD96], in which a method is accidentally taken over by another method, we introduce the term Type Capture to refer to similar problems caused by a type accidentally taken over by another type. While Type Capture can be seen as a special case of Non-localizable and Unintended Functionality caused by renaming existing and introducing new API types, it requires specific adaptation means (i.e., to avoid an intended type being taken over by some other type).

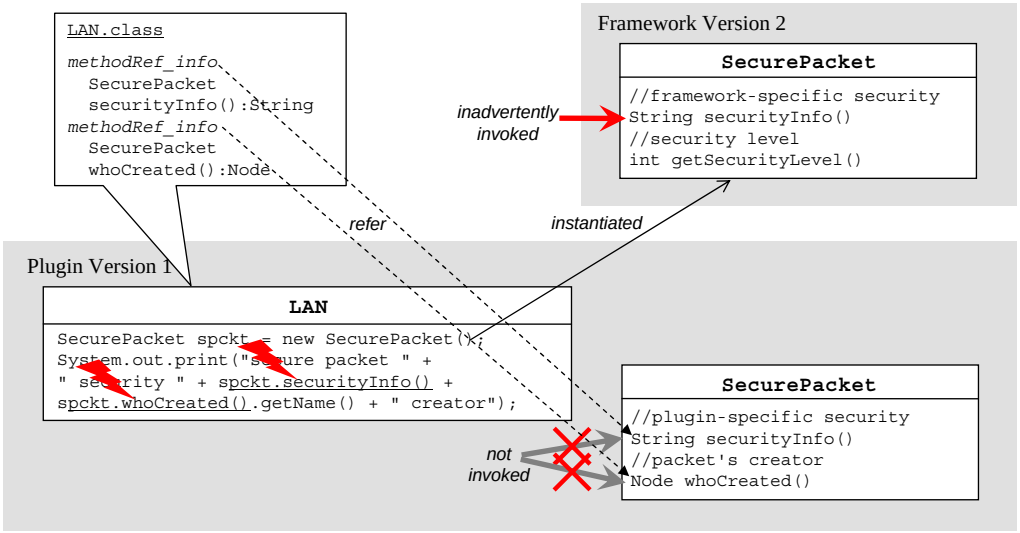
**Framework Type Capture.** This problem occurs when instead of a type expected by a framework caller dynamic linking locates a type defined in the plugin. Depending on the differences between the expected (framework) and the located (plugin) types, the framework caller may invoke methods, which are either not specified in the located type or have unexpected semantics.

For an example, let us assume the plugin of the first version defining a class *SecurePacket* (the bottom right part of Figure 2.11(a)) with two methods: (1) *securityInfo()* returning information of a security policy as defined in the plugin, and (2) *whoCreated()* returning the creator node of the packet. In the second framework release, framework developers introduce a new API class, accidentally called *SecurePacket*, with the same fully qualified name as the plugin's *SecurePacket* (the top right part of Figure 2.11(a)). The newly introduced framework class *SecurePacket* defines two methods: (1) *securityInfo()* returning information about the framework-specific security, and (2) *getSecurityLevel()* returning the packet's level of security. The two methods should be invoked in the framework's *Utils* on an instance of the framework's *SecurePacket* (the left part of Figure 2.11(a)). Thereafter, the framework is upgraded to its second, refactored version, while the plugin is not recompiled.

At application execution time, the plugin class *SecurePacket* is referenced (from some other plugin class), and is therefore loaded and registered with its class loader. Thereafter, the framework's *Utils* is to be loaded and, since it references the class *SecurePacket*, the JVM uses the fully qualified name of *SecurePacket* to check, whether the class is already loaded by *Utils*'s class loader. In case the framework and plugin types are loaded by the same class loader (e.g., by the system class loader), the JVM will consider *SecurePacket* as already loaded, and will instantiate the plugin's *SecurePacket* instead of loading and instantiating the framework's *SecurePacket* (as shown in Figure 2.11(a)). As a consequence, *Utils*'s call to *securityInfo()*




(a) Framework Type Capture. In case the plugin's *SecurePacket* is loaded first, its functionality takes over the one of the framework's *SecurePacket* expected in *Utils*' method calls.




(b) Plugin Type Capture. In case the framework's *SecurePacket* is loaded first, its functionality takes over the one of the plugin's *SecurePacket* expected in *LAN*'s method calls.

Figure 2.11: Type Capture. The framework's *SecurePacket* and the plugin's *SecurePacket* have exactly the same fully qualified names, and are loaded by the same class loader.

will inadvertently obtain the semantics of the method defined in the plugin (instead of the framework), while *Utils*'s call to *getSecurityLevel()* will not locate a method implementation. The former may lead to an application malfunction, and the latter will fail with a linkage error.

**Plugin Type Capture.** Similar to a plugin type taking over a framework type in case of a framework caller, a framework type may take over a plugin type in case of a plugin caller. In the example of Figure 2.11(b), in case the framework's *SecurePacket* is loaded before the plugin's *SecurePacket*, an instance of the framework's *SecurePacket* will be created in *LAN* instead of the expected plugin's *SecurePacket*. As a consequence, method invocations in the plugin's *LAN* may execute unintended framework methods (e.g., *securityInfo()*) or fail to locate a method implementation (e.g., *whoCreated()*).

**Effect.** In general, Type Capture may be caused by refactorings adding, renaming, and moving framework classes and interfaces (including type renaming as a consequence of the `RenamePackage` refactoring).[16] More specifically, Type Capture may occur, if for a framework and a plugin type (1) their fully qualified names are the same, and (2) the types are loaded by the same class loader.[17] At first glance, this situation seems improbable: framework developers should deploy the API types in dedicated packages, and application developers should create application-specific packages for plugin types. However, in practice application developers do not always carefully follow package naming conventions. Moreover, application developers may intentionally place plugin types in packages containing framework API types, to reuse certain API types accessible only within their packages (Java *package visibility*, applied by default, when no type visibility is specified). Finally, as a consequence of package renaming, framework and application package names can unintentionally coincide after upgrading the refactored framework.

### 2.2.4 Choosing Adaptation Means for Application-Breaking API Refactorings

Knowing the mechanics of application-breaking API refactorings, we can reason about the appropriate adaptation means to be applied. Moreover, considering existing approaches developed for component update, we can judge the applicability and availability of *intrusive* (modifying components directly) and *unintrusive* (using adapters) adaptation approaches.

#### Adapting Non-localizable Functionality

According to Becker et al. [BBG+04], any syntactic mismatch is adaptable. In case of Non-localizable Functionality, the types of old plugins must be reconciled with the changed API of the framework.

Intrusively, the obsolete information about API types contained in plugins must be updated to refer to the latest framework API. In case plugin sources are available and can be modified, they can be rewritten [BTF05, CN96, HD05] and then recompiled. If plugin binaries are accessible, they can be updated directly (e.g, [KH98]). In case plugins are unavailable, the framework API itself can be rewritten to reconcile the framework API and the plugins [DNMJ08].

Unintrusively, adapters can be created (e.g., [BCP06, IT03, YS97, ŞRS+07, ŞR07a, ŞRGA08]) that translate between the old plugin types and the refactored API types. Since, as stipulated by Becker et al., using adapters "theoretically every interface can be transformed into every other interface" [BBG+04], with regard to Non-localizable Functionality unintrusive adaptation can cover in fact all application-breaking API refactorings. We will discuss our adaptation patterns solving Non-localizable Functionality in Section 3.1.

#### Adapting Missing Functionality

When the functionality required does not exist, it needs to be provided by developers. In case an API method has been deleted, it can be recovered from an old framework version [DNMJ08]. Otherwise, developers need to specify the missing implementation manually. In some cases, a default implementation may suffice.

Intrusively, the missing implementation can be inserted either into the framework (in case of Missing Framework Functionality), or into plugins (in case of Missing Plugin Functionality). Unintrusively, it can be inserted into adapters [GHJV95, p. 142]. The latter is more restrictive: the implementation inserted may require access to private (framework or plugin) types and members, not visible from adapters. Since for each case of Missing Functionality the solution depends on the particular implementation that is missing, there is no general adaptation design for this problem group. However, in most of the cases, adapters following our adaptation patterns described in Section 3.1 can be extended with the corresponding implementation to compensate for Missing Functionality.

[16] Type Capture may also occur, when the framework type being refactored is not itself public, but is placed in an API package (in Java, by declaring the type as *package-private*). We treat this special case as an API refactoring, because it involves modifying the content of an API package.

[17] Types loaded by different class loaders are considered different by the JVM, even if the types' fully qualified names are the same.

### Adapting Unintended Functionality

Solving Unintended Functionality requires explicit control over the relations between framework and plugin objects to choose the "right" implementation at run time. Recalling our examples, for Unintended Framework Functionality: Inconsistent Method (Figure 2.9, page 31), it would mean that the framework's *Utils* should be able to invoke the plugin method *broadcast()* instead of the framework's *manycast()*, whenever an instance of *SecureNode* is asked to send a message. For Unintended Plugin Functionality: Method Capture (Figure 2.10, page 33), it would mean that the framework's *Utils* should be able to call the *getDescription()* method of *Node*, even if it gets an object of *SecureNode*.

Solving Unintended Functionality intrusively should be possible, at least, at the level of binary code. One solution would be to disable method dispatch for the methods otherwise accidentally overridden, so that the exact method initially referred to from the binaries is called.[18] Another solution (e.g., based on an aspect-oriented technology) would intercept messages at run time to invoke the intended methods. To the best of our knowledge, no such technology exists, and intrusive adaptation of Unintended Functionality remains a promising area of future work.

Unintrusively, since we have full control on the adapter creation, we implement proper method dispatch directly in adapters, solving Unintended Functionality. A proper implementation of method dispatch in adapters makes the unintended behavior invisible for the callers, which can therefore invoke the intended functionality. In case of Unintended Framework Functionality: Inconsistent Method, adapters localize the existing hook method implementation in plugins and call that implementation instead of the default hook method implementation. In case of Unintended Plugin Functionality: Method Capture, adapters dispatch to the otherwise accidentally overridden framework methods and avoid calling unintended plugin methods. We will discuss particularities of solving Unintended Functionality by Java adapters in Section 3.1.3.

### Adapting Type Capture

Solving Type Capture requires distinguishing the types of the refactored framework and existing plugins, regardless of the types' fully qualified names. To remain compatible with the host language (e.g., Java), a compensating solution should avoid such technical solutions as non-standard changes to the language execution environment (e.g., modifying the JVM loader).

Intrusively, Type Capture could be solved by prepending all plugin types with a unique prefix, and updating all existing type references correspondingly. To guarantee the type name uniqueness, the most practical solution would operate on plugin binaries by prefixing plugin type names with characters not allowed in the source code (e.g., #). Furthermore, to distinguish plugin types of different versions, the prefix should reflect the plugin versioning information. In addition, such solution would need to make the plugin metadata conform to the renamed binaries (in Java, by modifying JAR file manifests and META-INF/services entries) as well as to modify any script referring to the plugin code being rewritten. Effectively, such type renaming would deliver a plugin with its types named differently from the framework types, and from the types of plugins of other versions.

Unintrusively, we distinguish the plugin and framework types by using adapters and a dedicated strategy of type loading. Recalling our discussion from the Introduction (Figure 1.5 on page 7), adapters created between the refactored framework and an old plugin constitute an adaptation layer placed between the framework and the plugin. The adaptation layer reconstructs the old API expected by the plugin and translates between the refactored framework and the old plugin. Reconstructing the old API implies that for all renamed API types (potentially leading to Type Capture) their adapter types in the reconstructed API are named as before the refactoring, while for all newly introduced API types (potentially leading to Type Capture) no adapter types are present in the reconstructed API. As a consequence, there are exactly two name spaces, within which any type name is unique: (1) the name space of the reconstructed API and old plugin types, and (2) the name space of the refactored framework. What is still required is separating the two

[18] This solution would require disabling virtual (by default) method calls by making them non-virtual. While in such languages as .NET's CIL it is possible to specify, whether a method call is virtual (instruction *callvirt*) or non-virtual (instruction *call*), in Java source or binary code it is not possible.

name spaces at run time; in Java, it can be done by dedicating to each name space a separate class loader.[19] Since plugin types and framework types are loaded by different class loaders, no Type Capture is possible in our solution. We will show the insights of the adaptation layer in Section 3.2, describe precisely its creation in Chapter 4, and present technical details of separate class loading implemented in our tool in Section 5.1.2.

## 2.3 Summary

To foresee the general applicability of refactoring-based adaptation, we set up a critical case study of unconstrained API evolution. The results of our study—application-breaking API refactorings accounting for about 85% of application-breaking API changes—made us believe that refactoring-based adaptation is feasible in a large number of evolving frameworks. When developers tend to maintain the framework, most of their API changes affecting existing plugins can be modeled by refactorings, making refactoring-based adaptation possible. Even in the case of software replacement, where changes in our case study mostly went beyond refactorings, the role of refactorings needs further investigation. In contrast to the *big-bang* migration [RH06, p. 237] from the old to the new system (as in case of JHotDraw 7.0), when backward compatibility is a concern, framework may be replaced *gently* [RH06, p. 237] by small iterative changes, while simultaneously upgrading applications. In such scenarios of software migration, refactoring-based adaptation can be useful [ŞA08].

Admittedly, there is a number of frameworks, for which their maintenance may not repeat the pattern of refactorings discovered in our case study and the studies reported by other researchers [DJ05, DR08, SJM08]. For example, in comparison to a mature framework, a "young" framework in an early phase of development may evolve in a more drastic way. The APIs of certain frameworks (e.g., Web frameworks) typically expose many library types, changes of which (e.g., when upgrading the implementation language) are not encapsulated and cannot be modeled as refactorings. Since we did not know the details of the development process (e.g., how Eclipse [Ecl] or NetBeans [Net] were used to evolve the frameworks investigated in our case study), we could not estimate the impact of using an IDE's refactoring engine on the ratio of API refactorings. A future work should investigate the API evolution of a larger number of commercial and open-source frameworks varying in size, purpose, and the development policies.

However, what makes us optimistic is the fact that our results were obtained in a *critical* case study. Preserving backward compatibility will furthermore increase the probability of refactorings. Concerned with preserving existing applications, framework developers will avoid to the greatest extent possible application-breaking API changes, applied otherwise in unrestricted API maintenance. Even more important, maintaining backward compatibility dictates a certain "smooth" way of API evolution, when the transition between framework releases (e.g., in Eclipse) is supported by adapters that translate between the old and the modified APIs. In such cases, APIs will presumably evolve in small steps, mostly refactorings, enabling thus refactoring-based adaptation [DJ05].

We shed light on problems caused by API refactorings by explaining the mechanics of breaking changes. Except for certain cases of Missing Functionality, unintrusive (adapter-based) adaptation can compensate for the same application-breaking API changes as intrusive adaptation. However, in practice adapters are applicable also in cases, when plugins are unavailable for analysis and upgrade, or software licenses forbid plugin changes. In addition, developers are often reluctant to use a technology, which directly manipulates their code (as reported by Dig [Dig07, p. 131]). Therefore, we argue for adapters as general and practical adaptation means applicable in a large number of adaptation scenarios. Still, in case at some later point of framework maintenance the plugins become available and sufficient maintenance resources can be allocated for careful plugin analysis, update, and testing, one can switch to intrusive adaptation as an alternative to adapters.

[19] As another example, in .NET type name separation can be realized by registering the classes with the same names but different versions in the Global Assembly Cache.

*"So, the real work of any process of design lies in this task of making up the [pattern] language, from which you can later generate one particular design."*

–Christopher Alexander
*The Timeless Way of Building* [Ale79, p. 305]

# 3

# Designing Adapters for Binary Backward Compatibility in Refactored Framework APIs

The Adapter design pattern is used when "you want to use an existing class, and its interface does not match the one you need" [GHJV95, p. 140]. Discussing the pattern's power, Becker et al. argue: "The adapter pattern is very flexible as theoretically every interface can be transformed into every other interface" [BBG+04]. In particular, in case the Application Programming Interface (API) of a software framework evolved through refactorings, the pattern can be applied to make the code written using an old version of the API match the refactored API.

However, the original intent of the Adapter design pattern is to make "existing and unrelated classes [...] work in an application that expects classes with a different and incompatible interface" [GHJV95, p. 139]. In such cases of *component integration*, independently developed components being wired together are usually not aware of each other's APIs. By contrast, right from their creation plugins are developed against the framework's API reusing its types by object composition (black-box reuse), inheritance (white-box reuse), or the combination of both. Consequently, the implementations of frameworks and plugins are much more tightly coupled than those of components developed independently. In case of *component evolution* performed via refactorings, such coupling requires additional adaptation means to accommodate for API refactorings.

To provide binary compatibility of the refactored API with existing plugins, in this chapter we carefully define how adapters should be used in the context of refactoring-based API evolution. The chapter brings the following thesis contributions:

**C3: Adaptation design compensating for a refactored API type.** Since frameworks can be reused by a mixture of object composition and inheritance, we introduce four variants of the Adapter design pattern, named Black-Box Class and Interface Adapter patterns, and White-Box Class and Interface Adapter patterns, that are applicable in refactored framework APIs. To support binary compatibility, both patterns distinguish adapting refactored API classes and API interfaces. Black-box and white-box adapters are classes connecting types of the existing (old) plugins and the refactored framework API. While black-box adapters help existing plugins to locate the refactored API types of the framework, white-box adapters also support the refactored framework to call back functionality of existing plugins. This feature of the white-box adapters is especially important for adapting software frameworks that often heavily rely on the callback mechanism to realize inversion of control [JF88].

**C4: Adaptation design compensating for a refactored framework API as a whole.** Considering the adaptation of the framework's API as a whole, we introduce *exhaustive API adaptation*. Since for most API types we do not know how they are reused by plugins (by object composition or by inheritance), for all API types possibly changed by refactorings we apply *both* Black-Box and White-Box Adapter patterns and statically create adapters possibly needed to adapt these API types. At application execution time, depending on the run-time types of the objects requiring adaptation, the proper (black-box or white-box) adapters are instantiated. Moreover, to avoid existing plugins observing framework objects of refactored API types and, at the same time, the refactored framework observing plugin

objects of incompatible plugin types, we completely separate *the object domains* of the framework and existing plugins. All potentially dangerous objects coming from one site (the framework or plugins) are adapted (wrapped) before being sent to another site and unwrapped if sent back at some later point in time. The adaptation decisions (which adapters to instantiate and when) are implemented in adapters and are completely transparent for both sites.

**C5: Description of challenges, adopted solutions, and limitations in using adapters for refactored framework APIs.** We delimit precisely our approach by discussing encountered challenges, adopted solutions, and limitations. By caching and reusing adapter instances during exhaustive API adaptation, we preserve object identity in the presence of adapters and solve object schizophrenia associated with (the object version of) the Adapter design pattern. In case existing plugin code makes implicit assumptions about objects' physical structure or type (in Java, by using reflection), introducing adapters may cause that code to malfunction. To avoid the latter, in our adaptation design we anticipate, whenever possible, these dangerous assumptions to properly create adapters or, at least, warn framework developers about the possible negative impact of such assumptions. Concerned about unavoidable performance degradation caused by adapters, we introduce static and run-time optimizations to decrease the performance overhead. Finally, since using the Adapter design pattern implies also limitations due to the particularities of the pattern (e.g., no adaptation of public fields), we discuss the nature of refactorings we can or cannot adapt.

The structure of this chapter reflects its main contributions: Section 3.1 introduces the Black-Box and White-Box Class and Interface Adapter patterns, Section 3.2 discusses exhaustive API adaptation of refactored framework APIs, and Section 3.3 summarizes the problems, adopted solutions, and limitations in using adapters to compensate for API refactorings.

## 3.1 Adaptation Design Compensating for a Refactored API Type: Variants of the Adapter Design Pattern

In the line with other design patterns [GHJV95, p. 5], Gamma et al. describe the Adapter pattern (also known as Wrapper, Figure 3.1) in several pattern sections, such as the pattern name and intent, a motivating example of its use, general pattern applicability, pattern structure, its participants and their run-time collaboration [GHJV95, pp. 139–150]. The intent of the pattern is to "convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces" [GHJV95, p. 139]. The pattern's key participant, called Adapter, connects the Target interface expected by a caller (Client) to an Adaptee that provides the required functionality. When Client invokes a method of Target on an instance of Adapter, the latter calls Adaptee operations to serve the request on behalf of Client. The pattern can be implemented using either inheritance (class version) or object composition (object version), with the structure of the latter being shown in Figure 3.1.[1]

As mentioned in the introduction to this chapter, the original description of the Adapter design pattern focuses on its application for component integration. To make the pattern applicable in the context of component (in our case, framework) evolution, we identify furthermore two main particularities intrinsic to the evolution of framework APIs. These particularities must be considered to achieve our main goal—the binary backward-compatible framework upgrade—in the presence of black-box and white-box framework reuse.[2]

1. **Complex pattern collaboration.** Due to its primary focus on component integration, the collaboration of the Adapter pattern assumes black-box reuse: (1) Client calls Target, (2) Adapter calls Adaptee, and (3) Adapter returns the results from Adaptee to Client.

[1] In the following, we consider solely the object version of the pattern, since we are adapting Java frameworks, and the language does not support multiple inheritance (as required by the pattern's class version).

[2] Recalling our footnote from Introduction on page 2, we consider implementing an API interface a white-box reuse.

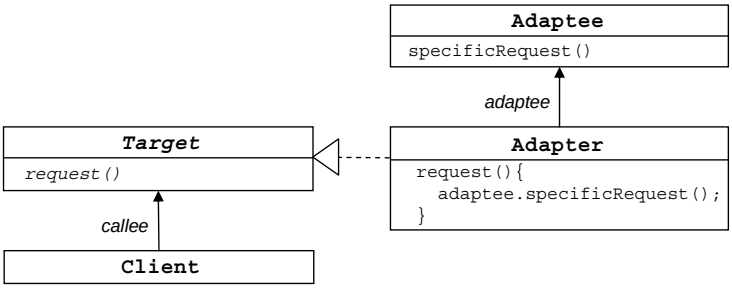



Figure 3.1: Adapter design pattern (object version), adopted from Gamma et al. [GHJV95, p. 141].

This general collaboration applies also to the adaptation in black-box framework reuse, since Client is always a plugin type calling some API methods. As a consequence, a black-box adapter must dispatch calls from the plugins to the framework: it receives a call request from Client (a plugin type), calls Adaptee (a framework type) and returns the answer to Client.

By contrast, due to class inheritance, in white-box framework reuse *both* plugins and the framework may act as Clients. While plugins may call inherited framework methods and supercall methods of their parent API classes, the framework may call back plugin methods. For example, by applying such design patterns as Factory Method [GHJV95, p. 107] and Template Method [GHJV95, p. 325] framework developers expect certain API methods to be defined (if an API method is abstract) or redefined (if an API method provides a default implementation) by plugin developers. When these API methods are called from the framework, the functionality implemented in plugins should be invoked, realizing thus inversion of control [JF88]. As a consequence, in such cases a white-box adapter must dispatch not only calls from the plugin to the framework, but also calls from the framework to the plugin. In both directions, the adapter must be able to access the functionality required by Clients.

2. **Reconstructing expected API types.** In the description of the Adapter design pattern it is assumed that all types that correspond to pattern participants, except for Adapter, exist; only the adapter class needs to be constructed. Again, this assumption holds when integrating independent components. However, when connecting existing plugins with the refactored framework, the old API types (expected by plugins) have been refactored and do not exist after the framework upgrade. Therefore, these old API types need to be constructed (more precisely, *reconstructed*) together with adapter classes. Moreover, in Java the binaries of existing plugins may contain the information about the physical representation of expected API types: whether an expected API type is specified by a separate API interface, or specified and implemented by an API class directly. To provide binary backward compatibility, while reconstructing an API type we must also satisfy such plugin expectations about its implementation. More specifically:

   - In case an expected API type to be reconstructed is used in a plugin by object composition (black-box framework reuse), for backward compatibility in general it would suffice to reconstruct the type itself (i.e., its name and interface, the latter consisting of a set of method signatures), regardless of whether the API type is physically specified by an API class or by a separate API interface. However, this would in addition require plugin recompilation; otherwise, the linker may possibly fail to locate the reconstructed type's methods. For example, if an existing plugin expects an API type to be specified by an API interface, while in the adapter Target (which corresponds to the expected API type) and Adapter are realized by a single adapter class, the linker of the Java Virtual Machine (JVM) will most probably fail to locate the adapter, unless the plugin is recompiled and its binaries are updated by the compiler.[3] Since we aim for *binary* backward compatibility, we have to consider also the physical representation of the API type being reconstructed.

[3]The actual run-time result depends on the particular JVM implementation, which varies among vendors.

**Terminology Note**

Strictly speaking, a **type** is a name denoting an interface—a set of signatures (operators' names, parameter names and types, and return types) specifying the method requests that can be sent to an object of that type. To distinguish the conceptual notion of a type's interface from the notion of Java interface as an implementation unit, by an **API type** we mean both a name and its associated interface accessible in the API. In Java terminology [GJSB05], an API type may be *specified* by a separate **API interface** (the methods of which are implemented in one or more classes), or *specified and implemented* by an **API class** directly.

- While in black-box framework reuse the physical representation of reconstructed API types matters for *binary* compatibility, in white-box framework reuse it is important for backward compatibility in general. For instance, if the expected type is an API interface, while the reconstructed type is specified directly by an adapter class, not only executing plugin binaries but also recompiling plugins will fail. Therefore, the reconstructed API type must mimic all public information previously available in the expected (old) API type, including the type's physical representation.

Considering these two main particularities of our adaptation context, we introduce four variants of the Adapter design pattern—the Black-Box Class and Interface Adapter patterns, and the White-Box Class and Interface Adapter patterns—that compensate for API type refactorings. The patterns reflect the details of the physical representation of reconstructed API types by defining class and interface adapters to account for the refactoring of API classes and API interfaces, respectively. In the patterns' collaborations, black-box class and interface adapters dispatch calls from old plugins to the refactored framework, white-box interface adapters dispatch calls from the refactored framework to old plugins, and white-box class adapters support bidirectional message exchange between the refactored framework and old plugins.

Consistent with the pattern descriptions of Gamma et al. [GHJV95, p. 5], we describe our patterns using a number of pattern sections. However, since our patterns are *variants* of the Adapter design pattern, we omit sections that do not add to the discussion on the patterns' difference to the original description of the Adapter pattern. The main intent of the presented sections (most importantly, Structure and Collaboration) is to denote the patterns' specifics with respect to the corresponding section of the Adapter pattern. All omitted sections should be considered as inherited, with language-specific adoption to Java, from the original pattern description [GHJV95, pp. 139–150]. For example, according to the Adapter pattern, the adapter introduced may also be responsible for the functionality not accessible in the adapted class [GHJV95, pp. 142]. This is inherited in our patterns: if the implementation of an API method is missing, the corresponding method implementation can be provided by an adapter class directly, compensating for Missing Functionality (discussed in Section 2.2.3). Therefore, compensating for Missing Functionality is not discussed furthermore when presenting the adaptation patterns.

Besides Missing Functionality, in our pattern descriptions we do not discuss solving Type Capture that may occur when the name of a newly introduced API type or of a renamed API type accidentally coincides with the name of an existing plugin type (as described in Section 2.2.3). A solution to this problem, which we

**Terminology Note**

A **class adapter** is an application of the Adapter design pattern for a refactored API class. It should not be confused with the **class version** of the Adapter design pattern, that is, the version based on inheritance. In their implementations, all our adapters (including class adapters) are using object composition as defined in the **object version** of the Adapter design pattern.

already discussed in Section 2.2.4 on page 36, goes beyond compensating for one particular refactored API type and implies separating the name spaces of the framework and plugins. However, all our adaptation patterns contribute to the general solution of Type Capture: if an API type is renamed, its corresponding black-box or white-box adapter will reconstruct the old API type as expected by an old plugin. Thereby, together with a dedicated type loading, the reconstructed API types help separating the name spaces of the framework and the plugin.

In our patterns, we need to refer to two versions of the API class (or interface) being adapted: the version before the refactoring and the version after the refactoring. We will reference the former as *unrefactored API class (interface)*, and the latter as *refactored API class (interface)*. In the examples of pattern applications, we use excerpts from RefactoringLAN—a set of examples of framework reuse and application-breaking API refactorings (introduced in Section 2.2),[4] and show how our adaptation patterns solve Non-localizable Functionality and Unintended Functionality (introduced in Section 2.2.3). Although in our examples we use source code (while omitting unnecessary implementation details), in fact, in the following we describe binary adapters. Some of the presented adaptation decisions had not been possible at the source level, because certain adapters would not have compiled. However, the decisions were realized by generating adapter binaries directly.

In the rest of this section, we will present the Black-Box Class and Interface Adapter patterns (Section 3.1.1 and Section 3.1.2, respectively), the White-Box Class and Interface Adapter patterns (Section 3.1.3 and Section 3.1.4, respectively), and the rationale of the patterns' design and realization in Java in Section 3.1.5.

### 3.1.1 Black-Box Class Adapter Pattern

#### Motivation

In case a framework API class being reused by object composition in a plugin is changed, the plugin might be unable to locate the previously available framework functionality.

In the first framework release, for creating network packets an API class *PacketFactory* is offering a method *createPacket()* with two method parameters (a node of type *Node* and a node identifier of type *int*), and with the return value of type *IPacket*. To obtain a new packet, a plugin class *LAN* may call the method passing a node and the node's identifier as the required method arguments. (This example was introduced by Figure 2.3 on page 25.)

In the second framework release, framework developers realize that the second parameter of *createPacket()* is redundant, since the node identifier (as required in the implementation of *createPacket()*) can be obtained from the instance of node passed as the first method argument. Therefore, framework developers remove the redundant second method parameter.[5] By refactoring the class' method, they also effectively refactor the API type specified by the class. As a consequence, *LAN* call sites of the method cannot locate it after the framework is upgraded (Non-localizable Framework Functionality, shown for our example by Figure 2.6 on page 28). One solution would be to update all call sites in the plugin to invoke properly the refactored method. However, this is not possible in case the plugin is unavailable for change, or is not subject to modifications.

Instead, we could define an adapter class *PacketFactory* offering a method *createPacket()* with the two formal parameters *Node* and *int*, exactly as before the refactoring. An object of the framework class *PacketFactory* is composed within an object of the adapter class *PacketFactory* (in its adaptee field), while the adapter class *PacketFactory* is implemented in terms of the (API type of the) *PacketFactory* as before the refactoring. The adapter method *createPacket()* is invoked by existing calls from the old plugin and, in turn, forwards the calls to the refactored framework method *createPacket()* (Figure 3.2).

[4] RefactoringLAN is summarized in Appendix A

[5] Fowler et al. motivate this refactoring as follows [FBB+99, p. 224]: "A parameter indicates information that is needed; different values make a difference. Your caller has to worry about what values to pass. By not removing the parameter you are making further work for everyone who uses the method. That's not a good trade-off, especially because removing parameters is an easy refactoring."

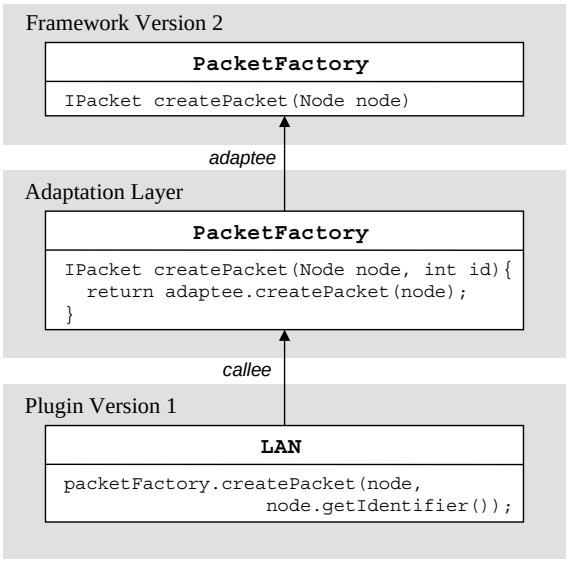



Figure 3.2: Application of the Black-Box Class Adapter pattern. The adapter accepts calls to *PacketFactory*'s old method *createPacket()* declared with two method parameters and forwards them to the refactored *createPacket()* declared with a single method parameter.

The problem caused by the aforementioned API method refactoring is solved as following: In case the plugin is calling *createPacket()* on the object of adapter *PacketFactory*, the latter will use the first method argument to call the method *createPacket()* on the framework object stored in the adaptee field, and then return the obtained value to the plugin.

Comparing to the original description of the Adapter pattern, this solution explicitly permits the adapter to be named exactly as the framework class. This is required by and is inherent to our adaptation context: except for type rename refactorings, the names of the expected (old) and provided (refactored) API types are the same (as *PacketFactory* in our example). However, conceptually the names are coming from different name spaces and can be effectively separated to avoid name clashes. Shortly, in Java these classes are loaded by different class loaders.

### Applicability

Use the Black-Box Class Adapter pattern, if

1. a framework API class used by object composition (black-box framework reuse) is refactored, and
2. existing plugins cannot be updated intrusively.

### Structure

Figure 3.3 shows the structure of the Black-Box Class Adapter pattern, annotated (by stereotypes) with the names of the participants of the Adapter design pattern.

### Participants

- **Refactored API Class** (Framework's *PacketFactory*)
  - defines the refactored API class.

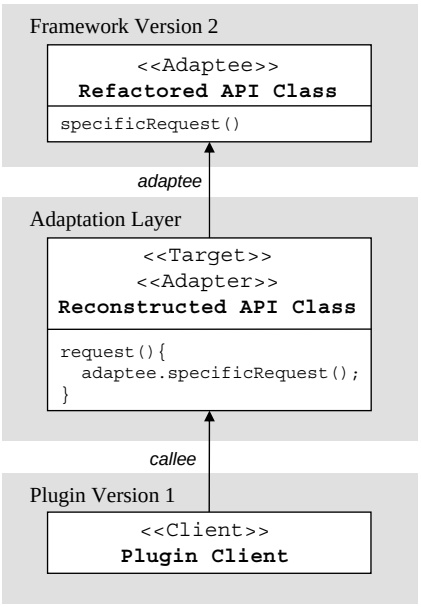



Figure 3.3: Black-Box Class Adapter pattern.

- **Reconstructed API Class** (Adaptation Layer's *PacketFactory*)
  - reconstructs the API type of the unrefactored API class.
  - adapts the API type of Refactored API Class to the API type of Reconstructed API Class.
- **Plugin Client** (Plugin's *LAN*)
  - collaborates with objects conforming to the API type of Reconstructed API Class.

#### Collaboration

Plugin Client calls operations on an instance of Reconstructed API Class. In turn, the instance forwards the calls to operations of Refactored API Class that carries out the requests.

### 3.1.2 Black-Box Interface Adapter Pattern

#### Motivation

Similar to an API class, a framework API interface reused by object composition in a plugin may change. As a consequence, the plugin might be unable to locate the previously available framework functionality, which was accessible via the API interface.

In the first framework release, an API interface *IPacket* contains a method *whoCreated()* with the return value of type *Node*. The interface is implemented by the framework's class *Packet*. The implementation of the method *whoCreated()* returns the node that created the packet in question. The plugin class *LAN* can obtain a network packet by invoking *PacketFactory*'s *createPacket()* (as described in the example of the previous Section 3.1.1). Thereafter, *LAN* may invoke on the packet the method *whoCreated()* to obtain the packet's creator. (This example was introduced by Figure 2.3 on page 25.)

In the second framework release, to follow adopted naming conventions, framework developers rename the interface method from *whoCreated()* to *getCreator()* and update the implementing framework class *Packet* correspondingly. As a consequence of the refactoring, *LAN*'s call sites of the method cannot locate it after

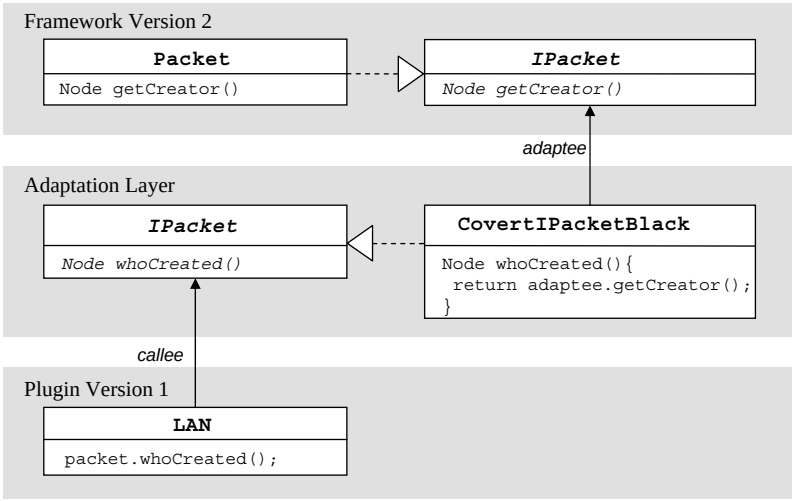



Figure 3.4: Application of the Black-Box Interface Adapter pattern. The adapter wraps the refactored *IPacket* of the framework, accepts plugin's calls to the method *whoCreated()* and forwards them to the framework's *getCreator()*.

the framework is upgraded (Non-localizable Framework Functionality, shown for our example by Figure 2.6 on page 28). Similar to the aforementioned API class refactoring, one solution would be to update all call sites in the plugin to invoke the renamed method. However, this is not possible in case the plugin cannot be modified.

Instead, we could introduce two adapter types: a (reconstructed) adapter interface *IPacket* declaring the method *whoCreated()* with exactly the same signature as before the refactoring, and a class named, say, *CovertIPacketBlack*, implementing the adapter interface *IPacket* (Figure 3.4). An object of the refactored framework type *IPacket* is composed within an object of the adapter class *CovertIPacketBlack*. In its implementation, *CovertIPacketBlack*'s *whoCreated()* method invokes the renamed framework method *getCreator()*. When the framework instantiates the class *Packet* implementing the refactored interface *IPacket* and returns the instance to the plugin, the adapter *CovertIPacketBlack* "wraps up" the refactored framework implementation and "hides" it behind the reconstructed API interface *IPacket*. Such adapter classes as *CovertIPacketBlack* are visible neither for the refactored framework nor for the existing plugin, because they are introduced by and hidden in the adaptation layer. Therefore, we call such hidden adapter classes "covert".[6]

The problem caused by the aforementioned API interface refactoring is solved as following: In case the plugin is calling *whoCreated()* (specified by the adapter interface *IPacket*) on the object of *CovertIPacketBlack*, the object will call the method *getCreator()* on the object stored in its adaptee field, and then return the obtained value to the plugin.

Similar to the Black-Box Class Adapter pattern, and for the same reasons, this solution explicitly permits the adapter interface to be named exactly as the framework API interface.

### Applicability

Use the Black-Box Interface Adapter pattern, if

1. a framework API interface used by object composition (black-box framework reuse) is refactored, and
2. existing plugins cannot be updated intrusively.

[6] In practice, the names of covert adapter classes are obfuscated to avoid possible name clashes with framework and plugin types. Since we generate binary adapters, we prepend the names of the covert adapter classes in the adapter binaries with characters not allowed in the source code (e.g., #).

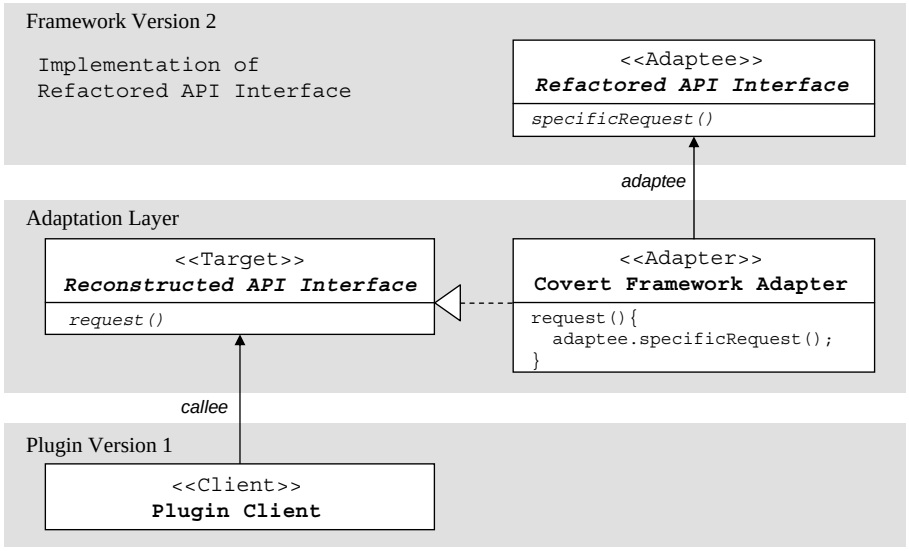



Figure 3.5: Black-Box Interface Adapter pattern.

**Structure**

Figure 3.5 shows the structure of the Black-Box Interface Adapter pattern, annotated (by stereotypes) with the names of the participants of the Adapter design pattern.

**Participants**

- **Refactored API Interface** (Framework's *IPacket*)
  - defines the refactored API interface.
- **Reconstructed API Interface** (Adaptation Layer's *IPacket*)
  - reconstructs the unrefactored API interface.
- **Covert Framework Adapter** (Adaptation Layer's *CovertIPacketBlack*)
  - adapts the API type of Refactored API Interface to the API type of Reconstructed API Interface.
- **Plugin Client** (Plugin's *LAN*)
  - collaborates with objects conforming to the API type of Reconstructed API Interface.

**Collaboration**

Plugin Client calls operations on an instance of Covert Framework Adapter. In turn, the instance forwards the calls to operations of Refactored API Interface that carries out the requests.

### 3.1.3 White-Box Class Adapter Pattern

A plugin may reuse the framework by subclassing its API classes. In this case, inherited framework methods may be called from the plugin, and plugin methods implementing (defining) or overriding (redefining) API methods may be called back from the framework. After an API refactoring, previously inherited framework functionality may not be accessible in the plugin, and previously called back plugin functionality may not

be accessible in the framework. Furthermore, other methods may be inadvertently invoked in the framework and plugin instead of intended methods.

In the first framework release, a framework API class *Node* offers three methods: (1) a concrete method *getIdentifier()* for obtaining a unique node identifier, (2) an abstract method *getNodeLocation()* for localizing a node in a network, and (3) a concrete method *broadcast()* for message sending. The framework class is subclassed in a plugin by *SecureNode* that implements *getNodeLocation()* correspondingly and overrides *broadcast()* to encrypt its messages before sending. In addition, *SecureNode* defines the *getDescription()* method returning the details of the encryption strategy used. (This example is a combination of two examples introduced by Figure 2.4 on page 25 and Figure 2.5(a) on page 26, from which we omit the *Node*'s method *getInfo()* for simplicity.)

The method *getIdentifier()* inherited in *SecureNode* is invoked by a usual method call from the plugin's class *LAN* and by a supercall from *SecureNode*. The other two methods declared in *Node* are invoked from the framework's *Utils*, executing the plugin implementation of *getNodeLocation()* and *broadcast()*.

In the second framework release, *Node* is refactored. To vary on communication strategies besides the default broadcasting, the method *broadcast()* is renamed to *manycast()* and now accepts as an additional method parameter an integer indicating a communication strategy. In addition, to respect adopted naming conventions, the methods *getIdentifier()* and *getNodeLocation()* are renamed to *getID()* and *getLocation()*, respectively. Finally, a new method *getDescription()* is introduced to *Node*, with a default implementation returning the node description.

Let us identify the problems caused by these refactorings, while assuming that the plugin is not recompiled (plugin recompilation would raise a number of compile-time errors) and that a plugin object as an instance of *SecureNode* is created after upgrading to the second framework version.

- In case the plugin is calling the previously available method *getIdentifier()* on this object, either via an ordinary call or via a supercall, the method implementation will not be found resulting in a run-time exception (Non-localizable Framework Functionality, Figure 2.7 on page 29).
- In case the object is passed to the framework, which asks for the node location by attempting to invoke the *getLocation()* hook method on the object, no corresponding implementation in *SecureNode* is found (Non-localizable Plugin Functionality: Unimplemented Method, Figure 2.8(a) on page 30).
- In case the framework asks the object of *SecureNode* to broadcast a message by invoking *manycast()* with the second parameter for broadcasting, the more specific method of *SecureNode* for sending encrypted messages will not be found. Instead, the Java method lookup will find and invoke the *manycast()* method of *Node* that will result in inadvertently sending a non-encrypted message (Unintended Framework Functionality: Inconsistent Method, Figure 2.9 on page 31).
- In case the framework invokes *getDescription()* on the object of *SecureNode*, the encryption description (as implemented in *SecureNode*) and not the node description (as implemented in the refactored *Node*) is obtained, which is not what is expected at the framework site (Unintended Plugin Functionality: Method Capture, Figure 2.10 on page 33).

One solution would be to manually adapt the plugin implementation of *SecureNode* by renaming its *getDescription()* to, say, *getEncryptionDescription()*, *getNodeLocation()* to *getLocation()*, and *broadcast()* to *manycast()*. For the latter method, developers would also need to extend its parameter list with an integer parameter for specifying a communication strategy. All existing method calls in the plugin to the manually adapted plugin methods would need to be updated to call the modified methods correspondingly (e.g., to specify the default value of broadcasting when calling *manycast()*). Moreover, all plugin calls to the modified framework methods would have to be updated as well (e.g., calling *getID()* instead of *getIdentifier()*). After applying these modifications, the plugin would need to be recompiled. However, such intrusive adaptation requires plugin availability and recompilation, and is cumbersome and error-prone.

Instead, we could define two adapter classes *Node* and *CovertNode* (both placed in Figure 3.6 in the Adaptation Layer) to translate between the framework's *Node* and the plugin's *SecureNode* (Figure 3.6).

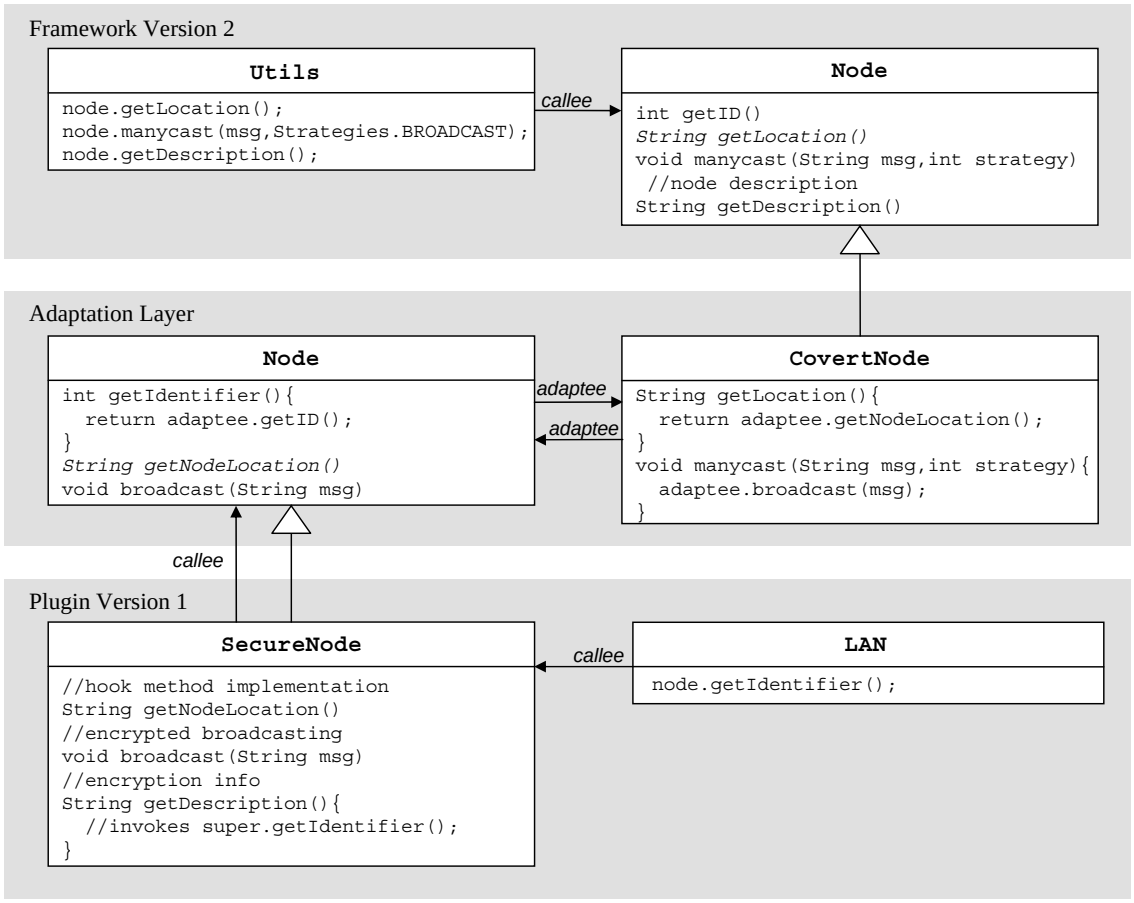



Figure 3.6: Application of the White-Box Class Adapter pattern. The adapter reconstructs the API class as expected in the plugin and dispatches method calls between an old plugin and the refactored framework API.

Since *SecureNode* expects the unrefactored (old) *Node* as its superclass, the adapter class *Node* reconstructs the API type of the unrefactored *Node*. At the same time, the adapter class *CovertNode* subclasses the framework's *Node*. Both adapter classes use each other by object composition in order to access from the plugin the inherited framework functionality (the adapter class *Node* via *CovertNode*) and from the framework the overriding plugin functionality (*CovertNode* via the adapter class *Node*, using an instance of its subclass *SecureNode*). The information, whether a plugin subclass (e.g., *SecureNode*) overrides an API method to be called back from the framework is inferred by *CovertNode* at run time by investigating the subclass' type information. Using this information, *CovertNode* performs multiple method dispatch by invoking the overriding plugin functionality (if found) or the default framework functionality (otherwise). The dispatch as performed in *CovertNode* is multiple, because its algorithm considers besides the type of the message receiver (in our example, an instance of *CovertNode*) also the type of the adaptee (in our example, an instance of *SecureNode*) as an additional, implicit method argument. We discuss this multiple method dispatch in detail in the pattern's Collaboration section.

An important requirement to perform proper adaptation is *object wrapping*: any framework object sent to the plugin and any plugin object sent to the framework are encased by a *wrapper*—an instance of the corresponding adapter.[7] In our example, whenever an object of *SecureNode* is sent to the framework, the object is encased by a wrapper—an instance of *CovertNode*. As a consequence, all framework calls invoked before the refactoring on the plugin object will be invoked after adaptation on the plugin object's wrapper.

[7] We will discuss object wrapping in detail in Section 3.2.

The problems caused by the refactorings of *Node* are solved as follows:

- In case a plugin is calling the method *getIdentifier()* on the object of *SecureNode* (either from *LAN* or via a supercall from *SecureNode*), the call will reach the implementation of the adapter class *Node* that will invoke the method *getID()* of the adapter class *CovertNode* inherited from the framework's *Node*.

- In case the plugin object of *SecureNode* is passed to the framework, which invokes the *getLocation()* method, since the object is wrapped, the method will be invoked on the object's wrapper (an instance of *CovertNode*), which in turn will call the plugin object's *getNodeLocation()*.

- Similarly, in case the framework invokes *manycast()*, a method of the wrapper (an instance of *CovertNode*) is invoked, which in turn will invoke the *broadcast()* method of its wrapped (i.e., plugin) object.[8]

- In case the framework invokes *getDescription()* (again, on the plugin object's wrapper), multiple method dispatch implemented in *CovertNode* will find no intentionally overriding method in *SecureNode* and, hence, will properly invoke the framework's *getDescription()* method by a supercall from *CovertNode* to the refactored framework *Node*.

This solution adapts *both* the framework to the plugin *and* the plugin to the framework. Addressing particularities of white-box framework reuse, the implementation of a white-box class adapter is in a sense a hybrid of the two versions of the Adapter pattern (i.e., based on inheritance and on object composition). Its key feature is the ability to support bidirectional communication between the framework and plugins by properly dispatching calls on inherited methods and supercalls (coming from the plugin), and callbacks (coming from the framework).

**Applicability**

Use the White-Box Class Adapter pattern, if

1. a framework API class subclassed in a plugin (white-box framework reuse) is refactored, and

2. existing plugins cannot be updated intrusively.

**Structure**

Since both the framework and the plugin may act as Clients, the White-Box Class Adapter pattern is in fact composed of two Adapter patterns—one for each calling site (i.e., the framework or the plugin). For clarity, in the pattern's definition we need to distinguish its participants depending on the calling site. For that, in Figure 3.7 we use prefixed names of Client, Target, Adaptee, and Adapter to annotate the White-Box Class Adapter pattern. For the case of a framework caller, we use FrClient, FrTarget, PlAdaptee, and Pl2FrAdapter, respectively. For the case of a plugin caller, we use PlClient, PlTarget, FrAdaptee, and Fr2PlAdapter, respectively. Moreover, to reflect the calling site, the adapter method names are prefixed with "fr_" and "pl_", respectively. However, the notation change is just a syntactic necessity; the semantics of the participants remain as in the original pattern description.

[8]This solution will suffice in case the framework asks for broadcasting, which is, in fact, exactly what should be expected from the old plugin, because it was developed with a framework version that did not support other communication strategies. Still, new calls of the refactored framework may also specify another communication strategy, such as multicast, expecting semantics different from broadcasting. In this case, *protocol adaptation* [YS97] going beyond refactoring-based adaptation is required; in Section 5.2, we show that protocol adaptation can be realized in combination with our adapters.

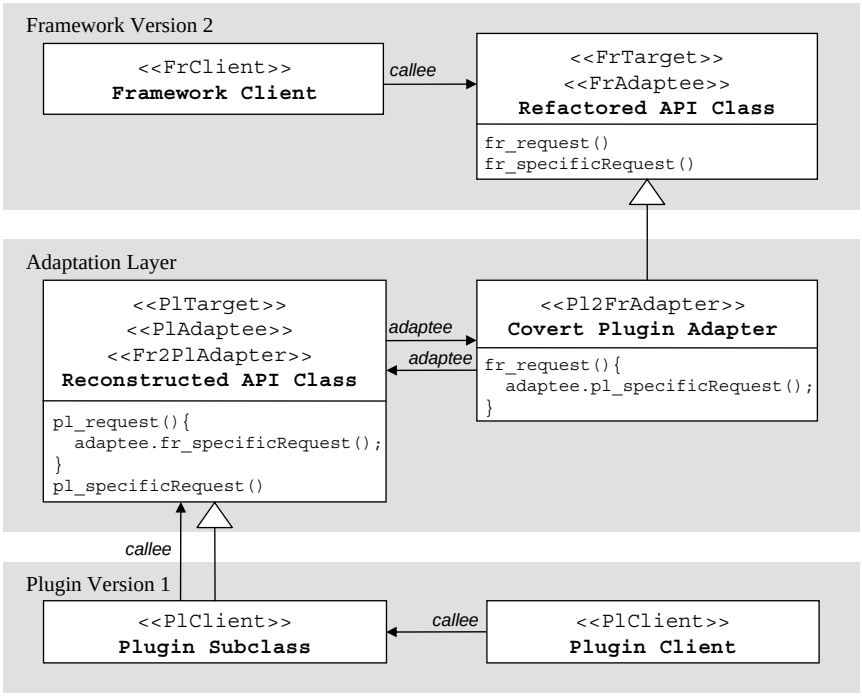



Figure 3.7: White-Box Class Adapter pattern.

**Participants**

- **Refactored API Class** (Framework's *Node*)
  - defines the refactored API class.
- **Reconstructed API Class** (Adaptation Layer's *Node*)
  - reconstructs the API type of the unrefactored API class.
  - adapts the API type of Refactored API Class to the API type of Reconstructed API Class.
- **Covert Plugin Adapter** (Adaptation Layer's *CovertNode*)
  - adapts the API type of Reconstructed API Class to the API type of Refactored API Class.
  - implements multiple method dispatch to call back methods of Reconstructed API Class overridden in Plugin Subclass.
- **Plugin Subclass** (Plugin's *SecureNode*)
  - defines a plugin subclass of the unrefactored API class.
- **Plugin Client** (Plugin's *LAN*)
  - collaborates with objects conforming to the API type of Reconstructed API Class.
- **Framework Client** (Framework's *Utils*)
  - collaborates with objects conforming to the API type of Refactored API Class.

Reconstructed API Class

```
public abstract class Node {
  //instance of Covert Plugin Adapter
  CovertNode adaptee;

  //forwarding to the framework
  int getIdentifier() {
    return adaptee.upcall_getID();
  }

  abstract String getNodeLocation();

  //forwarding to the framework
  void broadcast(String msg) {
    adaptee.upcall_manycast(msg,BROADCAST);
  }
}
```

Covert Plugin Adapter

```
public class CovertNode extends refactoredAPI.Node {
  //plugin instance
  Node adaptee;

  //multiple method dispatch
  int getID() {
    if (adaptee.getClass().declares("int getIdentifier()"))
      return adaptee.getIdentifier();
    else
      return super.getID();
  }

  //upcalling method
  int upcall_getID() {
    return super.getID();
  }

  //forwarding to the plugin without multiple method dispatch
  String getLocation() {
    return adaptee.getNodeLocation();
  }

  //multiple method dispatch
  void manycast(String msg,int strategy) {
    if (adaptee.getClass().declares("void broadcast(String msg)"))
      adaptee.broadcast(msg);
    else
      super.manycast(msg,strategy);
  }

  //upcalling method
  void upcall_manycast(String msg,int strategy) {
    super.manycast(msg,strategy);
  }
}
```

Figure 3.8: Adapter methods in the application of the White-Box Class Adapter pattern. In their implementations, the methods use Java type introspection and reflective method invocations (details omitted). For readability, the name of the refactored API class is prefixed with *refactoredAPI.*; in practice, the name need not be changed (in Java, the adapter *Node* and the framework *Node* are loaded by different class loaders).

### Collaboration

Plugin clients (i.e., Plugin Client and Plugin Subclass) call operations on an instance of Plugin Subclass. In case a call reaches Reconstructed API Class, the latter forwards the call to an operation of Covert Plugin Adapter. In turn, Covert Plugin Adapter uses an operation of Refactored API Class to carry out the request.

Framework Client calls operations on an instance of Covert Plugin Adapter. In turn, the instance forwards the call to an overriding operation (if any) in Plugin Subclass, or supercalls an operation of Refactored API Class (otherwise).

Let us take a closer look at the collaboration of the White-Box Adapter pattern by reconsidering, along with the pattern participants of Figure 3.7, also concrete classes from the pattern application example of Figure 3.6. The Java-like pseudocode of Figure 3.8 shows the adapter methods of Reconstructed API Class *Node* and Covert Plugin Adapter *CovertNode*. The collaboration of the two adapter classes provides the adaptation of the API methods of Refactored API Class to the API methods of Reconstructed API Class, and vice versa.

- **Adapting Refactored API Class to Reconstructed API Class.** This adaptation is performed in the methods of Reconstructed API Class (the adapter class *Node* in Figure 3.6) in case the plugin is acting as Client.

More specifically, the methods of Reconstructed API Class may be called exactly in two cases. First, Plugin Client (the plugin's *LAN*) invokes a method of Plugin Subclass (the plugin's *SecureNode*), the method is present in Reconstructed API Class and is not overridden in Plugin Subclass (e.g., *getIdentifier()* in Figure 3.6). Second, Plugin Subclass may supercall a method of Reconstructed API Class (the adapter class *Node*). In both cases, the call reaches Reconstructed API Class, which invokes the corresponding method of Covert Plugin Adapter (the adapter class *CovertNode*). For example, in the left listing of Figure 3.8 the invocation of *getIdentifier()* will reach the adapter class *Node* that will in turn invoke *CovertNode*'s method *upcall_getID()* (lines 6–8), the method's name to be explained in the next paragraph. As a consequence, the adapter protects from Non-localizable Framework Functionality.

Any method of Covert Plugin Adapter invoked from Reconstructed API Class (e.g., *upcall_getID()* shown by lines 14–16 in the right listing of Figure 3.8) is *upcalling*. An upcalling method has exactly the same signature as an API method in Refactored API Class, but prepended with a unique prefix "upcall_". Each upcalling method merely invokes its corresponding API method in Refactored API Class (as *upcall_getID()* invokes *super.getID()*). By using upcalling methods, we tell Covert Plugin Adapter to invoke the inherited framework functionality, instead of performing multiple method dispatch. We will come back to this issue at the end of the Collaboration section, after describing multiple method dispatch as implemented in Covert Plugin Adapter.

- **Adapting Reconstructed API Class to Refactored API Class.** This adaptation is performed in the methods of Covert Plugin Adapter (the adapter class *CovertNode* in Figure 3.6) in case the framework is acting as Client.

  More specifically, this adaptation is performed when Framework Client (the framework's *Utils* in Figure 3.6) attempts to invoke an API method, which is defined (for abstract API methods, such as *getLocation()* in Figure 3.6) or redefined (i.e., overridden, such as *manycast()* in Figure 3.6) in Plugin Subclass. Covert Plugin Adapter has to locate and invoke the corresponding defining or redefining methods (*getNodeLocation()* and *broadcast()*, respectively) in Plugin Subclass.

  As we mentioned in the Motivation section of this pattern and will further discuss in Section 3.2, an object of a plugin type sent to the framework is always encased by a wrapper—an instance of an adapter. In our example, an object of Plugin Subclass (*SecureNode* in Figure 3.6) sent to the framework is wrapped by an instance of Covert Plugin Adapter (*CovertNode*). Therefore, a method call of Framework Client (*Utils*) on the wrapper will invoke a method of Covert Plugin Adapter. There are exactly three cases handled in Covert Plugin Adapter, depending on the properties of the API method being invoked on behalf of Framework Client.

  1. The API method is newly introduced to the API (the method did not exist in the framework version, with which the plugin was developed). Such methods are simply inherited by Covert Plugin Adapter from Refactored API Class and are invoked to carry out the request of Framework Client. For example, in the right listing of Figure 3.8 *CovertAdapter* does not redefine *getDescription()* method, newly introduced to the refactored framework *Node*. Thereby, Covert Plugin Adapter prohibits accidental overriding of newly introduced API methods by existing plugin methods, such as the accidental overriding of the framework's *getDescription()* by the plugin's *getDescription()*. As a consequence, the adapter protects from Unintended Plugin Functionality: Method Capture.
  2. The API method is abstract (the method has no framework implementation). In this case, an implementation of the method in Plugin Subclass can be assumed, because otherwise the plugin would not have compiled with the old version of the API. For such methods, Covert Plugin Adapter defines adapter methods that invoke the corresponding methods of Reconstructed API Class on the instance contained in the adaptee field of Covert Plugin Adapter (e.g., containing an object of *SecureNode* in Figure 3.6). In lines 19–21 of the right listing of Figure 3.8, *CovertNode*'s method *getLocation()* will call *Node*'s *getNodeLocation()* on its adaptee, effectively invoking the method as implemented in *SecureNode*. As a consequence, the adapter protects from Non-localizable Plugin Functionality: Unimplemented Method.

3. The API method is concrete (the method is implemented in Refactored API Class or its superclasses). In this case, the plugin might override the method, but does not have to. Therefore, Covert Plugin Adapter performs multiple method dispatch to decide, whether to execute a plugin implementation or the default framework implementation. When dispatching, it uses the metadata about Plugin Subclass (in Java, using introspection) to check for the presence of an intentionally overriding plugin method, which corresponds to the API method being invoked by Framework Client.[9] In case overriding is detected, Covert Plugin Adapter calls the method of Reconstructed API Class on the plugin instance stored in the adaptee field of Covert Plugin Adapter (in our example case, an instance of *SecureNode*). Otherwise, the method of Refactored API Class is supercalled from Covert Plugin Adapter. In the listing of Figure 3.8, the two such adapter methods of Covert Plugin Adapter (i.e., *CovertNode*) performing multiple method dispatch are *getID()* (lines 6–11) and *manycast()* (lines 24–29). As a consequence of multiple method dispatch, the adapter protects from Unintended Framework Functionality: Inconsistent Method.

Since Java does not provide run-time information about method callers, we use upcalling methods in Covert Plugin Adapter to distinguish between Plugin Client and Framework Client. In the former case, Covert Plugin Adapter should invoke the framework functionality inherited in plugins; in the latter case, Covert Plugin Adapter needs to perform multiple method dispatch to detect, whether overriding plugin functionality exists. Since upcalling methods of Covert Plugin Adapter are invoked exclusively from Reconstructed API Class, their invocation implies a call coming from the plugin, which should be forwarded to the framework without multiple method dispatch.[10]

Besides avoiding unnecessary dispatch for the method calls coming from the plugin, by using upcalling methods we also avoid Plugin Subclass inadvertently invoking its own methods, when inherited API methods should be invoked instead. For our running example from Figure 3.6, let us assume that from one of its methods *SecureNode* (i.e., Plugin Subclass) invokes *super.broadcast()*, performing a supercall to the method defined in *Node* (i.e., Refactored API Class). In this case, the framework method in *Node* should be invoked, and not the overriding plugin method in *SecureNode*. However, performing multiple method dispatch in *CovertNode* (i.e., Covert Plugin Adapter) would detect and inadvertently invoke the overriding method in *SecureNode*. This would lead either to executing the unintended functionality of the overridden method (e.g., in case *super.broadcast()* in invoked from the *SecureNode*'s *getDescription()* method) or to entering a possibly infinite call loop (e.g., in case *super.broadcast()* is invoked from the *SecureNode*'s *broadcast()* method). Via the distinct upcalling method called from Reconstructed API Class we indicate Covert Plugin Adapter to supercall the framework instead of performing multiple method dispatch.

The pseudocode of Listing 3.1 summarizes the collaboration as performed between Covert Plugin Adapter and Reconstructed API Class. Comparing to the code of Figure 3.8, Listing 3.1 omits the details of adapting mismatches between method signatures of Refactored API Class and Reconstructed API Class (e.g., relating *getID()* and *getIdentifier()*, or *manycast()* and *broadcast()*). Corresponding method signatures are related by the adaptation code emitted at adapter generation time, taking into account the information about API refactorings (as we will describe for refactoring-based adapter generation in Chapter 4).

### 3.1.4 White-Box Interface Adapter Pattern

#### Motivation

A plugin may reuse the framework by implementing an API interface, while framework methods implemented by the plugin may be called back from the framework. After refactoring the API interface, plugin

[9] Multiple method dispatch is required, because we assume the unavailability of plugins for analysis. Otherwise, we could have optimized the dispatch away by generating one specific Covert Plugin Adapter for each discovered plugin subclass.

[10] Upcalling methods are a particular implementation decision, chosen for its simplicity and efficiency. The method caller could be traced differently, for example, setting a flag indicating the calling site. However, this decision would imply carrying out the flag's synchronization, and additional method calls to manage the flag's value.

Listing 3.1: Class collaboration and multiple dispatch in the White-Box Class Adapter pattern

```
public class ReconstructedAPIClass{
  private CovertPluginAdapter adaptee;

  public ReturnType MethodName(Parameters) {
    //call reaching this point indicates Plugin Client
    //hence, invoke upcalling method of Covert Adapter Class
    return adaptee."upcall_"+MethodName(Parameters);
  }
}

public CovertPluginAdapter extends RefactoredAPIClass{
  //field type is Reconstructed API Class
  //field value is an instance of Plugin Subclass
  private ReconstructedAPIClass adaptee;

  //call comes from Framework Client, perform multiple method dispatch
  public ReturnType MethodName(Parameters) {
    //check for overriding method in the adaptee
    if (adaptee.getClass().defines(MethodName, Parameters)) {
      //call the overridden plugin method
      return adaptee.MethodName(Parameters);
    } else {
      //no overriding found
      return super.MethodName(Parameters);
    }
  }

  //call comes from Plugin Client, upcall the framework
  public ReturnType "upcall_"+MethodName(Parameters) {
    return super.MethodName(Parameters);
  }
}
```

functionality, which was previously called backed from the framework via methods of the API interface, may not be accessible in the upgraded framework.

Let us reconsider the API interface *IPacket*, the refactoring of which was discussed for the Black-Box Interface Adapter pattern in Section 3.1.2, but now assuming white-box framework reuse. In the first framework release, the interface contains a method *whoCreated()* with the return value of type *Node*. The interface is implemented by the plugin's class *SecurePacket*, where the implementation of the method *whoCreated()* returns the packet's creator node. In white-box framework reuse, the method *packetSeenInNode()* of the framework class *PacketFactory* may be invoked, with an instance of *SecurePacket* passed as the actual argument to the method formal parameter of type *IPacket*. On that instance, *PacketFactory* may invoke the method *whoCreated()* to obtain the packet's creator. (This example was introduced by Figure 2.5(b) on page 26.)

In the second framework release, to follow adopted naming conventions, framework developers rename *IPacket*'s method *whoCreated()* to *getCreator()* and update existing framework calls to the method. After upgrading the refactored framework, the call to the renamed method in the framework's *PacketFactory* will not locate and invoke the plugin implementation of the method (Non-localizable Plugin Functionality: Unimplemented Method, as shown for our example by Figure 2.8(b) on page 30). One solution would be to make the plugin's *SecurePacket* properly implement the refactored API interface and, thereafter, to recompile the plugin. However, this is not possible in case the plugin is not available or cannot be modified.

Instead, we could introduce two adapter types: a (reconstructed) adapter interface *IPacket* declaring the method *whoCreated()* with exactly the same signature as before the refactoring, and a class named, say, *CovertIPacketWhite*, implementing the framework's refactored interface *IPacket* (Figure 3.9). A plugin instance of *SecurePacket* is composed as an object of the adapter type *IPacket* within an object of the adapter class *CovertIPacketWhite*. When the plugin instantiates the class *SecurePacket* implementing

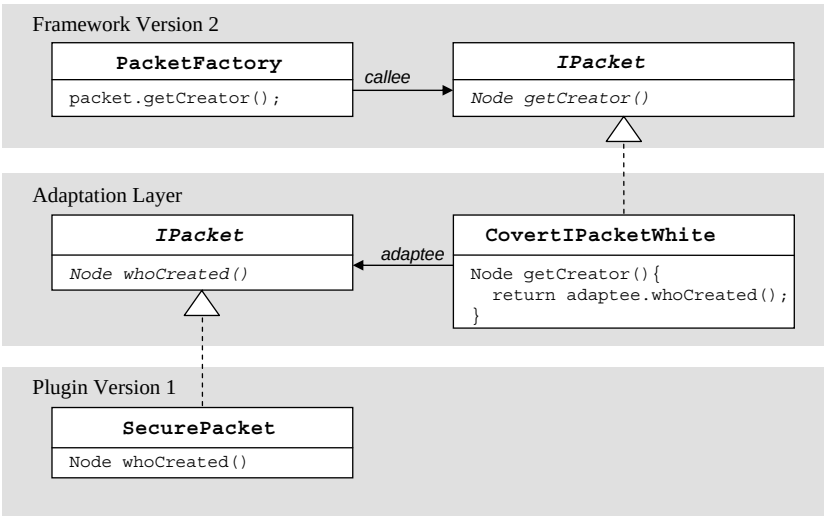



Figure 3.9: Application of the White-Box Interface Adapter pattern. The adapter class *CovertIPacketWhite* implements the refactored *IPacket* of the framework and uses the reconstructed adapter interface *IPacket* implemented by the plugin, enabling the refactored framework to access existing plugin implementation.

the adapter interface *IPacket* and returns the instance to the framework, it is wrapped in an instance of *CovertIPacketWhite*.

The problem caused by the aforementioned API interface refactoring is solved as following: In case the framework calls *getCreator()* (specified by the refactored interface *IPacket*) on the object of *CovertIPacketWhite*, the object will call the method *whoCreated()* on the object stored in its adaptee field, and then return the obtained value to the framework.

### Applicability

Use the White-Box Interface Adapter pattern, if

1. a framework API interface used by object composition (white-box framework reuse) is refactored, and
2. existing plugins cannot be updated intrusively.

### Structure

Figure 3.10 shows the structure of the White-Box Interface Adapter pattern, annotated (by stereotypes) with the names of the participants of the Adapter design pattern.

### Participants

- **Refactored API Interface** (Framework's *IPacket*)
  - defines the refactored API interface.
- **Reconstructed API Interface** (Adaptation Layer's *IPacket*)
  - reconstructs the API type of the unrefactored API interface.
- **Covert Plugin Adapter** (Adaptation Layer's *CovertIPacketWhite*)

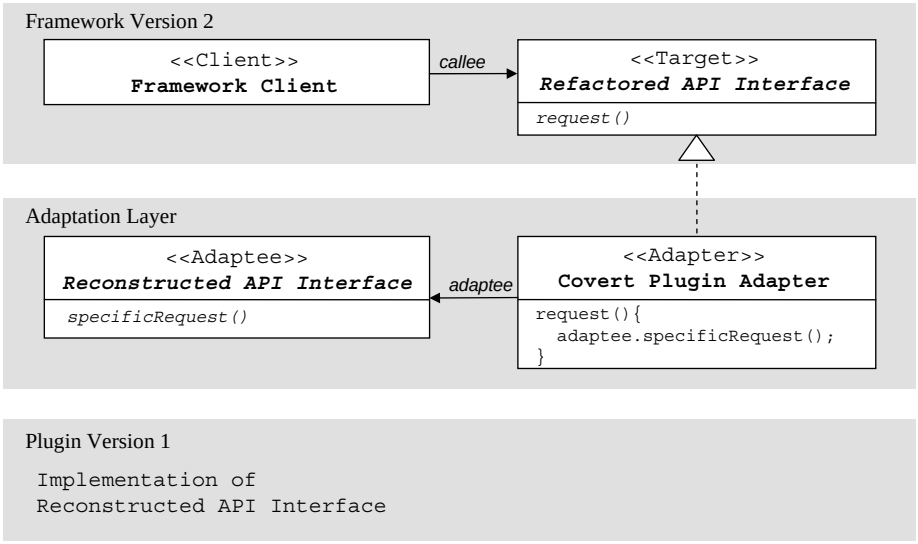



Figure 3.10: White-Box Interface Adapter pattern.

  - adapts the API type of Reconstructed API Interface to the API type of Refactored API Interface.

- **Framework Client** (Framework's *PacketFactory*)
  - collaborates with objects conforming to the API type of Refactored API Interface.

**Collaboration**

Framework Client calls operations on an instance of Covert Plugin Adapter. In turn, the instance calls operations of Reconstructed API Interface that carries out the requests.

### 3.1.5 Design Rationale of Black-Box and White-Box Adapter Patterns

In the previous sections of this chapter we made a step from the conventional Adapter design pattern to four novel adaptation patterns designated to compensate for application-breaking API refactorings in evolving Java frameworks. In doing that, we came from the general terms of the Adapter design pattern used to describe adaptation design in the context of component integration to the terms of the patterns for adapting refactored Java APIs (Table 3.1).

In this section, we describe in more detail why we came to the particular design of our adaptation patterns. We believe that explaining the design rationale will help to reuse design decisions when developing adapters for languages others than Java, and in different adaptation contexts. In particular, we will discuss how our main goal—binary backward compatibility of the refactored framework and its old plugins—and the particularities of the implementation platform (i.e., Java) impact the final design of our patterns. In the following section, we first look at the conceptual relations between Target, Adapter, and Adaptee (as defined in the Adapter design pattern [GHJV95, p. 141]), and examine particularities of realizing these relations in Java. In our discussions, we will omit the Client participant (and, therefore, its Framework Client and Plugin Client counterparts in our adaptation patterns), because Client simply indicates the calling site, while the particularities of its implementation do not add to the discussion of the patterns' design. Thereafter, we discuss how these particularities influence the design of our adaptation patterns. Armed with this knowledge, at the end of the section we will compare our design with a similar design developed with a different goal—to support refactoring of legacy code from inheritance to delegation [KS08].

| Patterns | Participants | Participants' Relations |
|---|---|---|
| Adapter pattern | Target, Adaptee, Adapter, Client | subtype, use, call |
| Black-Box Class Adapter pattern | Reconstructed API Class, Refactored API Class, Plugin Client | forward, call |
| Black-Box Interface Adapter pattern | Reconstructed API Interface, Refactored API Interface, Covert Framework Adapter, Plugin Client | implement, forward, call |
| White-Box Class Adapter pattern | Reconstructed API Class, Refactored API Class, Covert Plugin Adapter, Framework Client, Plugin Subclass, Plugin Client | extend, introspect, forward, call |
| White-Box Interface Adapter pattern | Reconstructed API Interface, Refactored API Interface, Covert Plugin Adapter, Framework Client | implement, forward, call |

Table 3.1: Terminology of the conventional Adapter pattern and the Black-Box and White-Box Adapter patterns. In the relations of the Black-Box and White-Box Adapter patterns, *call* stands for a client call, *forward* for a forwarding call, *implement* for implementing an interface, and *extend* for extending a class. *Introspect* is a special use relation used in the White-Box Class Adapter pattern to check for the presence of overriding plugin implementation.

### Java Adapters for Binary Backward Compatibility

To begin with, we should understand the conceptual relations between the types involved in adaptation and consider the particularities of our implementation platform (i.e., Java) to properly realize those relations in the Black-Box and White-Box Adapter patterns. Figure 3.11 summarizes the conceptual type relations between the key participants of the Adapter design pattern. While discussing the realization of these participants and their relations in the Black-Box and White-Box Class and Interface Adapter patterns, we will superimpose the participants of Figure 3.11 (denoted as stereotypes) and their relations on the UML class diagrams [Obj05] representing the patterns. The only extension to the standard UML notation is composing the stereotype names in dashed boxes and relating them by dashed arrows (as in Figure 3.11), to distinguish the relations between the participants of the Adapter design pattern from the relations between the participants of the Black-Box and White-Box Adapter patterns.

**Realizing subtype relation.** Quite obviously, an Adapter is a subtype of its Target since its interface contains the interface of Target (subtyping relation as defined by Gamma et al. [GHJV95, p. 13]), while its objects may substitute the objects of its Target in all programs without changing the program behavior (Liskov's substitution principle [Lis87]). Therefore, to achieve backward compatibility, we must realize Adapters as subtypes of their Targets.

When realizing subtype relation in Java adapters, we should consider that Java mixes the notions of class inheritance and subtyping. If, for example, a type is specified (and implemented) by a class, it is expected that the type's subtype is also specified (and implemented) by a class. Moreover, certain assumptions about types exist at the level of the language execution environment: for example, when locating a type, the Java

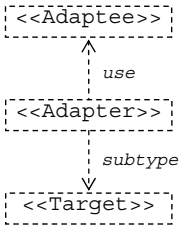



Figure 3.11: Subtype and use relations of the Adapter design pattern.

linker may rely on the information about the type's implementation (e.g., whether it is a class or an interface). Therefore, if the type is expected to be specified by an interface, substituting it with a class specifying the type directly (i.e., without implementing the interface) may lead to the linker not locating the replacing type's methods. Finally, since Java has a *nominative* type system [Pie02], the subtype relation must be declared explicitly, using predefined language keywords (i.e., *extends* and *implements* in Java).

**Realizing use relation.** While Adapter and Target are types related by subtyping, Adapter and Adaptee are types related by a use relation.[11] We formulate this use relation as follows: Whenever Adapter possesses the implementation of the method intended to be invoked on an instance of Target, Adapter uses its implementation for the method invocation; otherwise, Adapter uses the implementation of its Adaptee. In addition to reusing Adaptee's implementation, this definition allows Adapter to realize functionality not present in Adaptee.

Since after the API refactoring and adaptation the framework and its plugins are related via adapters, the realization of the conceptual use relation in adapters must be powerful enough to relate the both sites as before the API refactoring. Therefore, the realization of the use relation in our adapter patterns is directly influenced by the way the API is reused (i.e., by object composition or by inheritance) and corresponds to object composition (in black-box framework reuse) and to inheritance (in white-box framework reuse).

### Designing the Black-Box Adapter Patterns

In case of a refactored API class, where the old version was reused in a plugin by object composition, the plugin expects the Target interface (i.e., the API type, in our context) to be specified by an API class. Therefore, we need to create an adapter class specifying Target. Furthermore, to subtype Target the class of the corresponding Adapter must subclass the class specifying Target. As a possible solution, we could specify the Target interface by a separate class, with some default (e.g., dummy) method implementations. Afterwards, we could subclass that class by an adapter class that would override all methods of its base class with adapter methods.

Besides creating two classes, this solution has another drawback due to the fact that the plugin is aware of reusing an API class (and not an API interface). Namely, if a plugin attempts to instantiate the class by calling a class constructor, there is no way to make the plugin instantiate the class realizing Adapter instead, without either modifying the plugin's constructor call or using an additional technology (e.g., intercepting constructor calls and rewriting binaries at run time by an aspect-based tool, such as JAC [JAC]). However, remaining general and independent from additional technologies and tools (e.g., aspect orientation), we want to implement all our adaptation decisions in the base language (i.e., Java). Because we are in full control over the adapter construction, we realize both Target and Adapter as a single class, that is, Reconstructed API Class (Figure 3.12, the bottom part). After all, a type is a subtype of itself and realizing the adapter class as Reconstructed API Class does not affect the subtyping we require for adaptation. The class specifies the Target interface, defines dedicated type constructors (to properly instantiate the adapter class), and provides the corresponding implementation of the adapter methods. Or, seen from a different perspective, Reconstructed API Class includes the class realizing Adapter as a mixin [BC90].

Now we need to decide how to realize the use relation between Adapter and Adaptee, the API type of the latter specified and implemented by Refactored API Class (Figure 3.12, the top part). Once Reconstructed API Class is introduced, all plugin calls that were invoked on a framework object are invoked on an instance of Reconstructed API Class—a wrapper of that framework object. In black-box reuse, to perform its job a wrapper needs merely to invoke a method of its adaptee object, while the self reference of the method call invoked on the wrapper (and bound to the wrapper) does not need to be passed further to the adaptee object. Kniesel [Kni00, p. 15] calls this use relation *consultation* (as opposed to *delegation* [Lie86], in which the self reference would also be passed further on). In Java, consultation can be implemented by message forwarding (called by Kniesel *explicit resending* [Kni00, p. 17]), in our case, from wrappers to their adaptee objects. Figure 3.12 summarizes design decisions with regard to the implementation of subtype and use relations in the Black-Box Class Adapter pattern.

[11] Regardless of the Adapter pattern version (using object composition or inheritance), the pattern's Adapter is always a subtype of Target, but not of Adaptee. In the class version of the pattern, Adapter uses private inheritance that does not relate the types, only the implementation.

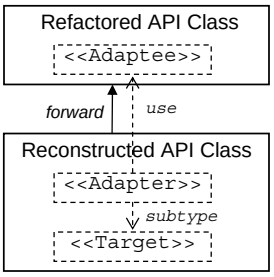



Figure 3.12: Subtype and use relations and their implementation in the Black-Box Class Adapter pattern.

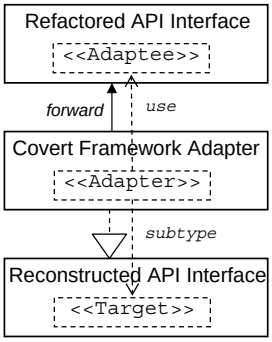



Figure 3.13: Subtype and use relations and their implementation in the Black-Box Interface Adapter pattern.

The design of the Black-Box Class Adapter pattern can be trivially adopted to the Black-Box Interface Adapter pattern coping with a refactored API interface implemented in the framework (Figure 3.13). In this case, since Adapter is always realized by a class, while Target is expected (e.g., by the linker) to be specified by an API interface, we cannot realize both Target and Adapter by a single adapter class. Instead, we reconstruct the interface as a separate implementation unit (Reconstructed API interface) and make the adapter class hidden in the adaptation layer (Covert Framework Adapter) implement that interface. The use relation to Adaptee (the API type specified by Refactored API Interface) is implemented as in the Black-Box Class Adapter pattern (i.e., by message forwarding) for the same reasons.

### Designing the White-Box Adapter Patterns

In contrast to black-box reuse, in white-box reuse Clients are possible at both sites (i.e., the framework and the plugins). In general, besides the methods that can be used directly on its instances, a class in a class hierarchy defines a two-sided *inheritance interface* [Hau93] (also called *specialization interface* [SG95]): one side defines class members inherited by subclasses and accessible by subclasses' clients, and the other side defines members of subclasses accessible from the base class' clients. Referring to our running example of the framework's *Node* subclassed by the plugin's *SecureNode* (as introduced by Figure 2.5 on page 26), one side of the *Node*'s inheritance interface consists of *getIdentifier()* and *getInfo()* inherited in the plugin, while the other side of the *Node*'s inheritance interface consists of *getNodeLocation()* defined (i.e., implemented) and *broadcast()* redefined (i.e., overridden) in the plugin.

The ability of accessing subclass functionality via the methods specified in the base class plays a key role in the implementation of many important engineering decisions (e.g., of the Template Method design pattern [GHJV95, p. 325]). However, it also lies at heart of the fragile base class problem [MS98], because by making assumptions about implementation particularities of base classes and subclasses, developers may commit themselves to semantics often hidden in and non-trivial to extrapolate from class implementations. In case such particularities are not understood by developers during application maintenance, applications

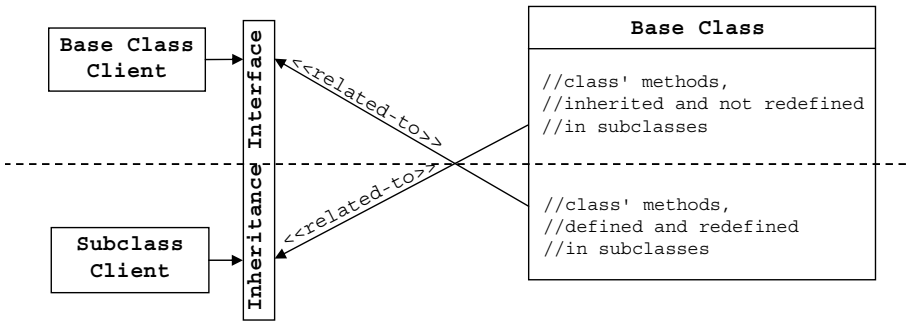



Figure 3.14: Inheritance interface of a base class in a class hierarchy (inspired by Hauck [Hau93]).

can be easily broken if the base class is replaced (e.g., upgraded to a new version).[12]

In particular, the refactoring of an API class can be seen as replacing its old version (called *unrefactored API class* throughout Section 3.1) with a refactored, improved version (i.e., *refactored API class*). After the refactoring, whereas one side of the class' inheritance interface is still defined by methods of old plugin subclasses, the other side is expressed by the refactored methods of the API class. By restructuring the class implementation that expresses only one side of the class' inheritance interface (the side defined in the API class, but not in its subclasses), API refactorings may introduce a mismatch between the two implementation sides of the inheritance interface. In our running example, by affecting the signatures of *getNodeLocation()* and *broadcast()*, and by introducing a new method *getDescription()*, refactorings of the API class *Node* introduce a mismatch between the *Node*'s two sides of the inheritance interface.

In general, to support safe class replacement, Hauck suggests to formally define an inheritance interface as a type specification, separate from class implementations [Hau93]. Given such specification that essentially describes the contract of a base class, developers relate (or *project*, in the terminology of Hauck [Hau93]) the implementation of a replacement class to the specification, stating explicitly how the class implements the contract specified by the inheritance interface (Figure 3.14). Thereafter, it is possible to ensure (e.g., using a type checker) that the replacement class fulfills all (type) constraints of the inheritance interface fulfilled by the class being replaced [Hau93].

If we had an explicit inheritance interface defined for a framework API class being refactored, we could use the inheritance interface to relate the refactored implementation of the API class and the old implementation of existing plugin subclasses. Via the inheritance interface, we would (re-)connect the refactored API class with its old subclasses and ensure the correctness of framework upgrade. However, in Java there is no explicit notion of the inheritance interface; it is defined implicitly by declaring certain class methods public and protected, and implementing some of these methods in subclasses (e.g., as in our running example of *Node* and *SecureNode*). Moreover, we do not want to burden framework developers with the task of writing additional adaptation specifications (in this case, the specifications of inheritance interfaces). Instead, we connect the refactored API class and its old plugin subclasses via the adapter classes of the White-Box Class Adapter pattern. In a sense, the pattern extrapolates an inheritance interface from a refactored API class and its old plugin subclasses, and relates to the extracted inheritance interface the actual implementation in the refactored framework and old plugins.

Using the naming conventions introduced when presenting the design of the White-Box Class Adapter pattern (Figure 3.7 on page 51), Figure 3.15 relates the notion of the inheritance interface to the participants of the Adapter pattern in the context of a refactored API class. The side of the class' inheritance interface inherited by a plugin subclass is the PlTarget interface expected by plugin clients. Similarly, the side of the class' inheritance interface accessed from the framework is the FrTarget interface expected by framework clients. Together, PlTarget and FrTarget comprise the (implicit) inheritance interface of the refactored API class in question. While PlTarget consists of all methods of the unrefactored API class inherited (and not redefined) by subclasses, FrTarget consists of all methods of the refactored API class, for which defining or redefining implementation may exist in old subclasses.

[12] For a set of excellent examples of the fragile base class problem we refer to Mikhajlov and Sekerinski [MS98].

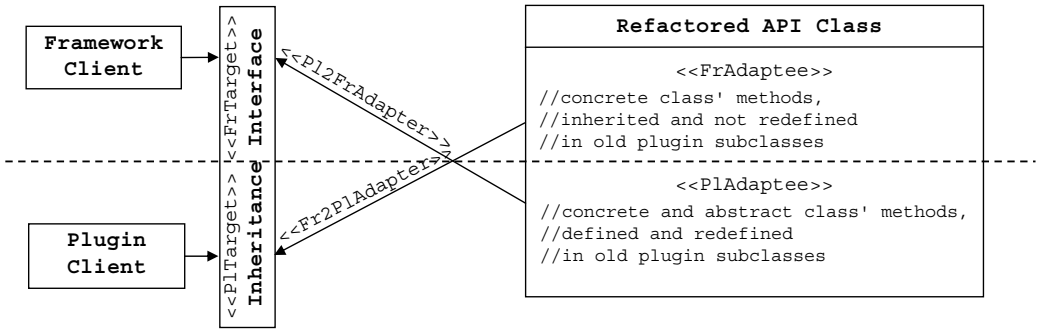



Figure 3.15: Inheritance interface and a refactored API class associated with the key Adapter pattern participants. Target participants are prefixed to reflect the calling site, while Adaptee participants are prefixed to reflect the site, where the required implementation is present ("Pl" for the plugin and "Fr" for the framework, respectively). At the same time, Adapter participants are prefixed to reflect the Adaptee-Target pair related by each Adapter ("Pl2Fr" and "Fr2Pl", respectively).

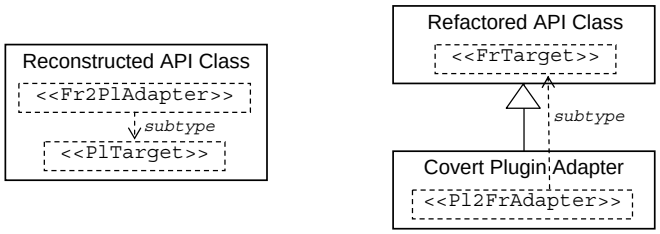



Figure 3.16: Subtype relations and their implementation in the White-Box Class Adapter pattern.

When realizing the types as expected by Java clients, since PlTarget and FrTarget are expected to be specified by classes (the unrefactored API class and the refactored API class, respectively), we need to specify them directly by classes (and not by separate adapter interfaces). The types of these two classes should then be subtyped by the types of classes realizing the corresponding Adapters (Fr2PlAdapter and Pl2FrAdapter, respectively), to implement the subtype relation of the Adapter pattern. Similar to the Black-Box Class Adapter pattern, and for the same reasons, Reconstructed API Class of the White-Box Class Adapter pattern realizes both the Target interface as expected by the plugin (i.e., PlTarget) and the corresponding Adapter (i.e., Fr2PlAdapter) as a single class (Figure 3.16, left part). At the same time, since FrTarget is defined by the refactored API class (called Refactored API Class in the pattern), Covert Plugin Adapter realizing Pl2FrAdapter subclasses Refactored API Class (Figure 3.16, right part).

As the final design step, we should implement the use relations of Adapters and their Adaptees (Figure 3.17). Fr2PlAdapter relates to its PlTarget the framework implementation of the inheritance interface represented by FrAdaptee (the upper use arrow in Figure 3.17). Similarly, Pl2FrAdapter relates to its FrTarget the plugin implementation of the inheritance interface represented by PlAdaptee (the lower use arrow in Figure 3.17). In the pattern realization, Reconstructed API Class realizing Fr2PlAdapter should use the functionality of FrAdaptee represented by Refactored API Class, while Covert Plugin Adapter realizing Pl2FrAdapter should use (via Reconstructed API Class) the overriding functionality of old plugin subclasses. Moreover, because we are adapting the two sides of an inheritance interface of the very same API class, the collaboration of the two adapter classes should (re-)connect the class hierarchy affected by API refactoring. This implies that the very same methods of the API class' inheritance interface invoked by calls from existing framework and plugin clients before the refactorings, will be invoked by those calls after the refactoring and pattern application.

Probably, the most intuitive solution in Java is to implement the use relations of the pattern as class inheritance, by extending Covert Plugin Adapter by Reconstucted API Class (Figure 3.18). Indeed, since Covert Plugin Adapter adapts functionality of the old plugin subclasses to the refactored base API class,

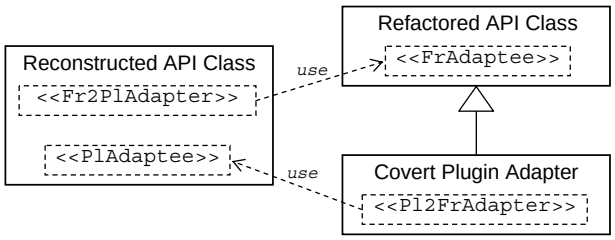
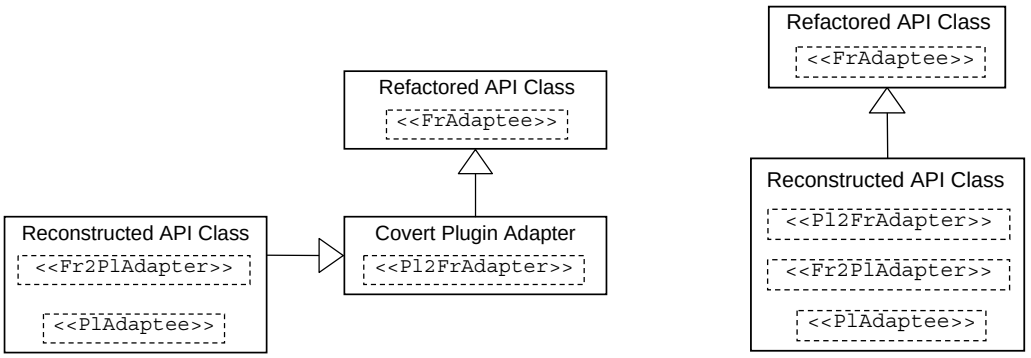



Figure 3.17: Use relations in the White-Box Class Adapter pattern.




(a) Relating adapter classes by subclassing.

(b) Realizing adapters as a single class.

Figure 3.18: Inheritance-based implementation of the use relations in the White-Box Class Adapter pattern. The adapters are inserted in the class hierarchy between the refactored API class and existing plugin subclasses, and are related by class inheritance.

while Reconstucted API Class adapts functionality of the refactored base API class to plugin subclasses, subclassing the former by the latter conceptually re-establishes the class hierarchy broken by API refactorings (Figure 3.18(a)). Moreover, this solution could be optimized by implementing the Adapters as a single class that inherits from the refactored API class, serves as the base class for the existing plugin subclasses, and realizes the adaptation logic to invoke the appropriate methods of the refactored API class and its plugin subclasses (Figure 3.18(b)).

Although intuitive, this inheritance-based solution has two severe disadvantages. First, we would have to solve class name collisions: although the name of Covert Plugin Adapter is not important (we could choose any unique name, or implement the both Adapters as a single class), the name of Reconstructed API Class is—it has to be named exactly as the API class before the refactoring. Unless the API class was renamed by a refactoring, the name of Refactored API Class will be exactly the same as of Reconstructed API Class prohibiting subclassing the former by the latter (indirectly, as in Figure 3.18(a), or directly, as in Figure 3.18(b)). We could solve this problem by renaming the whole refactored framework API (e.g., by moving it to another package) to disambiguate the names of all Refactored and Reconstructed API Classes. Although technically possible, we considered renaming the API undesirable.

While renaming the API is undesirable, the second disadvantage of this inheritance-based solution–the inability to protected from Unintended Functionality (discussed in Section 2.2.3)—is much severer. More specifically, this solution cannot cope with refactorings leading to Unintended Plugin Functionality: Method Capture (shown by Figure 2.10 on page 33). Recalling the problem, in case a newly introduced or refactored API method of the refactored API class has exactly the same signature as a semantically different method implemented in an old plugin subclass, the API method will be inadvertently overridden by the plugin method. When the framework invokes such a method on an instance of the plugin subclass, the self reference will be bound to the plugin instance, which will invoke its (inadvertently) overriding method, leading to an application malfunction. Relating the refactored API class and existing plugin subclasses via class

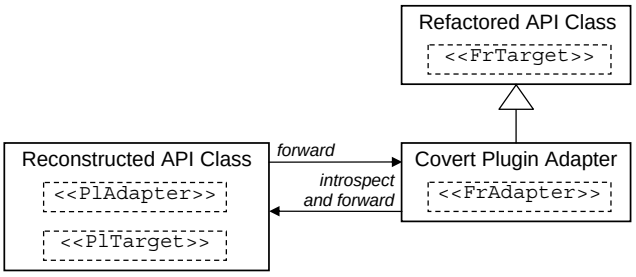



Figure 3.19: Delegation-based implementation of the use relations in the White-Box Class Adapter pattern. Delegation is realized by a combination of message forwarding between Reconstructed API Class and Covert Plugin Adapter, and the type introspection as performed in the multiple dispatch of Covert Plugin Adapter.

inheritance, we could not avoid accidental overriding, because we do not have control over the binding of the self reference and, hence, the methods to be invoked. Without implying additional technologies (e.g., aspect orientation), there is no way in this inheritance-based solution to intercept accidentally overridden methods called on a plugin instance, and invoke instead intended framework methods. Moreover, since we cannot analyze plugins statically, we could not even warn developers about a possible accidental overriding. Since inversion of control realized via the callback mechanism plays a crucial role in framework-based applications, we considered this disadvantage unacceptable.

What we require are adapters that are able to intercept the method calls coming from both sides of the inheritance interface of a refactored API class and redirect these calls appropriately. At the same time, we must implement the White-Box Class Adapter pattern in such a way, that, at least conceptually, the inheritance relation between the refactored API class and existing plugin subclasses is re-established.

To emulate the class inheritance, while giving adapters the required control power, we realize Reconstructed API Class and Covert Plugin Adapter as separate classes related by *delegation* (Figure 3.19). Delegation as a reuse mechanism has been shown to be at least as powerful as class inheritance [Lie86, Ste87, JZ91, GHJV95, KS08].[13] In particular, Gamma et al. argue that "delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse" [GHJV95, p. 21]. In our adaptation context, using delegation changes the binding of method receivers, since now adapters can intercept all calls to the methods of the inheritance interface and react on these methods appropriately. Whenever the plugin calls a method of the inheritance interface, the call will reach Reconstructed API Class, which will forward the call to a method of Covert Plugin Adapter invoking the functionality of the refactored API class. Whenever the framework calls a method of the inheritance interface, the call will reach Covert Plugin Adapter, which will introspect its plugin instance, and will forward the call to the overriding functionality of an old plugin subclass (if present) or to the functionality of the refactored API class (otherwise).

Besides the ability to intercept calls to the methods of the inheritance interface, another important consequence of using delegation is the separation of the name spaces of the framework and the plugin. When related by inheritance, Refactored API Class and Reconstructed API Class belong to the same name space and there is no other way to disambiguate them than by giving them distinct names. As we discussed, this would require renaming possibly all Refactored API Classes of the framework API. By using delegation, we effectively create two name spaces: that of the framework and that of the plugin. Therefore, we can solve the aforementioned name collisions by disambiguating types using the name spaces, from which the types originate. In Java, this can be realized using different class loaders to load separately framework and plugin types (the latter, together with the corresponding reconstructed API types). As a consequence, no renaming of the refactored API is required. This solution (described in detail in Section 5.1.2) would not be possible, if the refactored API class and existing subclasses were related by inheritance.

[13] Lieberman [Lie86] argues that delegation is more powerful than inheritance.

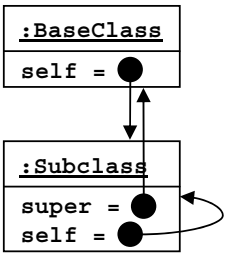



Figure 3.20: Modeling inheritance by delegation (inspired by Hauck [Hau93]). The black circles denote values of instance variables and the arrows denote the use relation. The subclass accesses the inherited functionality using its *super* variable, while the base class accesses the overriding functionality using its *self* variable.

Technically, our implementation of the delegation is very much in the spirit of Hauck, who suggested to use delegation (calling it *aggregation*) to represent class inheritance in prototype-based languages [Hau93]. Figure 3.20 summarizes his approach using a visual notation similar to UML object diagrams [Obj05]. In Hauck's approach, a base class (represented by an object) defines an instance variable *self*, while a subclass (represented by another object) defines an instance variable *super*, both variables with a type of an inheritance interface. In addition, the subclass defines an instance variable *self*, the type of which is an inheritance interface conformant to the type of *super*. By relating class implementations to the inheritance interfaces, the two classes are defined to be of types of their (conformant) inheritance interfaces. When creating an object of the subclass, its *self* is bound to the object itself and its *super* is bound to the *self* of an object of the base class. As a consequence, the object of the subclass can access the inherited methods using the *super* variable, while the object of the base class can access the overriding functionality using its *self* [Hau93].

Relating our design to the terminology of Hauck, Covert Plugin Adapter is the base class and Reconstructed API Class is its subclass. The adaptee field of Reconstructed API Class containing an instance of Covert Plugin Adapter corresponds to the *super* variable in the subclass, while the adaptee field of Covert Plugin Adapter containing an instance of Reconstructed API Class corresponds to the *self* variable of the base class in the Hauck's terminology. These two explicit variables in combination with message forwarding are used to implement delegation. One difference in our implementation is that we do not have to explicitly handle the definition of the *self* variable for the subclass (i.e., Reconstructed API Class), since in Java such variable (i.e., *this*) is implicitly defined and automatically bound to the objects of subclasses of Reconstructed API Class.

The most important differences to the work of Hauck are caused by the fact that we do not know the inheritance interface a priori and, moreover, we cannot infer it by statically analyzing plugins. This lack of knowledge impacts the static design of Reconstructed API Class, which has to define forwarding methods for *all* API methods defined in the unrefactored API class (because we cannot statically check, which methods of the unrefactored API class were indeed inherited but not overridden in the old plugin). This design decision does not decrease performance, since at run time Reconstructed API Class will forward indeed only the methods inherited in the plugin—all calls invoked on a plugin object to methods overridden in plugin subclasses will not reach Reconstructed API Class. However, at the other side (that of the framework) we have to infer the knowledge about the inheritance interface as implemented in plugin subclasses to decide, whether overriding plugin functionality exists. Because plugins are not available for static analysis, we need to perform the analysis at run time by using introspection on the adaptee variable of Covert Plugin Adapter pointing to a plugin object.

As a technical nuisance, in Java a subclass may explicitly call a superclass method using the keyword *super*. Although this call will reach Reconstructed API Class, simply forwarding it to Covert Plugin Adapter may not be appropriate in case the method is also overridden in the plugin. In this case, since Covert Plugin Adapter is not aware of the fact that this is a supercall, it will attempt to call back the overriding method in the plugin. This will result either in unintended semantics of the overridden plugin method, or in an

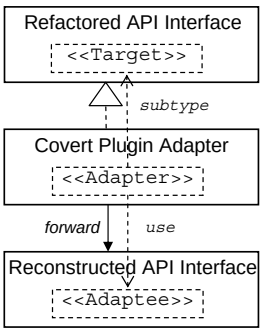



Figure 3.21: Subtype and use relations and their implementation in the White-Box Interface Adapter pattern.

infinite call loop in case the supercall comes from that overriding method (as we showed in an example in the Collaboration section of the White-Box Class Adapter pattern on page 54). To solve this problem, we exploit the fact that Reconstructed API Class is aware about the supercall (this is indicated by the call reaching Reconstructed API Class) and can, using an *upcalling method*, indicate Covert Plugin Adapter to supercall the framework (Section 3.1.3, page 52).

Finally, the design of the White-Box Class Adapter pattern can be trivially adopted to the White-Box Interface Adapter pattern coping with API interfaces implemented in plugins and later refactored in the API (Figure 3.21). Since Target is specified by an API interface (Refactored API Interface), to realize subtyping we make Covert Plugin Adapter implement the interface. At the same time, we reconstruct Adaptee (Reconstructed API Interface), the plugin implementation of which should be accessed from Covert Plugin Adapter. In contrast to class inheritance, in this case Target can only be called from the framework. Therefore, to implement the use relation between Adapter (Covert Plugin Adapter) and its Adaptee (Reconstructed API Interface) it suffices to use message forwarding similar to the one in the Black-Box Interface Adapter pattern, but now forwarding from the adapter to the plugin.

#### Design Similarity When Replacing Inheritance With Delegation

The design of the White-Box Class Adapter pattern resembles the design developed by Kegel and Steimann [KS08] to support the application of a specific refactoring—the refactoring of inheritance to delegation [Opd92, JO93, FBB+99, GS99, KS08]. Astonishingly, although with a motivation apparently different from that of Kegel and Steimann (adaptation instead of refactoring), we independently and simultaneously[14] developed a similar design. Not believing in a simple coincidence, we ought to take a closer look at the causes of this similarity to understand the common intrinsic properties of the two works.

In his PhD thesis, Opdyke introduces the refactoring of inheritance to delegation as "converting an association, modeled using inheritance, into an aggregation" [Opd92, p. 103]. Given two classes related by inheritance, the refactoring creates in the subclass a private field of the superclass, uses this field to access the previously inherited behavior in the subclass, and eliminates the inheritance relation between the two classes. The refactoring is furthermore described, under different names, by Opdyke and Johnson (as `ConvertingInheritanceToAggregation` [JO93]), Fowler et al. (as `ReplaceInheritanceWithDelegation` [FBB+99, p. 287]), and Genßler and Schulz (as `TransformingInheritanceIntoComposition` [GS99]).

By considering the consequences of applying this refactoring to a base class in a class hierarchy, Kegel and Steimann take a deeper look at the necessary preconditions of the refactoring to ensure its correctness [KS08]. They point out that in case of callbacks from the base class (called Delegatee after the refactoring) to the overriding functionality of the subclass (called Delegator after the refactoring), true delegation, and not

[14] At the time the paper of Kegel and Steimann [KS08] was first presented, our paper introducing black-box and white-box adapters [ŞRGA08] was in the review phase. As a terminology remark, initially we were calling Covert Plugin Adapter of the White-Box Class Adapter pattern *Dispatcher* [ŞRGA08].

a simple message forwarding from Delegator to Delegatee, must be implemented. Otherwise, in case the self reference is bound to an object of Delegatee, the overriding functionality in Delegator may fallaciously remain not invoked. Kegel and Steimann implement delegation in Java by subclassing Delegatee by a special inner class in Delegator [KS08]. The inner class knows all methods that were overridden in the former subclasses of its superclass and forwards back those messages to Delegator, whenever the message receiver is the object of Delegatee. At the same time, Delegator forwards to its inner class all methods not redefined in Delegator. Since inner classes in Java have access to the state of their outer classes, the combination of the forwarding of Delegator and the reverse forwarding of its inner class subclassing Delegatee can effectively implement delegation [KS08].

Indeed, while Kegel and Steimann consider replacing inheritance with delegation, we in fact attempt to reconstruct the inheritance affected by API refactorings. However, in both cases there is a change of a base class in the class hierarchy. Being aware of the associated fragile base case problem [MS98], we are concerned with how to avoid it by a proper implementation of delegation. Similar to Kegel and Steimann [KS08], we want to neither imply additional technology (e.g., aspect orientation to intercept messages) nor extend the language itself (e.g., as suggested by Aldrich and Donnelly [AD04]), and we cannot use any formal specification of the inheritance interface (as suggested by Hauck [Hau93], and by Stata and Guttag [SG95]). Implementing delegation in Java as a language without native support for it, we naturally ended up in a similar design. Relating the design of the White-Box Class Adapter pattern to the design of Kegel and Steimann [KS08], Refactored API Class performs the job of Delegatee, Covert Plugin Adapter of Delegatee's inner class, and Reconstructed API Class of Delegator.

Still, due to different goals and the application context, our design has several important differences from the design presented by Kegel and Steimann [KS08]. To start with, our forwarding methods are adapter methods, the implementations of which consider refactorings applied to the base API class to compensate for possible syntactic mismatches introduced by API refactorings. Moreover, we implement adapters as separate classes (and not as outer and inner classes) avoiding the name collision of Reconstructed API Class with Refactored API Class, as discussed in Section 3.1.5. More important, we do not have access to the plugins and cannot infer their overriding functionality; in other words, we cannot statically compute the inheritance interface. Therefore, we have to implement multiple dispatch in Covert Plugin Adapter. Our multiple dispatch is more flexible than the static decisions taken in the inner class of Delegator, since we need not only to locate the intentionally overriding plugin methods, but also to avoid accidental overriding. The latter was not a concern of Kegel and Steimann, because they could always statically check for possible accidental method overriding. Finally, Kegel and Steimann discuss, that "it is debatable whether the (hidden) subclassing of Delegatee [by the inner class] is acceptable for the purpose of the refactoring [because] it does not replace inheritance, but only moves it to a new class" [KS08]. Although indeed debatable for the refactoring, this subclassing is a must in the White-Box Class Adapter pattern.

## 3.2 Adaptation Design Compensating for a Refactored Framework API as a Whole: Exhaustive API Adaptation

While in the previous section we introduced adaptation patterns compensating for refactorings of a particular API type, in this section we will consider static and run-time issues involved in compensating for refactorings of a framework API as a whole. Statically, "adapter [...] expresses its interface in terms of the adaptee's" [GHJV95, p. 137]. At run time, an adapter object wraps a "foreign" object (in the sense of a type unknown to Client), matches Client's invocation requests to methods defined in the Target interface, and invokes corresponding methods of the wrapped object.

What should be considered a foreign object to be wrapped in case of a refactored framework API? A framework object of a refactored API type may be considered foreign from the point of view of a plugin, since, due to the type refactoring, modified method signatures of the object's type may no longer match old method invocation requests in the plugin. Similarly, an object of a plugin type subtyping an old API type (as before the refactoring) may be considered foreign from the point of view of the refactored framework, since old method signatures of the object's type may no longer match new or modified invocation requests

in the framework (in case of callbacks). Furthermore, an object becomes foreign only when crossing the boundaries of its domain of origin: either from the framework to the plugin, or from the plugin to the framework.

To compensate for API refactorings, we want to wrap all foreign objects crossing domain boundaries into adapter objects, which we call *wrappers*.[15] Effectively, we separate the object domains of the framework and plugin: no plugin object may enter the framework (object) domain and no framework object may enter the plugin (object) domain without being encased into a corresponding wrapper. By "corresponding" wrappers we mean that their adapter types depend on whether the objects being wrapped originate from the framework or from the plugin.

- An object originates from the framework, if its class is defined in the framework; its corresponding *framework wrapper* is an instance of a black-box adapter.
- An object originates from the plugin, if its class is defined in the plugin; its corresponding *plugin wrapper* is an instance of a white-box adapter.

**Definition 1** (Object wrappers)
A wrapper is an adapter object—an instance of an adapter, placed around an adaptee object. A framework wrapper is an instance of either Reconstructed API Class of the Black-Box Class Adapter pattern or Covert Framework Adapter of the Black-Box Interface Adapter pattern, placed around a framework object. A plugin wrapper is an instance of Covert Plugin Adapter of either the White-Box Class Adapter pattern or the White-Box Interface Adapter pattern, placed around a plugin object.

Because a framework API usually offers many possible ways of reusing the framework functionality, a plugin typically reuses only a subset of the whole framework API. Moreover, the plugin may reuse some API types by composition and others by inheritance. Since we assume that in general we do not have access to plugins, we cannot statically discover, how they were reusing the API of the framework before its refactoring. As a consequence, we do not know a priori, which wrappers will be required at run time (or, in other words, which adapters will need to be instantiated).

To generalize our adaptation to the framework API as a whole, we introduce *exhaustive API adaptation*—a set of static and run-time decisions for binary backward-compatible adaptation of refactored framework APIs. Its main adaptation decision is to create the *exhaustive adaptation layer* by thoroughly considering all possible situations, where the adaptation may be required, and to place this layer between the refactored framework and old plugins. The layer can be seen from two time perspectives: static (type creation) and dynamic (object creation). In the following, we call the former *static adaptation layer* and the latter *run-time adaptation layer*.

**Static adaptation layer.** An important consequence of applying the Black-Box and White-Box Adapter patterns is that we reconstruct the API types of the old API as they were before the API refactoring, resulting in the static adaptation layer that completely reconstructs the old API. Figure 3.22 shows black-box and white-box adapters created for two refactored API types: a concrete class *Node* and an interface *IPacket* (interface adapters for *INode* are omitted). The layer adapts the refactored API under an old API (to be used from the plugins) and the old API under the refactored API (to be used from the framework). Moreover, for every possibly refactored API type of the latest framework version we apply *both* Black-Box and White-Box Adapter patterns. By a "possibly refactored API type" we mean that the API type could have been refactored by framework developers. We call such API types *user-defined*, in contrast to library types, such as language built-in primitive, special and reference types (e.g., *int*, *void* and *Swing* types in Java), and types of third-party libraries, which (we assume) cannot be evolved by framework developers.[16] With the notable exception of library collection types addressed in detail in Section 3.2.3, library types require no adaptation. Finally, we do not create adapters for newly introduced API types; as a consequence, such types

[15] In the original description of the Adapter pattern, Adapter and Wrapper are synonyms [GHJV95, p. 139]. We use (non-capitalized) term *wrapper* to better distinguish between adapters (as classes) and adapter objects (i.e., wrappers), the latter as instances of adapters placed around wrapped objects.

[16] As a future work, we will consider a possible combination of our approach with the on-going work on adaptation of evolving library types [BTF05, KETF07].

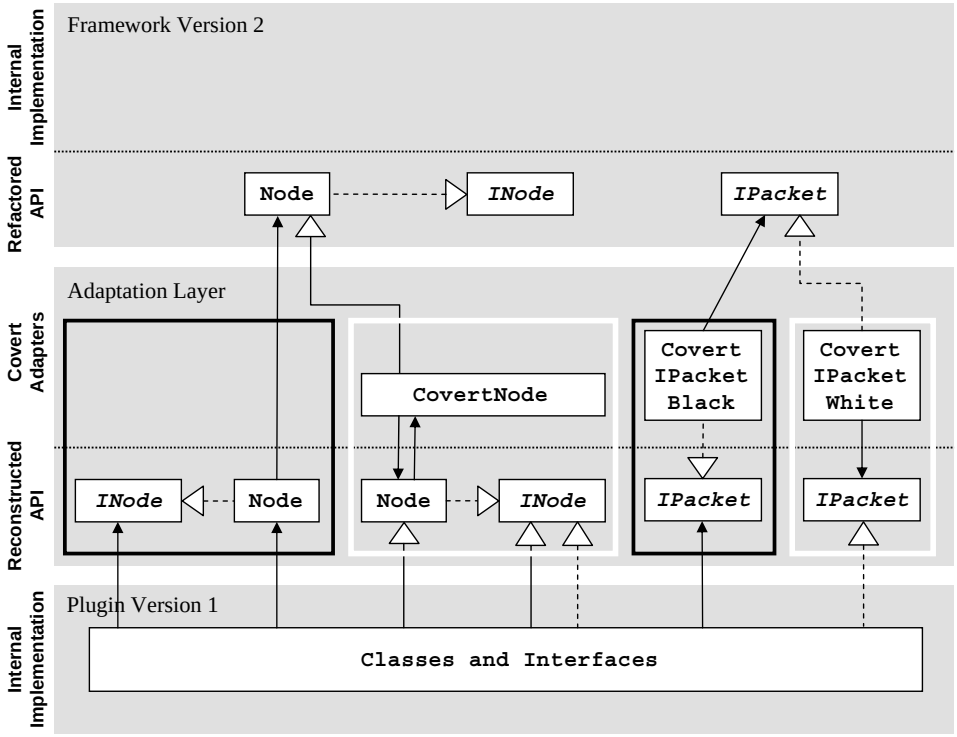



Figure 3.22: Static adaptation layer. The layer reconstructs the old API and reuses the refactored API. Black and white frames in the static adaptation layer group black-box and white-box adapters. Since we cannot statically analyze plugins, the relation between the reconstructed API of the adaptation layer and the plugin is of a *possible* reuse.

are not present in the reconstructed API and their names cannot conflict with the names of existing plugin types (which contributes to solving the Type Capture problem, as discussed at the end of Section 2.2.4 on page 36).

**Run-time adaptation layer.** At application execution time, depending on the domain of origin of an object that crosses domain boundaries and needs to be wrapped we instantiate the corresponding adapter of the static adaptation layer. All created wrappers make up the run-time adaptation layer (Figure 3.23). Framework wrappers adapt framework objects of refactored API types to the old plugin, while plugin wrappers adapt plugin objects of reconstructed API types to the refactored framework.

The central task of the run-time adaptation layer is the *complete* (with regard to the user-defined API types) separation of the framework and plugin object domains: besides objects of library types, plugins can only see their objects and framework wrappers, while the framework can only see its objects and plugin wrappers. In more detail, any framework object sent to the plugin is wrapped by a framework wrapper before being passed to the plugin. If a framework wrapper is later passed back to the framework, the adapted framework object is unwrapped from the wrapper before being sent to the framework. Similarly, any plugin object sent to the framework is wrapped by a plugin wrapper before being passed to the framework. If a plugin wrapper is later sent back to the plugin, the adapted plugin object is unwrapped from the wrapper before being passed to the plugin. This *wrapping* and *unwrapping* is done in the run-time adaptation layer and is completely transparent for both sites (i.e., the framework and the plugin).

Usually, most framework and plugin objects remain in their object domain and need no adaptation. The only exception is when one of the two sites (the framework or the plugin) invokes functionality of the other site, possibly sending in its objects or obtaining objects from the other domain. There are exactly two cases, when originally (before the API refactoring) objects may cross domain boundaries and, hence, require wrapping in the run-time adaptation layer:

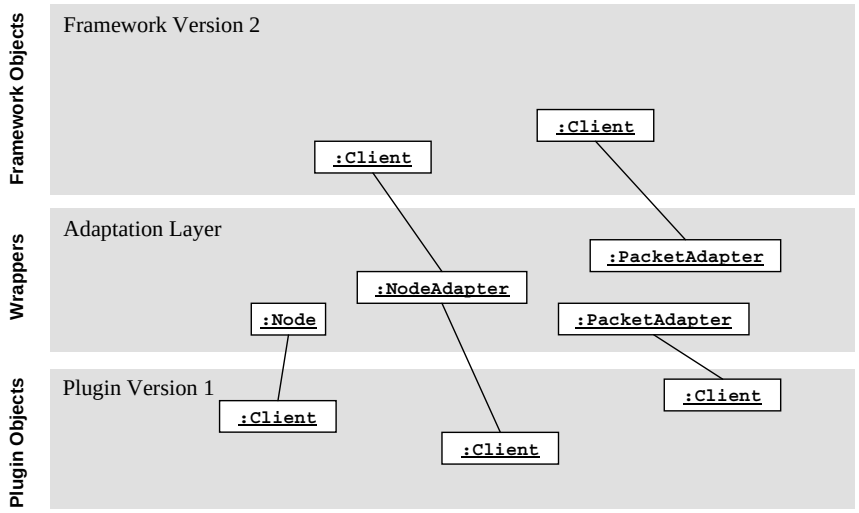



Figure 3.23: Run-time adaptation layer. Clients denote framework and plugin objects requesting functionality of another domain, while the objects of the run-time adaptation layer are framework and plugin wrappers.

1. A constructor of a framework API class is invoked from the plugin. In this case, the object obtained in the plugin originates from the framework, while the caller is always a plugin type.

2. A method defined in one domain is invoked from the other domain. In this case, objects passed as method arguments or return values may originate from the opposite domain. In contrast to an API constructor invocation, where only a plugin caller is possible, in case of a method invocation both the plugin and the framework (the latter, via callbacks) may act as callers.

In both cases the adaptation is required because the caller (of a constructor or a method) and the callee to be invoked are in different domains. Since the run-time adaptation layer completely separates the object domains of the framework and plugin, the calls originally invoking constructors and methods defined in the opposite domain will now result in invoking *adapter constructors* and *adapter methods* of wrappers. Thereby, the run-time adaptation layer intercepts all possible calls (i.e., constructor invocations and method invocations) between the two domains and instantiates adapters of the static adaptation layer, whenever needed.

**Adaptation layer as a whole.** Considering the adaptation layer as a whole, while its static adaptation layer reconstructs the old API and provides adapters that may need to be instantiated, its run-time adaptation layer instantiates these adapters to wrap objects crossing domain boundaries. A key facility of the adaptation layer is the *adapter cache*—a storage of the information about types and objects involved in adaptation, created and updated along with the creation of the static and run-time adaptation layers (Figure 3.24).

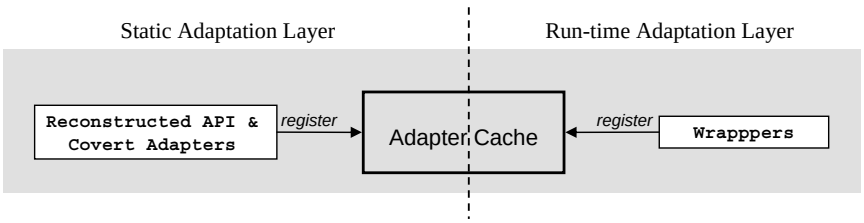



Figure 3.24: Adapter cache. The cache contains the information about all types of the static adaptation layer and all wrappers of the run-time adaptation layer.

Listing 3.2: Adapter cache

```
1  public class AdapterCache{
2    //adaptees and their adapters
3    private Map adaptees2adapters;
4
5    //wrapped objects and their wrappers
6    private Map wrapped2wrappers;
7
8    //cache wrapped objects and their wrappers
9    public void putWrapper(Object wrapped, Object wrapper){
10     wrapped2wrappers.put(wrapped, wrapper);
11   }
12
13   //given an object to be wrapped, return the corresponding wrapper
14   public Object getWrapper(Object toWrap){
15     //details specified in the following sections
16   }
17 }
```

- At adapter generation time, the adapter cache is created and filled in with the information about all types of the static adaptation layer as well as the correspondence of the adapters and their adaptees.
- At application execution time, the adapter cache is instantiated (as a singleton class) and constantly updated with the information about the created wrappers and their adaptee objects.

In the rest of this section we will discuss the design of the adapter cache (Section 3.2.1), object adaptation as done in adapter constructors (Section 3.2.2) and adapter methods (Section 3.2.3), and proper adapter creation and instantiation (Section 3.2.4). We will conclude the section with an example of exhaustive API adaptation: the adaptation of a framework API reused in a plugin both by object composition and inheritance (Section 3.2.5).

### 3.2.1 Adapter Cache

When an object needs to be wrapped, its type is considered in order to instantiate the corresponding adapter. The information about the type correspondence of adaptees and adapters is derived based on the history of API refactorings, when statically deriving the adapters. In the generated static adaptation layer, this information is stored in the adapter cache (line 3 of Listing 3.2) to be used at application execution time, when creating wrappers of the run-time adaptation layer.

The ability of the adapter cache to store the information about the wrappers of the run-time adaptation layer is shown by lines 6–16 of Listing 3.2. The reason for caching wrappers is to avoid creating different wrappers for the same object repeatedly crossing domain boundaries (e.g., returned by two API methods). Besides unnecessarily consuming system resources, creating different wrappers may also break existing applications that rely on *object identity*, since their comparison may fallaciously lead to inequality (although they wrap the very same object). Instead of creating a new wrapper every time a framework or a plugin object requires adaptation, we check, whether a wrapper for the object in question already exists in the adapter cache, and, if so, reuse that wrapper. Listing 3.2 shows two methods of the adapter cache related to storing wrappers (*putWrapper()*, line 9) and retrieving wrappers (*getWrapper()*, line 14). The latter method, depending on the presence of a wrapper for the given object, will either return the cached wrapper or instantiate the corresponding adapter.

### 3.2.2 Wrapping Newly Created Instances in Constructors of Black-Box and White-Box Adapters

As mentioned in the introduction of Section 3.2, one of the two cases when object wrapping is required is a class constructor defined in the framework and invoked from the plugin. By contrast, when framework

Refactored API Class

```
public class refactoredAPI.PacketFactory {
 String message;

 public PacketFactory() {
   this.message = "";
 }

 public PacketFactory(String message) {
   this.message = message;
 }
}
```

Reconstructed API Class

```
public class PacketFactory {
 refactoredAPI.PacketFactory adaptee;

 public PacketFactory() {
   adaptee = new refactoredAPI.PacketFactory();
   //cache itself as adaptee's wrapper
   AdapterCache.putWrapper(adaptee, this);
 }

 public PacketFactory(String msg) {
   adaptee = new refactoredAPI.PacketFactory(msg);
   //cache itself as adaptee's wrapper
   AdapterCache.putWrapper(adaptee, this);
 }
}
```

Figure 3.25: Wrapping newly created framework instances in constructors of black-box class adapters. After instantiating the corresponding framework type, an adapter constructor stores the framework instance in the adaptee field of the wrapper and updates the adapter cache. For readability, the adaptee's name is prefixed with *refactoredAPI.* to denote that the adaptee is in a different name space than its adapter (name spaces of the refactored API and the reconstructed API, respectively).

objects are created in the framework itself, they belong to the framework domain and require no wrapping at the time of their creation.[17] Therefore, wrapping at the time of constructor invocation is needed only when the plugin instantiates (1) an API class defined in the framework (black-box framework reuse), or (2) a plugin class subclassing an API class (white-box framework reuse).

Before upgrading the refactored framework, such constructor invocation executes a constructor of an API class, either directly (in black-box framework reuse) or via a constructor call chain as implemented in Java (in white-box framework reuse). In exhaustive API adaptation, since the static adaptation layer completely reconstructs the old API, any API constructor invocation in the plugin will eventually reach and execute an adapter constructor of the reconstructed API, which will then perform wrapping of the newly created (framework or plugin) instance.

Let us consider two examples (one for black-box framework reuse and one for white-box framework reuse) as excerpts from RefactoringLAN (Appendix A) and then discuss the conceptual differences in the wrapping for each of the two cases. In both cases, since a constructor can only be invoked on classes and not on interfaces, the instantiated adapter is always a *class* adapter.

- **API constructor invocation in black-box framework reuse.** Originally (before upgrading the refactored framework) the plugin invoked the constructor on an API class *PacketFactory*. Since the static adaptation layer completely reconstructs the old API, after the adaptation the old constructor invocation in the plugin invokes the constructor of the adapter *PacketFactory* (Figure 3.25). The latter is present in the static adaptation layer as Reconstructed API Class of the Black-Box Class Adapter pattern (the pattern introduced in Section 3.1.1 on page 43). In turn, the invoked adapter constructor is responsible for instantiating the corresponding adaptee (the framework's *PacketFactory*). As a consequence of the constructor invocation in the plugin, the adapter *PacketFactory* is instantiated and its instance—a framework wrapper encasing an instance of the framework's *PacketFactory*—is cached.

- **API constructor invocation in white-box framework reuse.** Originally (before upgrading the refactored framework) the plugin invoked the type constructor of a plugin class *SecureNode* subclassing API's *Node*. Since the static adaptation layer completely reconstructs the old API, after the adaptation the old constructor invocation of *SecureNode* invokes (according to the constructor call chain in Java) the constructor of its superclass *Node* (Figure 3.26). The latter is present in the static adaptation layer as Reconstructed API Class of the White-Box Class Adapter pattern (the pattern introduced

[17] They may require wrapping later, if sent to the plugin, as discussed in the following Section 3.2.3.

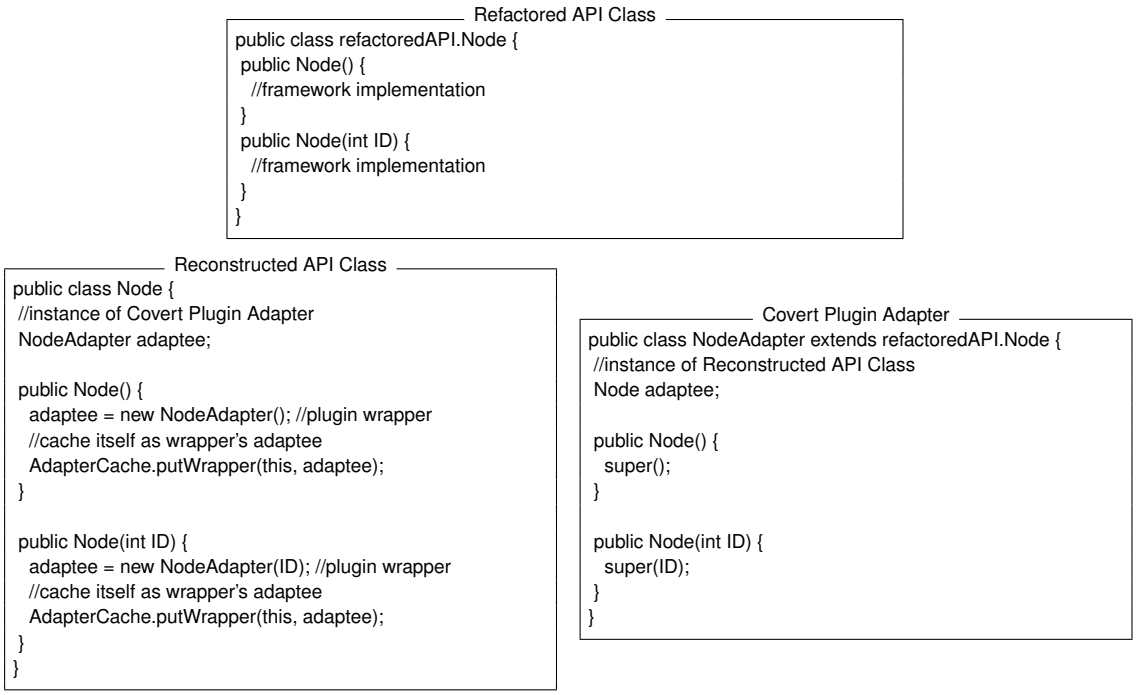



Figure 3.26: Wrapping newly created plugin instances in constructors of white-box class adapters. When a constructor call chain from a plugin subclass reaches a constructor of Reconstructed API Class, the constructor instantiates Covert Plugin Adapter. While the latter is used as adaptee of Reconstructed API Class to call inherited framework functionality, its instance is at the same time a wrapper of the created plugin instance.

in Section 3.1.3 on page 47). According to the pattern's design, the reconstructed *Node* will in turn create and store as its adaptee object an instance of *NodeAdapter* (Covert Plugin Adapter participant of the White-Box Class Adapter pattern) that inherits from the refactored API class *Node*. The created instance of *NodeAdapter* is then used to access the functionality that should be inherited from the framework's *Node* in the plugin's *SecureNode*. At the same time, the very same instance of *NodeAdapter* should be used as the plugin wrapper of the newly created *SecureNode* object, in case the latter is sent to the framework at some later point in time.

In both cases, adapters are instantiated at the time a constructor is invoked in a plugin. However, strictly speaking, the actual object wrapping is required only in black-box framework reuse: since the object to be created in the plugin belongs to the framework domain, the object must be immediately wrapped at the time of constructor invocation. By contrast, in white-box framework reuse, since the class of the object to be created is defined in the plugin, the object belongs to the plugin domain and no wrapping is needed at the time of constructor invocation. However, Covert Plugin Adapter (the adapter class *NodeAdapter*, in our example of Figure 3.26) still needs to be instantiated to access the inherited framework functionality, via its upcalling methods (as discussed in the Collaboration of the White-Box Class Adapter pattern, Listing 3.1 on page 55). Since the created plugin object may later be sent to the framework (requiring wrapping, as discussed in the next Section 3.2.3), we cache the created instance of Covert Plugin Adapter as a plugin wrapper of the newly created plugin object. As a consequence of this decision, for *all* objects of a plugin class subclassing a reconstructed API class, their wrappers already exist in the cache and can be reused, whenever needed.

**Choosing between instantiating a black-box and a white-box class adapter.** So far in our discussion we assumed that it is known, whether the reconstructed API class, a constructor of which is invoked in the plugin, is a black-box adapter or a white-box adapter. In practice, however, before creating and wrapping an adaptee instance in an adapter constructor, we have to decide, which pattern (i.e., the Black-Box Class

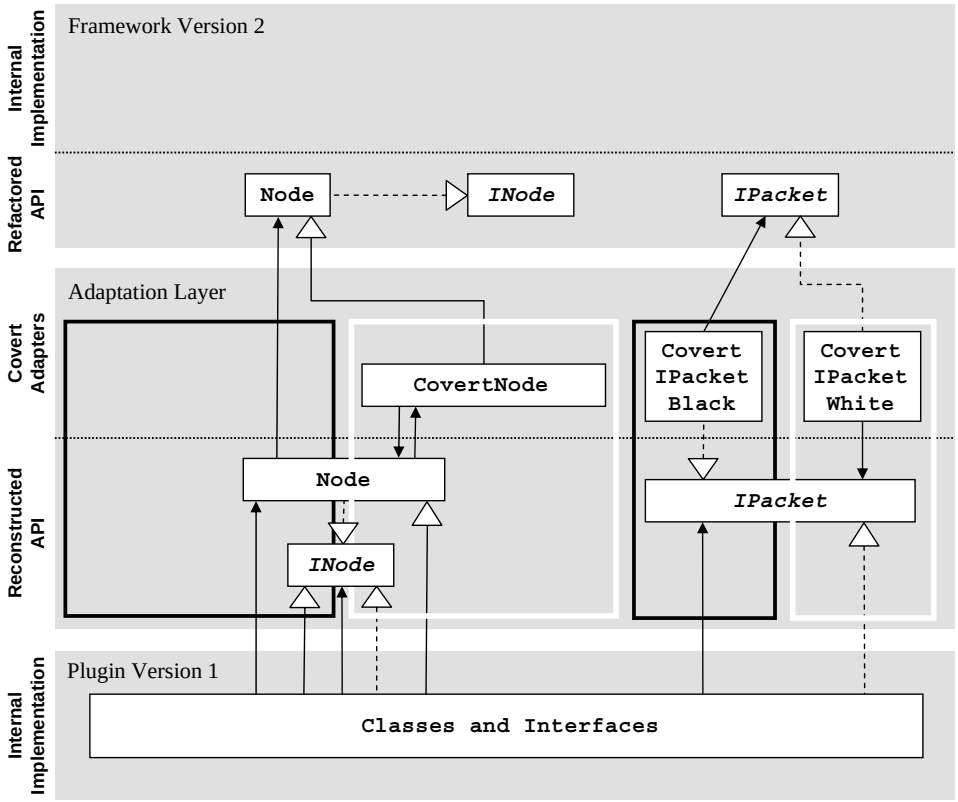



Figure 3.27: Sharing reconstructed API types in the static adaptation layer. Reconstructed API types *Node* and *IPacket* are shared by the applications of the corresponding black-box and white-box adapter patterns (sharing shown by the reconstructed API types crossing the patterns' boundaries). Covert adapter classes for *INode* are omitted.

Adapter pattern or the White-Box Class Adapter pattern) to apply. As a consequence, the constructor to be invoked should be either a black-box class adapter constructor (i.e., similar to Figure 3.25) or a white-box class adapter constructor (i.e., similar to Figure 3.26). In the rest of this section, we discuss in detail how we make this decision.

Considering a particular refactored API class (or interface), although conceptually the Black-Box and White-Box Class (or Interface) Adapter patterns are applied separately in the static adaptation layer (e.g., as shown in Figure 3.22 on page 69), in practice the applications of the two patterns overlap. Namely, because the both patterns reconstruct the very same API type (i.e., of an API class or an API interface before the refactoring), the patterns' applications share that reconstructed API type in the static adaptation layer (Figure 3.27). In practice, it means that the implementation of the reconstructed API type has to correspond to both (the black-box and the white-box) adapter patterns.

Regarding a reconstructed API interface (e.g., *IPacket* in Figure 3.27), its sharing by the corresponding applications of the Black-Box and White-Box Interface Adapter patterns has no implication on the interface implementation. Since a reconstructed API interface provides merely a set of method signatures (without method implementations), the interface is exactly the same in both pattern applications: it is either implemented (in the Black-Box Interface Adapter pattern) or composed (in the White-Box Interface Adapter pattern) by a corresponding covert adapter.

However, sharing a reconstructed API class (e.g., *Node* in Figure 3.27) requires that the class supports the adapter functionality (i.e., the implementations of adapter constructors and adapter methods, the types and instantiation logic of adaptee fields) specific for each of the two pattern applications. In practice, it means that the implementations corresponding to Reconstructed API Class of the black-box and white-box

class adapter patterns are merged in the same reconstructed API class. For example, the reconstructed API class *Node* of Figure 3.27 contains such functionality as black-box and white-box adapter constructors, forwarding methods of a black-box class adapter, and upcalling methods of a white-box class adapter.

When a constructor of the reconstructed API class is invoked from the plugin, we need to infer, whether for this particular constructor invocation the reconstructed API class in question participates in an application of either the Black-Box Class Adapter pattern or the White-Box Class Adapter pattern. Depending on the inferred information, we should execute the corresponding implementation of the invoked adapter constructor (i.e., either a black-box class adapter constructor as in Figure 3.25 or a white-box class adapter constructor as in Figure 3.26). Moreover, when an adapter method of the reconstructed API class is invoked at some later point in time, we should use that inferred information to decide, which (a black-box or a white-box) adapter method provided by that reconstructed API class to invoke.

To infer this information, we check (in Java, using introspection) the run-time type of the object, the creation of which in the plugin leads to invoking a constructor of the reconstructed API class in question.

1. In case the object's type is a reconstructed API class, the object being created is a framework wrapper. In other words, the plugin invokes the constructor of the reconstructed API class directly, as in black-box framework reuse. Therefore, for this particular constructor invocation the reconstructed API class participates in an application of the Black-Box Class Adapter pattern. As a consequence, we invoke the class' implementation of the corresponding black-box class adapter constructor (as shown for *PacketFactory* by Figure 3.25).

2. Otherwise, the object's type is a plugin class (subclassing a reconstructed API class) and the object being created is a plugin object. In other words, the plugin invokes the constructor of the reconstructed API class indirectly via a constructor call chain, as in white-box framework reuse. Therefore, for this particular constructor invocation the reconstructed API class participates in an application of the White-Box Class Adapter pattern. As a consequence, we invoke the class' implementation of the corresponding white-box class adapter constructor (as shown for *Node* by Figure 3.26).

Finally, the result of the type checking is saved, within the object being created, as a boolean value indicating the "color" of the object's class (*true* for "black" and *false* for "white"). The value is then used to decide for the implementations of the class' adapter methods to be invoked at run time.

### 3.2.3 Wrapping and Unwrapping Method Arguments and Return Values in Adapter Methods

The second case of an object crossing the domain boundaries between the framework and the plugin may occur, when an object of one domain is sent as a method argument to or return value of a method defined in the other domain. This occurs when the method caller (invoking the method) and the method callee (defining the method) are in different domains (the framework versus the plugin). For example, the plugin may send a plugin object to a framework API method and obtain a framework object as the return value of the method invocation. Since after the introduction of the run-time adaptation layer, its wrappers intercept all method calls from callers of one domain to callees of the other domain, such a method call will result in invoking an adapter method of a wrapper. The latter is then responsible for properly adapting method arguments obtained from the caller before sending them to the callee (i.e, its adaptee object), and the return value obtained from the callee before returning it to the caller.

To detect objects crossing domain boundaries, all objects passed as methods arguments to adapter methods and all objects obtained as return values after forwarding to methods of adaptees need to be investigated.

- **Method Arguments.** As method arguments the caller may send only objects it is aware of: objects of the caller's domain, the domain of adapters, or of library types. In the first case, the objects are wrapped to be sent to the callee. An example of the second case is a plugin sending as a method argument a framework wrapper obtained from a previously executed adapter method of another

wrapper. In such cases, the adaptee objects contained in the adapter objects are unwrapped, since they belong to the callee's object domain (in the example, to the framework domain). In the third case, that of library types, no adaptation is required, except for collection types (as we will discuss shortly in this section).

- **Return values.** Once all arguments are prepared (i.e., possibly wrapped and unwrapped), the adapter method uses them to forward the call to the callee (i.e., the wrapper's adaptee object). The return value obtained from the forwarding call is an object the callee is aware of: an object of the callee's domain, the domain of adapters, or of library types. In the first case, the objects are wrapped to be sent to the caller. An example of the second case is the framework returning a plugin wrapper previously obtained from a callback. In this case, the adaptee object is unwrapped to be returned to the caller, since the object belongs to the caller's object domain (in the example, to the plugin domain). Finally, return values of library types are treated similarly to method arguments.

The pseudocode in Listing 3.3 summarizes the (simplified) algorithm used for wrapping and unwrapping method arguments (lines 3–15) and return values (lines 35–45) in adapter methods. Since exception types also represent a (possible) return type, they are wrapped and unwrapped in a way similar to the other types (lines 21–32).[18] The helper functions used in the listing are:

- **isWrapper**(*Object*) checks, whether the object is a wrapper. If true, the function indicates that the wrapper's adaptee object returns to its domain of origin and needs to be unwrapped. The function uses the type metadata added to the adapters (as *annotations* in Java) at the time of adapter creation.
- **isUserDefined**(*Object*) checks, whether the object is an instance of a class defined in the refactored framework or in the existing plugin. The former is indicated by the object's class specifying one or more refactored API types, while the latter by the object's class specifying one or more reconstructed API types. If true, the function indicates that the object leaves the domain of its origin and must be wrapped.
- **isLibraryCollection**(*Object*) checks, whether the object is of a library collection type. Although itself not user-defined, a library collection (e.g., Java *List*) may contain objects of user-defined API types. When such collections are encountered in the adaptation layer, we iterate over their elements to find objects of user-defined API types (for wrapping) and adapter types (for unwrapping). Arrays are treated similarly to collections.
- **wrap**(*Object*) wraps the given object. The function calls the *getWrapper()* method of the adapter cache sending the object to be wrapped. If no wrapper for the object exists, the adapter cache will create a new framework or plugin wrapper; otherwise, an existing wrapper will be retrieved from the cache. In its implementation, **wrap** replaces the object to be wrapped with its wrapper on the call stack, before forwarding the call to the adaptee object (therefore, the method's "call-by-reference" notation used in Listing 3.3).
- **unwrap**(*Adapter*) obtains (either via a method call or via a direct field access) the adaptee object of the given wrapper. Similar to **wrap**, **unwrap** is implemented by replacing objects on the call stack (in this case, a wrapper by its adaptee object).

When wrapping a method argument (or a return value), it needs to be passed to a constructor of the adapter being instantiated, which stores the argument in the adapter's adaptee field. Therefore, each adapter needs to declare a constructor accepting an object to be wrapped (in our implementation, with the formal parameter type *Object*). However, the signature of such constructor could conflict with the signature of an adapter constructor defined for wrapping newly created instances (as discussed in the previous Section 3.2.2), in case the signature of the latter contains a single formal parameter of the very same type (i.e., *Object*). To distinguish the constructor dedicated for argument wrapping from other adapter constructors, we use in its signature a *surrogate* parameter of formal type unique in the adaptation layer. In our design, such unique

[18] The algorithm's code snippet coping with exception chaining as possible in Java is provided in Appendix B.1.

Listing 3.3: Wrapping and unwrapping method arguments and return values in adapter methods

```
1  <AccessModifier> <ReturnType> methodName(arguments) {
2    //wrap or unwrap method arguments
3    foreach (argument in arguments) {
4      if (argument != null) {
5        if (isWrapper(argument)) {
6          unwrap(argument);
7        } else if (isUserDefined(argument)) {
8          wrap(argument);
9        } else if (isLibraryCollection(argument)) {
10         foreach (element in argument) {
11           //wrap or unwrap as for ordinary objects
12         }
13       }
14     }
15   }
16
17   //forward to adaptee via reflection (details omitted)
18   Object value;
19   try {
20     value = adaptee.methodName(arguments);
21   } catch(Throwable t) {
22     //wrap or unwrap the exception object
23     if (isUserDefined(t)) {
24       wrap(t);
25       throw t;
26     } else if (isWrapper(t)) {
27       unwrap(t);
28       throw t;
29     } else {
30       throw t;
31     }
32   }
33
34   //wrap or unwrap return value
35   if (value != null) {
36     if (isWrapper(value)) {
37       unwrap(value);
38     } else if (isUserDefined(value)) {
39       wrap(value);
40     } else if (isLibraryCollection(argument)) {
41       foreach (element in argument){
42         //wrap or unwrap as for ordinary objects
43       }
44     }
45   }
46   return value;
47 }
```

type is the singleton class *AdapterCache* (i.e., the class implementing the adapter cache). Since none of the adapter constructors dedicated for wrapping newly created instances can have a formal parameter of type *AdapterCache*, we effectively disambiguate them from *surrogate constructors* for wrapping method arguments and return values. Figure 3.28 shows the surrogate constructor of the *PacketFactory* adapter, the other adapter constructors of which were introduced by Figure 3.25 on page 72.[19]

Finally, to properly identify an object's domain of origin in the **isUserDefined** function, we must consider

[19] As a technical remark, we do not need to implement argument wrapping and, therefore, surrogate constructors in white-box class adapters. This is a consequence of our decision to cache instances of Covert Plugin Adapters of the White-Box Class Adapter pattern as plugin wrappers at the time of plugin object creation (as discussed for Figure 3.26 on page 73). For these plugin objects, their wrappers are always present in the adapter cache and can be reused in adapter methods. As a consequence, surrogate constructors are declared only in black-box class adapters, and (as the only constructors, and consistently with black-box class adapters) in black-box and white-box interface adapters.

Reconstructed API Class

```
public class PacketFactory {
 refactoredAPI.PacketFactory adaptee;

 //surrogate constructor
 public PacketFactory(AdapterCache surrogateParameter, Object toWrap) {
   adaptee = (refactoredAPI.PacketFactory) toWrap;
   AdapterCache.putWrapper(adaptee, this);
 }
}
```

Figure 3.28: Surrogate constructor for wrapping of method arguments and return values. Since we do not need the value of the surrogate parameter, when invoking a surrogate constructor we can pass *null* as its first argument.

**Terminology Note**

A **method parameter** is a name associated with a type, called the **formal parameter type**. For an object to be passed as a method argument, the object's type must be conformant to the formal type of the corresponding method parameter. In statically typed languages, such as Java, the type conformance can be verified by a type checker during compilation [Mey00, p. 467-472]. The ability of passing the objects of various types conformant to the type expected by method callers lies at heart of **polymorphism**—the powerful ability to vary method implementations depending on the particular type of the message receiver.

the actual type of a method argument in question, and not the formal type of the corresponding method parameter. In the presence of type polymorphism, as in Java, the actual type of a method argument may be more specific than the formal parameter type declared in the method signature. For example, an object of the framework class *Packet* may be passed as an argument, where the formal parameter type is declared to be of *Object*. To correctly identify the object's domain of origin, instead of considering the formal parameter type (e.g., *Object*), we should check for the class of the object (e.g., *Packet*) passed as an argument to an adapter method. The same applies for the types of the return values obtained from the forwarding calls to the adaptees. Besides identifying objects' domain of origin, this type checking is also crucial for proper adapter creation (i.e., creating an adapter class) and instantiation (i.e., creating a wrapper), as we discuss in the next section.

### 3.2.4 Proper Adapter Creation and Instantiation

If a method argument needs to be wrapped, considering the actual argument type is crucial to decide for the correct adapter to be instantiated. The same applies for method return values. Let us consider the following example of a black-box reuse, with three API types excerpted from RefactoringLAN: an API interface *INode* implemented by an API class *Node*, subclassed in turn by an API class *Server*. The API methods of *INode* are a subset of the API methods of *Node*, the methods of which in turn are a subset of the API methods of *Server*. At application execution time, the framework creates a *Server* object and returns it to the plugin from three distinct API methods as the value of a formal return type (1) *INode* of the first API method, (2) *Node* of the second API method, and (3) *Server* of the third API method. The framework is thereafter refactored and upgraded.

In exhaustive API adaptation, to the three, possibly refactored, API types (the names of which we prefix with *refactoredAPI.* in the rest of this section) we have the corresponding adapters in the static adaptation layer: *CovertINodeBlack* implementing reconstructed API *INode* (for *refactoredAPI.INode*), *Node* (for *refactoredAPI.Node*) and *Server* (for *refactoredAPI.Server*). Moreover, to the three aforementioned API methods correspond three adapter methods in the reconstructed API, in which the object returned by the framework should be wrapped to be sent to the plugin. This wrapping is done in one of the three adapter

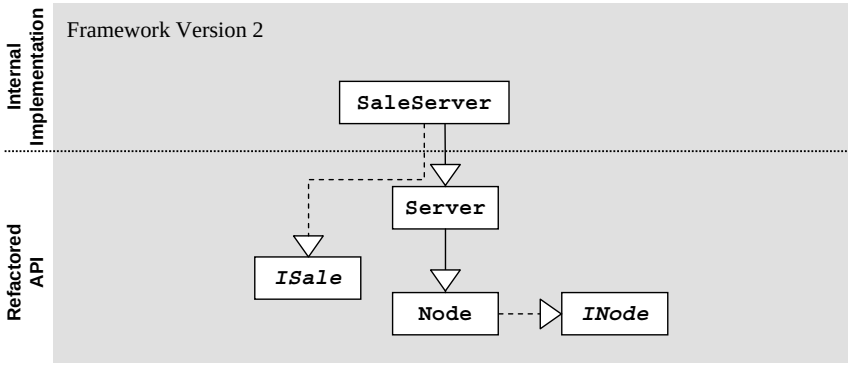



Figure 3.29: Unrelated types in the refactored API. Whereas *SaleServer* is a subtype of both *Server* and *ISale*, the latter two are not related in the API by subtyping.

methods, namely, in the one, in which the framework object is encountered for the first time in the run-time adaptation layer. Thereafter, the framework wrapper should be cached and safely reused in the other two adapter methods.

However, in case the first adapter method should perform wrapping, by considering the formal return type *refactoredAPI.INode* of the first API method and instantiating the interface adapter *CovertINodeBlack*, we cannot safely reuse the created framework wrapper in the other two adapter methods. Namely, the wrapper is unaware of the methods defined in *refactoredAPI.Node* and *refactoredAPI.Server* and not present in *refactoredAPI.INode*. Similarly, if we instantiate the *Node* class adapter for the *refactoredAPI.Node* return value of the second method, we cannot safely reuse the wrapper in the third adapter method, due to server-specific methods defined in *refactoredAPI.Server*.

Instead, when we encounter an object instantiating *refactoredAPI.Server* in the run-time adaptation layer for the first time, we want to instantiate the class adapter for *refactoredAPI.Server* (i.e., the adapter class *Server*). For that, we should consider the actual run-time type of the object (i.e, the object's class *refactoredAPI.Server*), and not the formal type of the method return value. An instance of this adapter can then be reused safely, whenever the object to be wrapped is passed as *refactoredAPI.INode*, *refactoredAPI.Node*, or *refactoredAPI.Server*. For an object to be wrapped, we call such a corresponding adapter the object's *specialized adapter*.

In our example, since *refactoredAPI.Server* is an API class, its class adapter has been generated together with the reconstructed API and can be instantiated whenever needed. However, an object's class can be an internal framework class not present in the API, say *SaleServer* extending *refactoredAPI.Server* (Figure 3.29).[20] Since *SaleServer* is not in the API, no adapter was created for it when creating the static adaptation layer. Can we consider *Server* adapter to serve also as the specialized adapter for objects of *SaleServer*? We can, unless *SaleServer* implements an API interface unrelated in the API to *refactoredAPI.Server* by subtyping. For example, in addition to subclassing *refactoredAPI.Server*, *SaleServer* could implement an API interface *refactoredAPI.ISale* (Figure 3.29). In this case, using *Server* adapter cannot guarantee its objects will answer messages of *refactoredAPI.ISale*.

In general, when wrapping an object, we might not find its specialized adapter in the static adaptation layer. In this case, we need to create the object's specialized adapter at run time and then instantiate it to wrap the object in question. We call such adapters *dynamically created*. In our example, we will create a new adapter that is a subtype of the adapters for *refactoredAPI.Server* and *refactoredAPI.ISale*. Technically, we cannot subclass both existing adapter classes, because Java does not support multiple (class) inheritance. Therefore, our dynamically created adapter, say, *ServerISale* will subclass the reconstructed API class *Server* and implement the reconstructed API interface *ISale*.

[20]Since the three figures of this section—Figure 3.29, Figure 3.30, and Figure 3.31—are not referenced in other sections of the thesis, they are not included in RefactoringLAN to avoid unnecessarily complicating it.

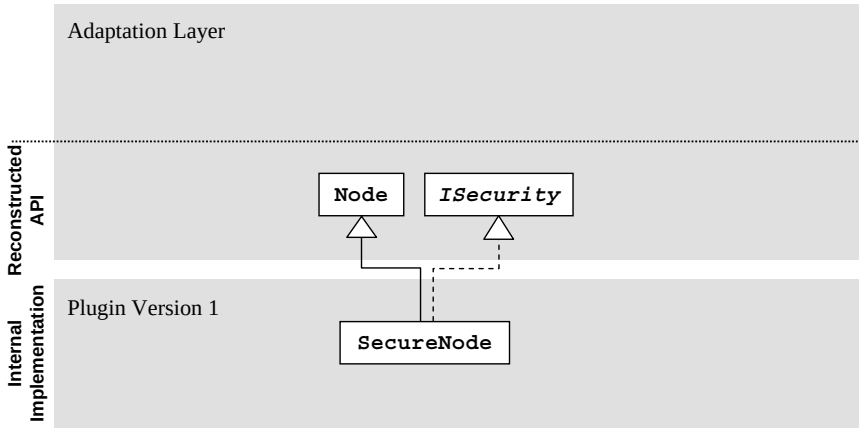



Figure 3.30: Unrelated types in the reconstructed API. The types *Node* and *ISecurity*, subtyped by the plugin's *SecureNode*, have no subtyping relation in the reconstructed API.

The dynamic adapter creation is required only when for an object to be wrapped (1) no wrapper exists, and (2) no specialized adapter is found in the adapter cache. For instance, if we consider yet another internal framework class *BikeSaleServer* subclassing *SaleServer* and implementing no additional API interface, we can safely reuse the aforementioned dynamically created adapter *ServerISale* to adapt the objects of *BikeSaleServer* (i.e., *ServerISale* is the specialized adapter of the objects of *BikeSaleServer*).

While for wrapping framework objects (as in the previous example) we consider the type relations in the refactored API, for wrapping plugin objects we need to consider the type relations in the reconstructed API. Let us reconsider our excerpt from RefactoringLAN, where a plugin class *SecureNode* subclasses the framework's *Node*. For the two classes, the application of the White-Box Class Adapter pattern (shown by Figure 3.6, page 49) results in the reconstructed API class *Node* (Reconstructed API Class) and the adapter class *CovertNode* (Covert Plugin Adapter). However, the latter can be considered the specialized adapter for the objects of *SecureNode* only if *SecureNode* does not implement any (reconstructed) API interfaces unrelated to the reconstructed *Node* (e.g., *ISecurity*, as shown in Figure 3.30). Otherwise, the first time we need to wrap an object of *SecureNode*, we have to dynamically create the specialized adapter for *SecureNode* by subclassing the adapter *CovertNode* and implementing the reconstructed API interface *ISecurity*. The resulting adapter can then be instantiated to wrap the objects of *SecureNode*.

The Java-like pseudocode of Listing 3.4 reveals the internals of the adapter cache responsible for proper adapter creation and instantiation. The method *getWrapper()* (line 14) is called from the function **wrap** of an adapter method, whenever an object encountered in the adapter method must be wrapped. If the method does not find a cached wrapper for the given object (lines 16–17), it will call the cache method *lookupSpecializedAdapter()* (line 20), passing the class of the object to be wrapped. The *lookupSpecializedAdapter()* method will then query the *adaptees2adapters* map to see, whether for the given class there is a corresponding adapter generated in the static adaptation layer (line 30). If not, the adapter supporting the given class, and all API interfaces implemented by the class and its ancestors, will be created dynamically (lines 31–60). Finally, the existing or newly created adapter is instantiated by passing the object to be wrapped to the adapter's surrogate constructor (lines 22–25). To guarantee proper adapter creation and instantiation in multi-threaded programs, the methods of Listing 3.4 are implemented as synchronized methods (omitted from the listing).

An alert reader may have noticed that for a given object, the algorithm implemented in the *lookupSpecializedAdapter()* method might not find any API superclass of the object's class (the comparison in line 52 of Listing 3.4 evaluating to *true*). This occurs when a (framework or plugin) class implements one or more API interfaces without subclassing any API class (as shown for a plugin class in Figure 3.31). In this particular case, the dynamically created adapter will implement all corresponding (refactored or reconstructed) API interfaces without subclassing any adapter class. In fact, this is the only case when we create and instantiate an interface adapter, and not a class adapter.

Listing 3.4: Adapter cache (extended)

```
public class AdapterCache{
  //adaptees and their adapters
  private Map adaptees2adapters;

  //wrapped objects and their wrappers
  private Map wrapped2wrappers;

  //cache wrapped objects and their wrappers
  public void putWrapper(Object wrapped, Object wrapper){
    wrapped2wrappers.put(wrapped, wrapper);
  }

  //given an object to be wrapped, return the corresponding wrapper
  public Object getWrapper(Object toWrap){
     //reuse existing wrapper for this object, if any
    if (wrapped2wrappers.containsKey(toWrap))
      return wrapped2wrappers.get(toWrap);

    //otherwise create new wrapper
    Class adapter = lookupSpecializedAdapter(toWrap.getClass());

    //get and invoke surrogate constructor that will update adapter cache
    Constructor constr = adapter.getConstructor(AdapterCache.class, Object.class);
    Object wrapper = constr.newInstance(null, toWrap);
    return wrapper;
  }

  //given the class of adaptee, return the corresponding most specific adapter
  public Class lookupSpecializedAdapter(Class target) {
    Class adapter = adaptees2adapters.get(target);
    if (adapter == null) {
      //target is not an API class, investigate it
      List interfaces = new ArrayList();

      //collect implemented API interfaces
      do {
        Class[] implemented = target.getInterfaces();
        for (int i = 0; i < implemented.length; i++) {
          Class temp = adaptees2adapters.get(implemented[i]);
          if (temp != null) {//true for API interfaces, false for other interfaces
            interfaces.add(temp);
          }
        }

        //move up the inheritance hierarchy
        target = target.getSuperclass();
        adapter = adaptees2adapters.get(target);
      } while (target != Object.class && adapter == null);

      //generate (and serialize) new adapter (details omitted)
      if (interfaces.size() > 0) {
       if(target == Object.class)
         //dynamically create interface adapter
       else
         //dynamically create class adapter
       //store adaptee/adapter pair
       adaptees2adapters.put(target,adapter);
      }
    }
    return adapter;
  }
}
```

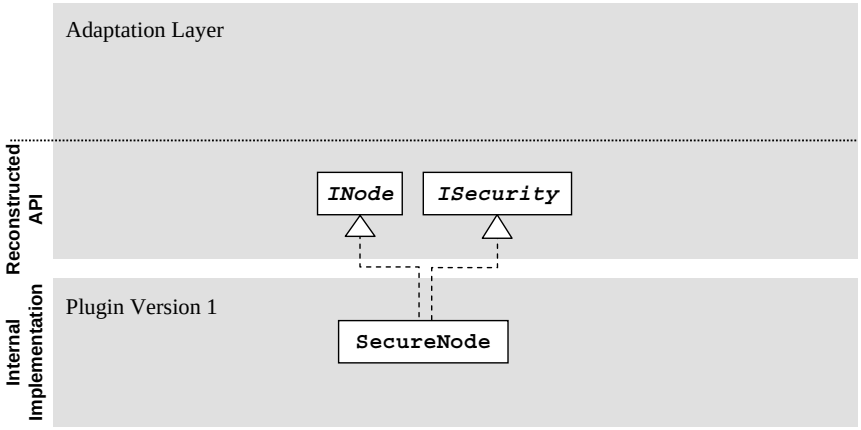



Figure 3.31: No API superclass of the object's class. The plugin class *SecureNode* implements two API interfaces without subclassing any API class.

Finally, to avoid repeated dynamic creation of the same adapters (e.g., when the application is stopped and run again), we persist the dynamically created adapters onto a storage medium for later reuse. As a consequence, over time the size of the static adaptation layer increases, while the need for dynamic adapter creation decreases.

### 3.2.5 Example of Exhaustive API Adaptation: Framework and Plugin Wrappers in Concert

Let us look at a realistic scenario of exhaustive API adaptation, reconsidering examples of framework reuse introduced in Section 2.2.2, API refactorings shown in Section 2.2.3, and adaptation pattern applications discussed in Section 3.1, where all examples are excerpted from RefactoringLAN (Appendix A). In the first framework version, for modeling network nodes and packets the framework API offers the classes *Node* and *PacketFactory*, and an interface *IPacket*. The *PacketFactory* class defines a factory method *createPacket()* that, given a node and a node identifier, returns a packet of type *IPacket*. In addition, given a node and a packet in question, *PacketFactory*'s method *packetSeenInNode()* reports (as an instance of an API class *Report*) the trace of the packet as seen in the node (Figure 3.32, top part).

In the plugin developed with the first framework version, a plugin class *SecureNode* subclasses the framework's *Node*. At run time (Figure 3.32, bottom part), the plugin creates an object of *SecureNode*, then instantiates *PacketFactory* and obtains a packet from the *PacketFactory*'s *createPacket()* method. Later in the execution, the plugin calls *PacketFactory*'s *packetSeenInNode()* method to obtain a report about the occurrences of the packet in the node. During the communication of the framework and plugin, objects of both domains (denoted in Figure 3.32 by two different kinds of circles, one for each domain) cross domain boundaries.

In the second framework release, the API is refactored. Among API refactorings, the `RemoveParameter` refactoring applied to the *PacketFactory* method *createPacket()* removes the second method parameter. Moreover, the `RenameMethod` refactoring changes the name of the *PacketFactory*'s *packetSeenInNode()* method to *tracePacket()* (Figure 3.33, top part). The refactorings of other API types (e.g., *Node* and *IPacket*) are omitted in the rest of this section for simplicity.

Applying exhaustive API adaptation results in the static adaptation layer completely reconstructing the API of the first framework version. Among its adapters, the static adaptation layer contains black-box and white-box adapters for possibly refactored API types *PacketFactory*, *IPacket*, *Node*, and *Report*. If now the old plugin code is executed (Figure 3.33, bottom part), all API constructor invocations in the plugin will result in invoking adapter constructors of the reconstructed API, and creating wrappers for newly created plugin and framework instances (as discussed in Section 3.2.2). In particular, the constructor invocation of

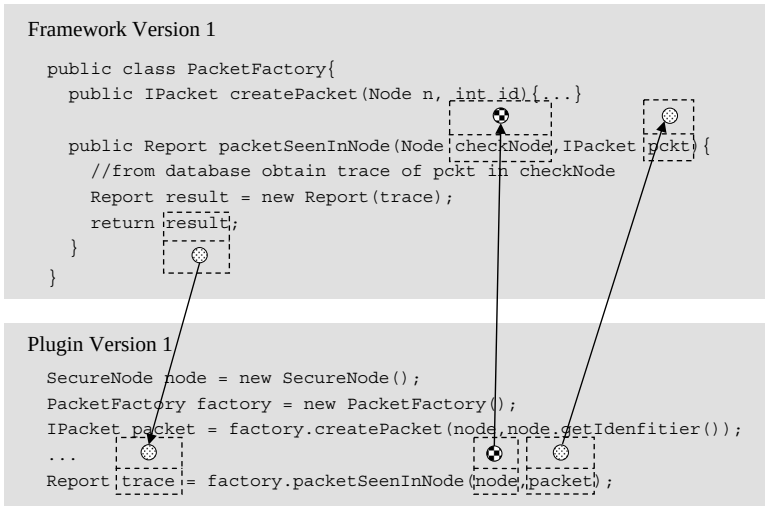



Figure 3.32: Object exchange between the framework and the plugin of the same version. For an object, its textual (variable name) and graphical (circle) representations are combined in a dashed box. Arrows show the object exchange when invoking the *packetSeenInNode()* API method in the plugin.

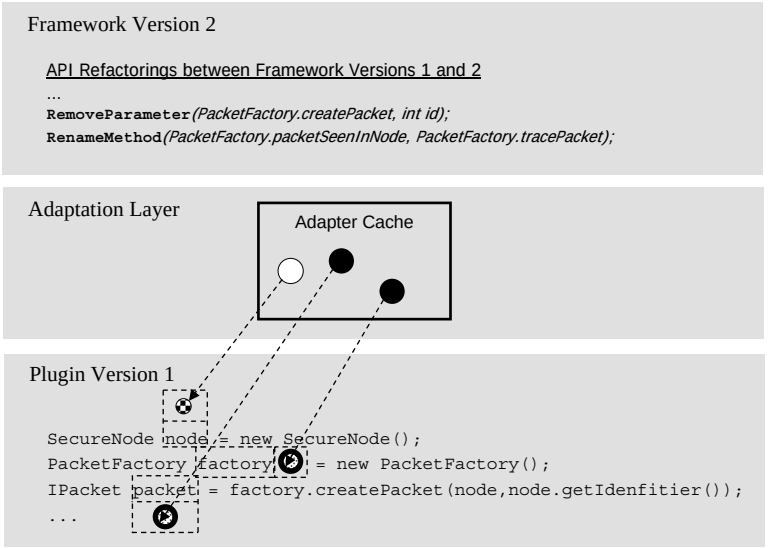



Figure 3.33: Object exchange and the adapter cache after the framework upgrade and adaptation. Cached wrappers are shown as white circles (plugin wrappers) and black circles (framework wrappers). Arrows from the wrappers in the cache point to their adaptee objects, which the wrappers could possibly wrap (the plugin object *node*) or are already wrapping (the framework objects *factory* and *packet*).

*SecureNode* creates and caches a plugin wrapper—an instance of *CovertNode* adapter, while the constructor invocation of *PacketFactory* creates and caches a framework wrapper—an instance of *PacketFactory* adapter. All created wrappers are cached in the adapter cache of the adaptation layer (Figure 3.33, middle part).

As a consequence of wrapping a newly created instance of the framework's *PacketFactory*, the *factory* object created in the plugin is a framework wrapper. Its adapter method *createPacket()* has the same signature as before the `RemoveParameter` refactoring was applied to the *PacketFactory*'s *createPacket()*. As any adapter method, the wrapper's *createPacket()* performs wrapping and unwrapping of method arguments and return values. In particular, the method wraps an obtained framework packet into another framework wrapper (*packet* in Figure 3.33), which is cached and returned to the plugin.

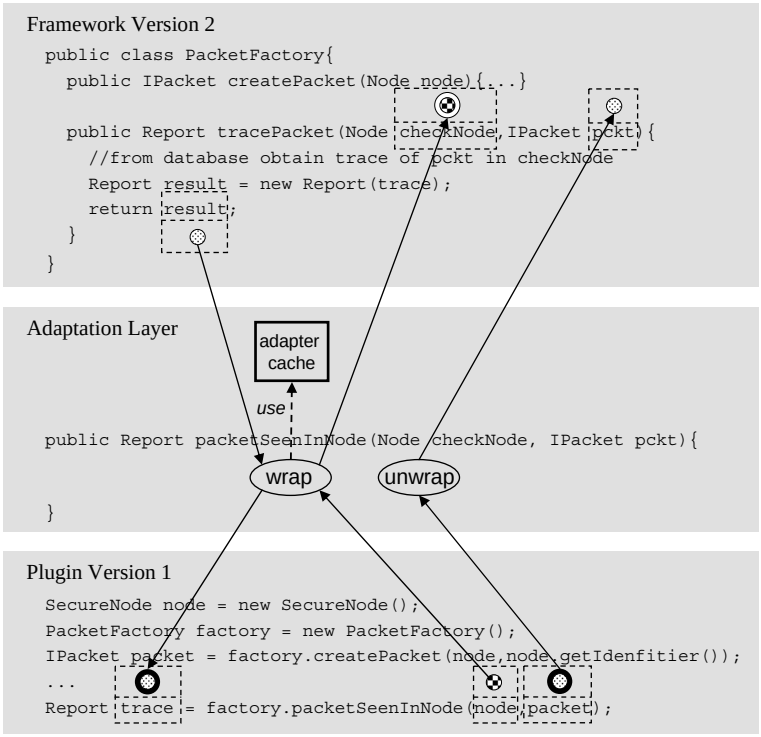



Figure 3.34: Object wrapping and unwrapping in an adapter method. Using the adapter cache, the *packetSeenInNode()* adapter method wraps its first argument (a plugin object) and unwraps its second argument (a wrapped framework object) before forwarding the call to the framework. Similarly, the method wraps the obtained return value (a framework object) before returning it to the plugin.

Let us now consider the object wrapping and unwrapping performed upon the invocation of the old API method *packetSeenInNode()*, called *tracePacket()* in the refactored API (Figure 3.34). As mentioned in the previous paragraph, now the *factory* object, on which the method *packetSeenInNode()* is invoked in the plugin, is a framework wrapper. Therefore, the *packetSeenInNode()* method is an adapter method forwarding to the *tracePacket()* of the refactored API's *PacketFactory*. Since the first method argument of the adapter method being invoked is a plugin object *node* (of type *SecureNode*), the object is wrapped. Since the second method argument is a framework wrapper *packet*, its adaptee object is unwrapped. Then both objects obtained (the plugin wrapper and the framework object) are used as actual method arguments to invoke the framework's *tracePacket()*. Since the method invocation returns a framework object (of type *Report*), the latter is wrapped in a framework wrapper that is returned from the adapter method to the plugin. If that framework wrapper does not exist yet, the adapter for *Report* is instantiated in the adapter method; otherwise, the wrapper is retrieved from the adapter cache.

## 3.3 Description of Challenges, Adopted Solutions, and Limitations in Using Adapters for Refactored Framework APIs

While realizing our adapters for particular API types and exhaustive API adaptation for the API as a whole, we encountered a number of conceptual and technical problems due to the introduction of adapters as entities different from their adaptees, the programming style of framework and plugin developers, the performance overhead implied by adapters, and particularities of the Adapter design pattern. In this section, we shortly overview the most interesting problems met, discuss solutions adopted and, in some cases, limitations implied. In those cases, where our solutions are based on the adaptation decisions presented in previous

sections, we shortly restate the main issues involved.

### 3.3.1 Solving Object Schizophrenia in the Presence of Adapters

Object composition as a reuse mechanism used in the object version of the Adapter design pattern introduces *object schizophrenia* [SR02b]: what is intended to appear as a single object is actually broken up into several, each possibly possessing its own identity, state, and behavior. This may lead to various problems, when an object is unable to properly respond to messages, although the necessary information is available in the object. In our case, such problems could have appeared due to *split identity*, *split state*, and *split behavior*.

#### Solving Split Identity

In philosophy, the identity is whatever makes an entity identifiable among other entities of the same kind. In object-oriented programming languages, such as Java, there are two notions strongly related to the general notion of identity: object equivalence and object identity.

Two objects are considered equal, if their values are equal (under a given equivalence relation). In most of the current object-oriented programming languages, root language types expose predefined methods (such as *equals()* in Java) expressing object equivalence. These methods are inherited and can be redefined in subclasses. Checking for object equivalence is implemented by invoking these methods on the objects to be compared.

Two objects are considered identical, if they are represented by the same (bits of) memory. In practice, since objects in the memory are usually accessed through references (e.g., of program variables), checking for object identity is implemented by comparing object references.

Both object equivalence and object identity are of a particular importance in those applications, where the behavior depends on the result of object comparison (for equivalence or identity). Since wrappers are instances of adapters different from their adaptees, and, furthermore, are objects physically different from the adaptee objects, introducing adapters may split the identity of the objects being adapted between the wrapped objects and their wrappers. Therefore, we ought to identify and address all cases, when previously correct object comparison may malfunction after adaptation.

**Preserving object equality.** When an adaptee object is wrapped, its methods for comparing object equivalence might not be accessible for existing callers that expect the semantics of object equivalence as defined before adaptation.

*Solution.* Our solution is straightforward: besides other adapter methods, the wrappers also forward the aforementioned predefined methods to the adaptee objects thereby preserving expected object equivalence. By calling the realization of object equivalence of their adaptees, the adapters also share their adaptees' values used for object comparison.

**Preserving object identity.** Reference comparison relies on objects' physical location: only should the two object references being compared point to the same physical location, they are considered to refer to the same object; otherwise, they are considered to refer to two different objects. Since a wrapped adaptee object and its wrapper are physically different objects, they reside in different memory locations. We identify and address two cases where comparing object references in the presence of adapters could lead to erroneous results.

1. Comparing an adaptee object and its wrapper. Assume the framework creates an object of one of its API types and saves that object in a variable before sending the object to the plugin. Since the framework object crosses domain boundaries, it is wrapped in the run-time adaptation layer. If later the framework wrapper is passed back from the plugin to the framework, the result of comparing it with the object saved in the framework variable would be erroneous, since the objects reside in different memory locations.

   *Solution.* Our solution to this problem is implied by unwrapping objects in the adaptation layer (as discussed in Section 3.2.3), making it impossible to compare a wrapper and its adaptee object.

Refactored API Classes

```
public class refactoredAPI.Packet {
 private Message msg;

 public void setMessage(String msg) {
  this.msg = new Message(msg);
 }

 public void setTimestamp() {
  msg.setTimestamp(getLocalTime());
 }
}

public class refactoredAPI.EncryptedPacket
                extends refactoredAPI.Packet {

 public void setMessage(String msg) {
  super.setMessage(encrypt(msg));
 }

}
```

Reconstructed API Classes

```
public class Packet {
 private refactoredAPI.Packet adaptee;

 public void setMessage(String msg) {
  adaptee.setMessage(msg);
 }

 public void setTimestamp() {
  adaptee.setTimestamp();
 }
}

public class EncryptedPacket
                extends Packet {

 private refactoredAPI.EncryptedPacket adaptee;

 public void setMessage(String msg) {
  adaptee.setMessage(msg);
 }
```

Figure 3.35: Split state in black-box class adapters. The left part of the figure shows two refactored API classes related by inheritance, and the right part lists the corresponding black-box class adapters. When *setTimestamp()* is called on an instance of the adapter *EncryptedPacket*, the adapter method inherited from *Packet* is called that will inadvertently set the timestamp on the message managed through the adaptee field of *Packet* adapter.

2. Comparing wrappers of the same adaptee object. As discussed in Section 3.2.3, the same object may be repeatedly sent from the framework to the plugin or from the plugin to the framework. If we created a new wrapper each time the same object crossed domain boundaries, we would end up with different wrappers for the same adaptee object. If the references of those wrappers were compared, since the wrappers were physically different, they would fallaciously appear to wrap different adaptee objects.

   *Solution.* By reusing previously created wrappers stored in the adapter cache (Section 3.2.1) and, furthermore, instantiating objects' specialized adapters (Section 3.2.4), we ensure that each object crossing domain boundaries is associated with exactly one wrapper.

**Solving Split State**

Besides sharing the common behavior, class inheritance implies also sharing states. For example, in the left part of Figure 3.35 the class *EncryptedPacket* subclassing the class *Packet* also shares *Packet*'s message *msg*. Although the message is stored in a private field of *Packet*, *EncryptedPacket* can access and modify the field's value via the redefined method *setMessage()* and the inherited method *setTimestamp()*. As a consequence, method calls in the plugin's *LAN* invoking *setMessage()* on an instance of *EncryptedPacket* will store an encrypted message in the *msg* private field, while invoking *setTimestamp()* on that instance will modify the timestamp of the stored message.

Assuming *Packet* and *EncryptedPacket* are present in the refactored API, their corresponding black-box class adapters are listed in the right part of Figure 3.35. Plugin calls in *LAN* previously invoking the API methods *setMessage()* and *setTimestamp()* on an instance of the framework's *EncryptedPacket* are now invoking the corresponding adapter methods of a framework wrapper—an instance of the adapter *EncryptedPacket*. However, the calls will result in a different behavior than before the adaptation. Whereas the call to the wrapper's *setMessage()* will store an encrypted message in the wrapper's adaptee object, the call to the wrapper's *setTimestamp()* will modify the timestamp of the message of the adaptee object stored in the adaptee field of *Packet*. This may result in an unexpected behavior, such as a null pointer exception (e.g., in case, the message of the adaptee object in *Packet* is not initialized), and a wrong timestamp of the message stored in the adaptee object of *EncryptedPacket* (e.g., in case, the timestamp is later inquired during network

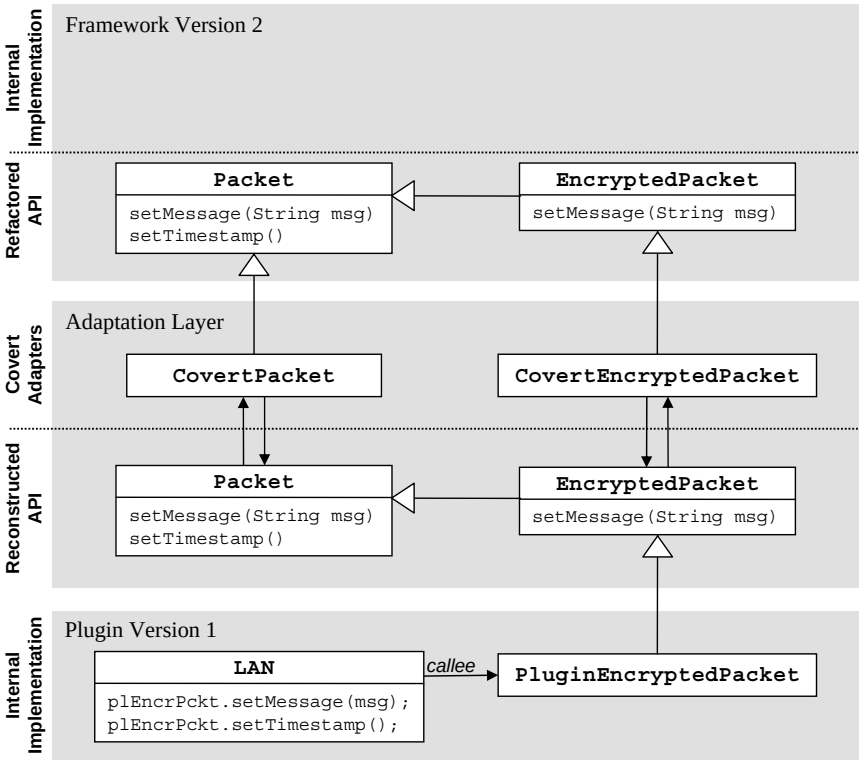



Figure 3.36: Split state in white-box class adapters. Invoking the inherited method *setMessage()* on an instance of *PluginEncryptedPacket* will modify the adaptee object as stored in *CovertEncryptedPacket*, whereas invoking *setTimestamp()* on the very same instance will affect the state of the adaptee object of *CovertPacket*.

inspection). The reason is that the state of the wrapper is effectively split into the states of the two adaptee objects (stored in *Packet* and *EncryptedPacket*, respectively).

For a similar problem appearing in the presence of white-box class adapters, let us consider the very same API classes *Packet* and *EncryptedPacket*, the latter subclassed in the plugin by *PluginEncryptedPacket*. As a consequence, *PluginEncryptedPacket* inherits the methods *setTimestamp()* (as defined in the framework's *Packet*) and *setMessage()* (as redefined in the framework's *EncryptedPacket*). For this framework reuse scenario, Figure 3.36, in which we use a graphical notation instead of code for the sake of compact presentation, shows the corresponding classes resulting from the application of the White-Box Class Adapter pattern. If now on an instance of *PluginEncryptedPacket* the plugin will invoke the methods *setMessage()* and *setTimestamp()*, the method invocations may result in an unexpected behavior, similar to the example of the black-box class adapters from the previous paragraph.

In general, the problem of the split state is inherent to using message forwarding in the object version of the Adapter pattern (and, hence, our variants of the pattern). In case a locally defined method of an adapter should be called (e.g., *setMessage()* of *EncryptedPacket*), message forwarding will bind the self reference in the adapter method to the locally stored adaptee object. By contrast, in case a method inherited in the adapter (from one of its adapter ancestors in the reconstructed API class hierarchy) is to be called (e.g., *setTimestamp()* of *Packet*), message forwarding will bind the self reference to the adaptee object of the ancestor. If the method execution changes the state of the adaptee object (as in our examples of Figure 3.35 and Figure 3.36), the method execution will fallaciously change the state of the wrapper expressed in the states of different adaptee objects.

*Solution.* To avoid split states in the presence of adapters we must make sure that in all its adapter methods a wrapper always uses the same adaptee object. In other words, we must ensure that the self reference in all

Reconstructed API Classes

```
public class Packet {
 protected refactoredAPI.Packet adaptee;

 public void setMessage(String msg) {
   adaptee.setMessage(msg);
 }

 public void setTimestamp() {
   adaptee.setTimestamp();
 }
}

public class EncryptedPacket
                  extends Packet {

 public void setMessage(String msg) {
  ((EncryptedPacket) adaptee).setMessage(msg);
 }
```

Figure 3.37: Solving split state in black-box class adapters. The adaptee field is declared protected, and inherited by the descendants of the top-most reconstructed API class. To reflect the adaptee type specialization, the adapter implementations down the type hierarchy cast the adaptee field to the type of the adaptee used from a particular adapter. While not necessary in this example, type casting is required when a more specific adaptee introduces methods not present in the adaptee used in the top-most reconstructed API class.

adapter methods of a wrapper will always be bound to the same adaptee object. Technically, we considered two implementations of this solution, both performed as the final step of generating the static adaptation layer.

1. Redefining inherited adapter methods. For every inherited adapter method in a reconstructed API class, we create a locally defined adapter method, which forwards to the adaptee object stored in the private adaptee field of that reconstructed API class. In both our examples, this would mean adding the adapter method *setMessage()* to the reconstructed *EncryptedPackage* with a method implementation forwarding to the adaptee object stored in the private adaptee field of *EncryptedPackage*.

2. Inheriting adaptee fields. By making the adaptee field protected (instead of private) we permit reconstructed API classes sharing adaptee objects. Only the top-most reconstructed API class in the hierarchy defines the adaptee field, which is inherited by the reconstructed API classes down the hierarchy. In our examples, this would mean declaring the adaptee field protected in *Packet* (assuming it is the top-most adapter) and inheriting the field in *EncryptedPacket* (as exemplified for black-box class adapters in Figure 3.37). Since an inherited adaptee field must hold instances of any possible adaptees starting from the one used by the top-most reconstructed API class, the field's type conceptually is the least specific adaptee type (i.e., the one used by the top-most adapter).

Interestingly, both implementations can be realized as refactorings. In the former case, refactorings add forwarding adapter methods to reconstructed API subclasses that are semantically equivalent to adapter methods of reconstructed API superclasses. In the latter case, refactorings pull up adaptee fields in the hierarchy of reconstructed API classes. In practice, both solutions turned out to be equivalent in power. We decided to adopt the second solution, because it implied less members in the adapter implementation (which is relevant for the speed of class loading) and because we found reusing of adaptee fields somewhat more intuitive from the perspective of object-oriented programmers. Since the implementation of this solution belongs to the time of generating the static adaptation layer, its formal description is presented as part of the adapter generation in Section 4.3.3.

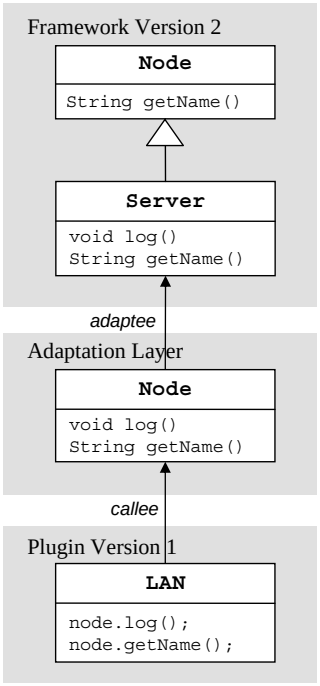



Figure 3.38: Split behavior. Whereas *LAN* expects to invoke the *getName()* method of *Node*, the message forwarding will invoke the *Server*'s *getName()*.

### Solving Split Behavior

Split behavior may appear in the presence of inheritance of classes used as adaptees, when message forwarding from an adapter may inadvertently call a method of its adaptee, while another adaptee method with possibly different semantics is expected.

For an example, let us assume that in the first framework release a concrete API class *Node* offers a method *log()* for logging the node content to a database. In the second framework release, framework developers realize that the method should be applicable only to certain nodes, namely, servers. Therefore, they apply the `PushDownMethod` refactoring resulting in the method *log()* pushed down from *Node* to its subclass *Server*. In addition, framework developers override *Node*'s *getName()* method in *Server*, introducing a slightly different method semantics.[21]

In the generated static adaptation layer, to localize the method *log()* the reconstructed API class *Node* uses *Server* as its adaptee (Figure 3.38). Calling *log()* in the plugin on a wrapper instantiating the adapter *Node* will execute to the intended behavior. However, calling *getName()* on the wrapper will invoke the overridden *Server*'s *getName()*, and not the *Node*'s *getName()* as expected in the plugin. It can potentially lead to an application malfunction, if the expected method semantics differ from the semantics of the overriding method. In this case, the behavior expected in the plugin is split between the two API classes (the adaptee *Server* and its base class *Node*), and the adapter should be able to find the intended API methods.

*Solution.* To forward exactly to the API methods with expected semantics, in such cases the adapters must be able to make *non-virtual* calls to the intended methods, bypassing thus methods overridden in adaptees. For example, the forwarding implementation of the adapter method *getName()* shall issue a non-virtual call to *Node*'s *getName()* instead of calling the *getName()* method of its adaptee *Server*. Unfortunately, while

[21] Overriding is what Meyer calls *redefinition*, arguing that it cannot change the semantics, since otherwise "this is contrary to the spirit of redefinition. Redefinition should change the implementation of a routine, not its semantics" [Mey00, p. 482]. Moreover, one could argue that the overriding method is added to be called (e.g., as an enhancement), also from existing plugins. However, then it is questionable, why the API method being overridden is kept altogether. We refrain from further discussions on these subtle issues; instead, we aim for providing framework developers an adaptation technology of choice that they could apply on demand.

in such languages as .NET's Common Intermediate Language (CIL) it is possible to specify, whether a method call is virtual (e.g., CIL's instruction *callvirt*) or non-virtual (e.g., CIL's instruction *call*), in Java source or binary code it is not possible. In Java, our solution is to warn (at adapter generation time, when detecting such situations) framework developers about possible split behavior and let them manually verify the intention of redefining an API method, such as *getName()* in Figure 3.38.

### 3.3.2 Dangerous Assumptions about Objects' Physical Structure and Type

In case existing code makes assumptions about an object's physical structure (i.e., members' order) or its run-time type, introducing adapters may lead to that code malfunctioning at run time. In Java, such dangerous assumptions can be made by using reflection—a technique for working with the meta-level of an object-oriented system. In Java, reflection is restricted to introspection and permits to gather type information, create instances, and invoke methods of reflected types.

#### Assumptions about Objects' Physical Structure

An old API type and its corresponding reconstructed API type of the static adaptation layer differ in their physical structure due to the introduction of adapter-specific members, such as adaptee fields and surrogate constructors. Therefore, in case existing code makes assumptions about the physical structure of objects in use, introducing wrappers of the run-time adaptation layer may invalidate those assumptions.

**Plugin Calls Relying on Member Order.** Since we reconstruct exactly the syntactic part of the old API (i.e., type hierarchy, signatures of API methods), reflective method invocations in existing plugins execute properly after introducing the adapters. However, existing code may malfunction, when developers misuse reflective method calls to API types by relying on the relative order of type members. This may occur, for instance, when an array of constructors (or methods) declared by an API class is obtained by reflection, one member of the array is obtained by its index and invoked afterwards. For example, in the Java code of Figure 3.39 a plugin reads raw string data and uses them to reflectively create an instance of a type not known at compile time. After the API class is refactored and adapted, since the corresponding reconstructed API class contains also adapter-specific members, the wrong constructor may be found and invoked leading to a run-time exception or an unexpected behavior. In Figure 3.39, a run-time error will arise, because the adapter class adds an additional constructor dedicated for wrapping method arguments (the surrogate constructor, discussed in Section 3.2.3), invalidating thus the plugin assumption about the member order. As a consequence, the surrogate constructor will be invoked with an argument of type *String*, leading to a type casting exception.

*Solution.* To avoid malfunctioning adapters when using reflection, application developers must completely verify obtained references to meta-level structures. For example, they have to check, whether the full method information (i.e., the method name, all parameter types, and the return type) matches their expectations before invoking the method. Moreover, when using reflection the developers should not assume a fixed order of the API type members and use instead the full method information in reflective method invocations.

**Default Object Serialization.** Serialization is the process of translating (writing) an object into a sequence of bytes to be saved onto a storage medium or sent over a network. At later deserialization (reading), the bytes are translated back to a clone of the original object. For convenience, Java offers a default serialization mechanism, in which reflection is used internally to serialize and deserialize objects according to default serialization criteria. However, by using the default serialization developers commit themselves to the physical layout of the object being serialized. Should the object's class evolve in a new software version, in most of the cases the default serialization will fail to translate back already serialized objects due to the structural mismatch of the old and new class versions.[22]

For instance, assume that in the first framework version an object of an API class is serialized. Thereafter, the object's class is refactored and, hence, adapted in the second framework version. In case a wrapper

[22] An on-line discussion of common problems related to Java class evolution and serialization is available at `http://macchiato.com/columns/Durable4.html`.

Old API Class

```
public class Packet {
  private String message;

  public Packet(String message) {
    this.message = message;
  }
}
```

Reconstructed API Class

```
public class Packet {
  //adaptee object
  refactoredAPI.Packet pckt;

  //surrogate constructor
  public Packet(AdapterCache surrogateParameter, Object pckt) {
    this.pckt = (refactoredAPI.Packet) pckt;
  }

  //constructor for wrapping newly created instances
  public Packet(String message) {
    pckt = new refactoredAPI.Packet(message);
  }
}
```

Plugin

```
public class MyPlugin {
  public void create() {
    String className; //read in class name
    String value;   //read in value
    Object o = Class.forName(className).getConstructors()[0].newInstance(value);
  }
}
```

Figure 3.39: Dangerous use of reflection. The top part of the figure shows the snippets of the old API class (as expected in the plugin) and the corresponding reconstructed API class. In the presence of adapters, the plugin code in the bottom part of the figure will invoke a wrong constructor.

needs to be created for the serialized object, the default Java serialization mechanism will fail to restore the serialized object, since the latter now needs to be stored in the wrapper. Therefore, we need to tell adapters how to properly deserialize the objects to be used as adaptee objects. Moreover, in the opposite direction, we also need to tell adapters how to properly serialize adaptee objects, so that in a future deserialization the adapters could safely assume the objects being deserialized are indeed adaptee objects.

*Solution.* To support proper serialization in the presence of adapters, developers must implement a custom serialization process of their objects using standard language mechanisms (e.g., by providing the private instance methods *readObject()* and *writeObject()* in Java). In the adapters, we define, using the same language mechanisms (i.e., by implementing *readObject()* and *writeObject()* methods in Java), a custom serialization process, which is called by the language execution environment to serialize and deserialize adapter objects. In these adapter methods, we call in turn the custom serialization of adaptee objects emulating thus the standard serialization behavior with one level of method redirection, via adapters (as reflected by the pseudocode of Listing 3.5).

In more detail, Listing 3.6 shows insights of Java adapters related to the serialization mechanism, while the full serialization of Java adapters, including the support for custom object replacement, is provided in Appendix B.2. The adapter's *readObject()* method will be called by the Java serialization mechanism

Listing 3.5: Adapter serialization (general approach)

```
//adaptee field of type Adaptee
void deserialize(InputStream in){
  adaptee = new Adaptee();
  Method deserializer = adaptee.getClass().getMethod("deserialize");
  deserializer.invoke(in);
}

void serialize(OutputStream out){
  Method serializer = adaptee.getClass().getMethod("serialize");
  serializer.invoke(out);
}
```

Listing 3.6: Adapter serialization in Java (core algorithm)

```
//adaptee constains a (framework or plugin) adaptee object
private void readObject(ObjectInputStream in) throws IOException {
    Class<?> clazz = ... //the adaptee type,
                         //filled in at adapter generation time
    Constructor<?> constructor;
    try {
        constructor = clazz.getConstructor();
    } catch (NoSuchMethodException nsme) {
        try { //no public constructor available, seek protected
            constructor = clazz.getDeclaredConstructor();
        } catch (Exception ex) {
            //neither public nor protected default constructor found
            //custom serialization mechanism flawed, critical failure
        }
    }

    try {
        adaptee = constructor.newInstance();
    } catch (Exception ex) {
        //custom serialization mechanism flawed, critical failure
    }
    try {
        Method method = clazz.getDeclaredMethod("readObject",
                ObjectInputStream.class);
        method.setAccessible(true); //circumvent accessibility
        method.invoke(adaptee, in);
        method.setAccessible(false); //can only be false
    } catch (NoSuchMethodException nsme) {
        //custom serialization not found, critical failure
    } catch (Exception ex) {
        //custom serialization mechanism flawed, critical failure
    }
}

private void writeObject(ObjectOutputStream out) throws IOException {
    Class<?> clazz = adaptee.getClass();
    try {
        Method method = clazz.getDeclaredMethod("writeObject",
                ObjectOutputStream.class);
        method.setAccessible(true); //circumvent accessibility
        method.invoke(adaptee, out);
        method.setAccessible(false); //can only be false
    } catch (NoSuchMethodException nsme) {
        //custom serialization not found, critical failure
    } catch (Exception ex) {
        //custom serialization mechanism flawed, critical failure
    }
}
```

whenever an adaptee object should be read. It will instantiate an object of the adaptee and reflectively call that object's *readObject()* method to read the raw data into the adaptee object. In the opposite direction, the adapter's *writeObject()* method will be called by the Java serialization mechanism whenever an adapter object should be written out (e.g, to a file). The method will forward to the adaptee's *writeObject()* method to serialize the adaptee object instead of the adapter object. Thereby we emulate the Java serialization behavior in adapters by intercepting serialization calls from the language execution environment and forwarding them to the serialization methods defined in adaptees.

Use of Instanceof

```
public void record(Node node) {
  //do something general

  if (node instanceof Server) {
    //do something specific for servers
  }

  else if (node instanceof Workstation) {
    //do something specific for workstations
  }
}
```

Figure 3.40: Assumptions about the run-time type of an object. By investigating the run-time class of the object passed as a method argument, developers implement type-specific behavior in separate parts of the method.

### Assumptions about Objects' Type

As we discussed in Section 3.2.3, due to the polymorphism supported in Java, an object passed as a method argument (or a return value, as implied here and in the following discussion) does not have to be exactly of the parameter's formal type; the object type must only conform to the parameter type declared in the method signature. In many cases, objects of subtypes of the formal parameter type are passed as arguments, while developers can safely assume that these objects support at least the methods of the formal parameter type. However, using code introspection developers may assume that an object is of a more specific type than the formal type of the corresponding method parameter. Based on such assumptions, developers may implement certain behavior, which may be negatively impacted in the presence of adapters.

For an example, let us assume two API classes *Server* and *Workstation* both subclassing class *Node*. Objects of both subclasses are obtained calling a framework factory method *getNode()* with the formal return type *Node*. After calling the factory method, plugin developers send the obtained framework object to a plugin method *record()*, investigate the object's run-time type and implement different behavior (e.g., by reflectively calling the object's type-specific methods) depending on the object's actual type (Figure 3.40).

Whereas the example of Figure 3.40 uses an object of an API type (*Node*), we can extend the example to a situation where the formal parameter type is a library type. For instance, framework developers could have chosen *Object* as the type of the factory method's return value, while plugin developers would perform the rest of the run-time checking as in Listing 3.40.[23] Moreover, similar assumptions may be made for library collection types and arrays. For example, a framework method may return a collection of nodes as an instance of Java *List*, while plugin developers would write code that iterates over the collection's elements and explicitly casts them from the collection's internal representation (i.e., the type *Object*) to *Node*.

One may argue that in such cases programmers are not using the full potential of polymorphism available in Java: "It is almost always a mistake to explicitly check the class of an object" [JF88]. An arguably more adequate object-oriented implementation would provide two factory methods (returning *Server* and *Workstation*, respectively). Similarly, plugin developers would overload *record()* with *Server* and *Workstation* and leave the language execution environment do proper method dispatch depending on the actual type of the method argument. However, the aforementioned *instanceof* programming style is fairly common for programmers simulating "by-hand" method dispatch in their code. Often, it is preferred to avoid bloating the framework API with many API methods. In addition, some developers (debatably) argue for better code readability of this style comparing to overloading methods. Although we do not share this point of view, we want our technology to also work in the presence of such type assumptions.

As we discussed in detail in Section 3.2.3, when wrapping method arguments in adapter methods we cannot consider formal types of parameters and return values. Otherwise we could end up in one or both of the following problems:

[23] Definitely, this is an example of a bad programming style. Unfortunately, it is not rare.

1. Inadequate wrappers. The created wrapper is of a too general adapter type and cannot answer certain method invocations. For example, if we instantiated an adapter of *Node* to wrap the framework object of type *Server*, while the plugin invoked (e.g., using reflection) a *Server*'s specific method on the wrapper, the latter would not be able to respond to the invocation.

2. Object leakage. In case a parameter type were a library type, while the actual method argument were an object of a user-defined class, no wrapper would be created for that object. The same applies to passing library collections as method arguments and return values, which could contain objects of user-defined types. We call this situation *object leakage*—an object of one domain (i.e., the framework or the plugin) enters another domain (i.e., the plugin or the framework) without wrapping.

Such implicit type assumptions (e.g., reflected by using *instanceof* or type casting) are hard to derive from existing code requiring sophisticated code analysis. Moreover, in general, we cannot expect deriving them from plugins presumably unavailable for code analysis.

*Solution.* In our algorithm for adapter instantiation (described in Listing 3.4 on page 81), given an object to be wrapped we consider the object's class to instantiate the object's specialized adapter, which (by its definition) also supports the formal type of the method parameter. In the example of Figure 3.40, if the framework factory method returns a *Server* object, a wrapper instantiating the adapter for *Server* will be created to wrap the framework object. If the wrapper is later passed to *record()*, the *instanceof* clause for *Server* will succeed as expected inside the *record()* method.

Furthermore, whenever a library collection (or an array) is encountered in the run-time adaptation layer, the collection's (array's) elements are iterated, investigated and, possibly, wrapped similar to ordinary objects. For instance, if a collection of nodes is to be returned from the framework as a *List*, the collection will be detected and investigated in the run-time adaptation layer, and all collection's nodes will be wrapped before sending the collection to the plugin.

All in all, our algorithm for adapter instantiation protects from the object leakage and inadequate wrappers. Even in the case the algorithm fails to find the most specific adapter, we will dynamically create the latter at run time (Listing 3.4 on page 81).

### 3.3.3 Static and Run-Time Optimizations for Reducing Adapter Performance Overhead

The message forwarding used in the object version of the Adapter pattern (and, hence, in our pattern variants) inevitably implies performance penalties of at least one message redirection call per adapter method. The penalties increase when user-defined types are present in method signatures and, hence, the corresponding arguments need to be wrapped and unwrapped. In such cases, in addition to surrogate constructor calls and cache lookups (for wrapping), and accessor calls (for unwrapping), time is also consumed for type investigation in the **isAdapter** and **isUserDefined** functions (as discussed for adapter methods in Listing 3.3 of Section 3.2.3). For collection types, this is especially important, because each element of a collection needs to be investigated, and possibly wrapped or unwrapped. Moreover, the multiple method dispatch as performed in Covert Plugin Adapter of the White-Box Class Adapter pattern adds performance overhead of plugin type introspection. Finally, the dynamic adapter creation may negatively impact the overall performance of an adapted system and should be avoided, whenever possible. Concerned with performance overhead, we were able to considerably reduce it by applying static (at adapter generation time) and run time (at application execution time) optimizations.

#### Static Performance Optimizations

Our main optimization having a considerable impact on the performance of adapted systems is creating statically black-box and white-box class and interface adapters for all API types of the latest framework version. Thereby, we shift the time of adapter creation from application run time to the time of generating the static adaptation layer (i.e., before performing the actual framework upgrade). Although it comes for

the costs of additional physical space required to store the generated adapters, for most of the frameworks (with an notable exception of frameworks used in embedded systems, the adaptation of which we did not investigate) we consider the storage costs much less important than the gained run-time performance improvement of not creating most adapters dynamically.[24] Such "eager" adapter generation can be seen as a particular case of *partial evaluation* [Fut71], in the sense adapter creation is partially evaluated before the framework upgrade. Moreover, in case an adapter still has to be created dynamically, we store its definition for later reuse (possibly, also in application's subsequent executions).

As a further optimization in maximizing static adapter creation, one could consider analyzing framework sources to locate internal framework classes that subtype unrelated API types (the latter discussed in Section 3.2.4). For all such classes, one could create adapters statically, minimizing the need for dynamic adapter creation. However, since we assume no availability of plugins, the static adapter creation will still not be complete—at least, for certain plugin types their adapters may need to be created dynamically.

We statically optimize the algorithm for wrapping and unwrapping of method arguments and return values in adapter methods (summarized in Listing 3.3 on page 55). At adapter generation time, since all built-in value (e.g., *int*) and final (e.g., *java.lang.String*) types in the method signatures do not need wrapping and unwrapping, the code realizing the functions **isWrapper**, **isUserDefined**, and **isCollectionType** for method arguments of such types is not emitted decreasing the code size and the run-time overhead. Moreover, since the number of parameters of adapter methods is known at the time of adapter generation, we unroll the loop for parameter checking before emitting the code. As a consequence, parameterless methods add no additional overhead to the forwarding calls. Furthermore, certain collections and arrays need no treatment in adapter methods, such as the ones that are declared to contain only *Strings*. In addition, the object replacement as required by wrapping and unwrapping can be efficiently implemented in Java, since its stack architecture enables direct replacement of object references on the stack.

As another important static optimization, we allow a special kind of object leakage called *harmless object leakage*. Recalling our discussion in Section 3.3.2, object leakage may occur if an object of a non-library type of one domain crosses domain boundaries without adaptation. Although in general it may lead to run-time malfunctions, in some cases we can safely assume that in another domain the object will not be sent any message it cannot answer. For example, if the framework defines an internal (non-public) type subclassing *java.io.InputStream* and returns an object of this type to the plugin, since the latter is not aware of any framework internal methods and may invoke only object's methods defined in the standard Java *InputStream*, the object leakage is harmless. In these cases, we do not need to adapt such objects. This optimization is directly related to the notion of a user-defined API type as defined in our adaptation context, that is, an API type that could have been refactored by framework developers. Since in the aforementioned example there is no user-defined API type as a supertype of the framework's internal type, we can safely skip its adaptation. This optimization decision is valuable, because in most of the cases types used for input and output (which are time-consuming operations) are either library types or such harmless internal framework types, for which we avoid run-time overhead.

We considered reducing the size of the static adaptation layer created in exhaustive API adaptation. For that, one may take into account the refactoring information to demarcate only the API types actually refactored and, furthermore, all their dependent types. To find such types one could use, similar to Dig et al. [DNMJ08], the *points-to* analysis [Wei80]. Thereafter, it would be possible to reduce the static adaptation layer to contain only adapters for the API types that may indeed require adaptation. However, this optimization may lead to dangerous object leakage. Imagine a user-defined API class not affected by a refactoring and, furthermore, not dependent on any refactored API type. As a consequence, the optimization will carve out the class' adapter from the generated static adaptation layer, while the class' instances will never be wrapped in the run-time adaptation layer. If now a plugin object is sent as an argument to a method of an (unwrapped) instance of that API class, the object will bypass the run-time adaptation layer (i.e., will not be encountered in any adapter method) leaking in the framework object domain without wrapping. In general, a safe reduction of the static adaptation layer is possible only when, in addition to framework API types, plugins types can also be analyzed. However, since our main assumption is the unavailability of plugins for analysis and update, we did not perform this optimization in our adaptation context.

[24] As a side remark, recall that we do not have to create adapters for library types.

**Run-time Performance Optimizations**

At run time, we speed up the dynamic adapter lookup (when wrapping) by caching wrappers in the adapter cache using a Java (weak) hash map. Since references to objects in the adapter cache may hinder garbage collection, we use weak references, as supported in Java, to the adaptee objects cached and avoid thus unused objects remaining in memory. Whenever an adaptee object and its wrapper are not used anymore, they get garbage-collected automatically.

Coping with the run-time overhead introduced by multiple method dispatch of the White-Box Class Adapter pattern, we cache (in Covert Plugin Adapters of the White-Box Class Adapter pattern) the metadata about the presence or absence of overriding functionality in the plugin. As a consequence, the need for type introspection is reduced to one introspection per call of an adapter method implementing multiple method dispatch. As a possible future optimization, such cache could also be made persistent, therewith reducing the need for introspection in case the application is restarted.

### 3.3.4 Implications of Using the Adapter Design Pattern

The focus of the Adapter design pattern, also reflected in its structure, is to adapt interfaces—named sets of method signatures. Therefore, in the context of API refactoring the pattern is applicable only to a subset of all possible refactorings, namely, those affecting interfaces. We call the refactorings of this subset *basic adaptable refactorings*, meaning not that they are the most basic refactoring transformations possible, but rather that they are basic transformation units to be considered in a refactoring-based adaptation. In fact, we already gave examples of such refactorings, when classifying application-breaking API refactorings by the mechanics of the problems introduced (Section 2.2.3 on page 27). In this section, we overview such refactorings from the point of view of their resulting changes that affect existing callers. In our discussion, since we are interested only in framework API refactorings, we do not consider other refactorings (e.g., adding private members), possibly applied to framework types.

1. **Refactorings affecting API method signatures and locations.**[25] Examples are renaming methods (`RenameMethod`), adding, removing and permuting method parameters (e.g., `AddParameter`, `RemoveParameter`), and moving methods between classes related by inheritance (`PushDownMethod`) or by object composition (`MoveMethod`). Moreover, consistently with Dig and Johnson [DJ05], as members of this group we treat also refactorings specializing method return types and generalizing parameter types, replacing a primitive type with an object type (`ReplaceDataValueWithObject`), and replacing a type with a collection that contains several elements of the replaced type. Such changes can be considered refactorings, once the corresponding mapping from old to refactored types is provided [DJ05]. Since such refactorings make existing API methods appear differently or in different types than expected by old method callers and, moreover, may introduce conflicts with the signatures of plugin methods, refactorings of this group may lead to Non-localizable Functionality and Unintended Functionality.

2. **Refactorings affecting API type names and locations.** Examples are type renaming (`RenameClass`, `RenameInterface`, and the type renaming implied by `RenamePackage`) and moving of types (`MoveClass` and `MoveInterface`). Since such refactorings may make API methods appear in different (i.e., renamed) API types and, moreover, may introduce name conflicts of refactored API types and existing plugin types, refactorings of this group may lead to Non-localizable Functionality and (as a special case) Type Capture.

3. **Refactorings adding API types and methods.** Examples are adding to the API an empty class (`AddEmptyClass`),[26] adding a new method (`AddMethod`), extracting a method (`ExtractMethod`), and adding a class representing a new exception type (`AddException`). An interesting refactoring of

[25] By a method signature we mean the name of the method, the set of its parameters, and its return type.

[26] The refactoring *AddClass* that adds a class with an implementation can be seen as adding an empty class, and then adding to it new methods. We will discuss this *composite* refactoring later in this section.

this group, which is important for API adaptation, is `AddEncapsulatedField`, which adds to an API class a private field with a pair of accessor API methods (e.g., *get()* and *set()*) accessing and modifying the field's value. Since refactorings of this group may introduce conflicts of type names and method signatures of newly introduced API types and methods with existing plugin types and methods, refactorings of this group may lead to Type Capture and Unintended Functionality.

4. **Refactorings removing API types and methods.** Examples are inverses of the refactorings of the previous group, removing API types and methods (e.g., `RemoveEmptyClass`, `RemoveMethod`, `RemoveException`, `RemoveEncapsulatedField`). Since such refactorings remove previously available API types and methods, they lead to Missing Functionality. Consistently with our discussion from Section 2.2.3, as a special member of this group we also consider adding a new abstract API method, which can be seen as adding a concrete API method and then removing the method implementation.

In case API refactorings go beyond these basic adaptable refactorings, the Adapter pattern (and hence, our variants of the pattern) might be unable to compensate for such refactorings. For instance, since no notion of a public field is supported in the pattern, it cannot compensate for the refactorings of public API fields (e.g., `RenameField`, `MoveField` applied to such a field). In our approach, we require that all API fields are encapsulated (declared private and accessed by *get()* and *set()* accessor methods) to implement information hiding [Par72]; support for refactorings of encapsulated fields is then implied by the adaptation of the corresponding accessor methods.

Certain refactorings operate at an implementation level below method signatures. In particular, `InlineMethod` replaces a method call with the body of the called method and then deletes the latter. Since the Adapter pattern does not define any means to access method implementations (it is not the purpose of the pattern), we can compensate for this refactoring only by adding to the adapter the removed method (its signature and implementation) without changing the method that resulted from `InlineMethod`. Although such adaptation suffices in most of the cases, in general we cannot guarantee its behavioral correctness, because the inlined method could have been used as a (non-abstract) hook method in the old API.[27] In this case, the method added in the adapter will never be called back from the framework and, hence, the overriding plugin functionality will not be invoked (Unintended Framework Functionality: Inconsistent Method, shown by Figure 2.9 on page 31). Still, we can safely compensate for the opposite refactoring `ExtractMethod` by deleting the method in the reconstructed API (to avoid Unintended Plugin Functionality: Method Capture, shown by Figure 2.10 on page 33).

In case an old API method is deleted without a deprecate-replace-remove cycle, the corresponding implementation must be defined in an adapter. Although the Adapter pattern describes such opportunity [GHJV95, p. 142], it is not always possible in practice, because implementing such methods may require accessing private members of the refactored framework not visible from adapters.

### Adapting Composite Refactorings

What about more complex API refactorings, involving larger type and method changes? For them, we can rely on the Adapter design pattern, if a refactoring can be modeled (seen) as a *composite* of basic adaptable refactorings—a chain of basic refactorings, for which the pattern applies.[28]

For example, `ExtractClass` can be modeled as creating a new class (`AddEmptyClass`), adding an encapsulated field of the type of the new class to the base class (`AddEncapsulatedField`) and, by using the field, moving a set of public members of the base class to the new class. By compensating each such basic refactoring we will also compensate for the whole composite `ExtractClass`. However, if at least one of the constituents of a composite refactoring is not an adaptable refactoring, the whole complex refactoring

[27] In fact, in this case one could argue, that the transformation is not a refactoring, since it changes API semantics by removing a variability point—a hook method.

[28] The formal definition of a refactoring chain, as introduced by Roberts [Rob99], will be provided in Section 4.1.1.

cannot be adapted either. For instance, if one of the public members being moved in `ExtractClass` is a field, we cannot safely adapt the whole `ExtractClass` refactoring.

Similar discussions apply to other composite API refactorings, such as `AddClass`, `ExtractSubclass`, and `ExtractInterface`. In particular, we model `AddClass` as `AddEmptyClass` adding an empty API class, followed by a sequence of `AddMethod` refactorings adding API methods. Similar, we model `ExtractSubclass` as `AddEmptyClass` adding an empty subclass of the base class, followed by a sequence of `PushDownMethod` refactorings moving method down from the base class to the newly created subclass. Finally, we model `ExtractInterface` as `RenameClass` renaming the original class, followed by `AddInterface` adding an interface named as the class before the rename and implemented by the renamed class.

In many cases, the set of refactorings making up a composite refactoring can be modeled differently, depending on the developers' expertise and taste. It is therefore important to remember that the basic refactorings used for modeling a composite refactoring should not go beyond the type changes that can be applied to an interface (i.e., a named set of method signatures). Otherwise, certain constituents of the composite refactoring and, hence, the whole composite refactoring may not be compensated by the Adapter pattern. For instance, if we dissected furthermore the `AddEncapsulatedField` basic adaptable refactoring into a sequence of the refactorings `AddField` adding a public field, `HideField` reducing the field's visibility, and `AddMethod` adding the field's accessor methods, we could not compensate for the first refactoring in the chain (i.e., `AddField`) in isolation. For such cases, one needs to consider all related refactorings, and model them together as an adaptable refactoring (e.g., `AddEncapsulatedField`).

In general, it is hard to provide an unbiased estimation of how many complex refactorings can be modeled as chains of basic adaptable refactorings. Since such modeling is highly dependent on the experience of developers in applying refactorings, the results may vary considerably among developers. In our case study (Section 2.1) we managed to boil down all large refactorings to adaptable ones. Since also advanced architectural changes can be modeled as sequences of refactorings [TB01], we believe that the limitations of the Adapter design pattern still do not prohibit the realistic application of refactoring-based adaptation.

#### Adapters in the Presence of Thread Synchronization

If properly implemented, synchronization provides for multiple threads consistently accessing a shared state. By implementing a certain state synchronized, it is possible to avoid errors otherwise potentially introduced by multiple threads simultaneously accessing the state. In Java, synchronization is based on a monitor—a lock associated with every object—to control the exclusive access to the object. For such access, a thread needs to acquire and own the object's monitor; until the monitor is released by the owning thread, all other threads attempting to acquire the monitor are blocked. In Java, synchronization can be implemented at the level of methods (by declaring them *synchronized*) or at the level of statements (by placing them in a *synchronized* code block) [GJSB05].

Since wrappers are objects physically different from the adaptee objects, introducing adapters splits the monitors of the objects being adapted between the wrapped objects and their wrappers (similar to how it splits the object identity, as discussed in Section 3.3.1 on page 85). This split should not pose additional problems if synchronization is implemented at the level of API methods, because in this case the corresponding adapter methods are also synchronized, by construction. By forwarding to the adaptee objects, such wrappers' methods also share their adaptees' monitors, thereby preserving expected synchronization.

However, if access to a certain framework shared state is realized in a synchronized code block of a plugin, adapters introduced do not capture this synchronization access. As a consequence, splitting of monitors of wrappers and wrapped objects may become dangerous. For instance, a framework thread can potentially access the shared state simultaneously to a plugin thread modifying that shared state. Because the synchronization is realized at the implementation level below method signatures (in other words, the synchronization contract is defined at the level of method statements, and not of method signatures), the Adapter pattern cannot correctly cope with it. As a consequence, the intended synchronization is broken leading to an inconsistent shared state.

While considering this as a limitation of the Adapter pattern, one could also argue for preferring synchronized methods over synchronized statements to realize access to a shared framework state. Whereas in the former

case, the synchronization contract is defined explicitly at the level of API methods, in the latter case the contract is implicit and not enforced by the API (hopefully, it is properly reflected in API documentation). Arguably, such implicit contract can be broken not only by introducing adapters, but also by application developers consciously or unconsciously disregarding it while implementing plugins. A general solution for developing reusable software is to always make component contracts explicit [GAO95].

#### Equivocal Adaptees

Let us conclude with a problem connected to the object composition as used in our adapters. In the first framework release the framework API contains a concrete class *Node* with the methods *print()* (for displaying user messages to the console) and *log()* (for writing node contents to a database). In the second framework release, by reconsidering domain-specific requirements of the framework, its developers realize that printing should be done only by workstations, while logging only by servers. Therefore, framework developers apply to *Node* two API refactorings: `PushDownMethod` moving *log()* to the *Node*'s subclass *Server*, and `ExtractSubclass` moving *print()* to the *Node*'s new subclass *Workstation*. At the same time, the *Node* class remains public and non-abstract.

In exhaustive API adaptation both the Black-Box Class Adapter pattern and the White-Box Class Adapter pattern should be applied to compensate for *Node*'s refactorings. However, at adapter creation time, we do not know, which of the two API classes (i.e., *Workstation* or *Server*) we should use in the patterns' applications. We call this problem *equivocal adaptees* (Figure 3.41).

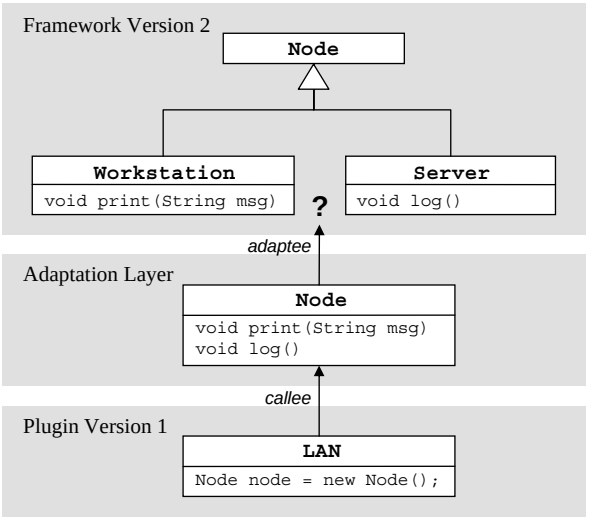
Non-deterministically deciding for one of the possible adaptees may lead to creating an adaptee object that cannot answer messages at run time (e.g., a *Workstation* adaptee object when asked to log a message). A possible solution would be asking developers to specify the API class to be used in the particular adapter pattern application. Since developers know about the semantics of the API refactorings they applied, their choice may be more appropriate than choosing the API class non-deterministically. For example, by considering developer documentation (e.g, source comments or a developer guide) framework developers may assume that existing plugins do not ask an instance of *Node* for logging, in case the instance is used in existing plugins as a workstation. In this case, framework developers would specify *Workstation* to be used in the applications of the Black-Box and White-Box Class Adapter patterns.

In fact, the problem of equivocal adaptees may indeed lead to run-time errors only when the plugin is calling an API class constructor. In our example, the problem appears, if the plugin invokes a constructor of *Node*, either directly (in black-box framework reuse) or via a constructor chain (in white-box framework reuse). Otherwise, if the node is created in the framework and returned then to the plugin (which is possible in black-box framework reuse), we know precisely the type of the node (i.e., *Workstation* or *Server*) and can decide for the appropriate adaptee. Although in this case the adaptee object could still fail to answer messages at run time, such errors would indicate an abuse in the plugin (e.g., asking a workstation to log to a database) and not the problem of equivocal adaptees.

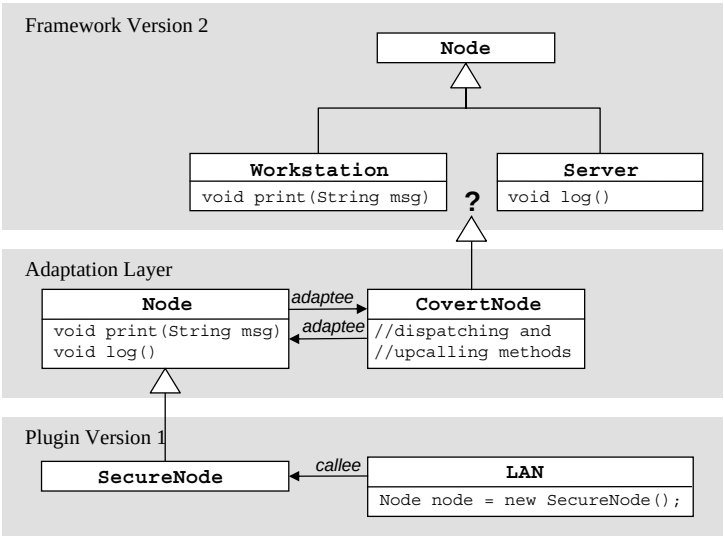
In general, the problem of equivocal adaptees is connected to the purpose of such refactorings as `Extract-Subclass` and `PushDownMethod`—refactorings for specialization [Opd92, pp. 71–86]. In case the set of possible objects of an original class is split in mutually disjoint object sets of the specialized subclasses, it is in general impossible to safely map objects of the old type to the objects of the new types without additional mapping criteria (e.g., specified by developers). However, although we envisage the problem of equivocal adaptees, we cannot estimate its practical impact, since we did not encounter such combinations of refactorings in our case study and the studies reported by other researchers [DJ05, DR08, SJM08].

## 3.4 Summary

The conventional description of the Adapter design pattern focuses on integrating independently developed components and assumes black-box component reuse. However, we want to apply the pattern to accommodate for refactorings of framework APIs and, therefore, need to precisely denote the particularities of our




(a) Application of the Black-Box Class Adapter pattern. It is unclear, which API class should be used as the adaptee of the reconstructed *Node*.




(b) Application of the White-Box Class Adapter pattern. It is unclear, which API class should be subclassed by the covert adapter class *CovertNode*.

Figure 3.41: Equivocal adaptees. In the application of `PushDownMethod` and `ExtractSubclass` to *Node* some of the methods were moved to different subclasses of *Node*. As a consequence, it is not possible to automatically decide, which API class should be used from the *Node*'s black-box and white-box class adapters.

adaptation context. Since plugins are developed reusing the framework API types by object composition and by inheritance, they are highly dependent on the framework API. Moreover, the framework is also dependent on the plugins due to the callback mechanism accessing overriding plugin functionality. In addition, our main goal—binary backward compatibility—requires extra means to avoid plugin recompilation. With these particularities in mind, we design the adaptation of refactored framework APIs.

Considering the adaptation of refactored API types, we define four variants of the Adapter design pattern to compensate for refactorings of API classes and API interfaces in black-box framework reuse and white-box framework reuse. While black-box adapters support plugins in calling the refactored API, white-box adapters

support the refactored framework in calling back old plugins. The most sophisticated adapters—white-box class adapters—support bidirectional communication between the framework and plugins, and protect against inadvertent overriding of API methods.

Considering the adaptation of the framework API as a whole, we perform exhaustive API adaptation by statically generating adapters for all possibly refactored API types and then instantiating the corresponding adapters at run time. The generated adapters make up the static adaptation layer, which reconstructs exactly the framework API expected by an old plugin. The instances of adapters (i.e., wrappers) form the run-time adaptation layer, which separates the object domains of the framework and the plugin. In the run-time adaptation layer, all objects crossing domain boundaries are wrapped and unwrapped according to their domain of origin. In addition, exhaustive API adaptation encapsulates decisions on how to properly create and instantiate adapters, and reuse wrappers depending on the properties of the objects to be wrapped.

Delimiting our technology, we discuss challenges we met, solutions adopted, and limitations we had to accept. With the help of the adapter cache implemented in the adaptation layer we solve the object schizophrenia caused by the object composition as used in our adapters. Moreover, anticipating certain assumptions developers may make about the run-time structure and type of the objects they are using, we either address these issues directly in exhaustive API adaptation or suggest additional adaptation means otherwise. Using static and run-time optimizations, we decrease the performance overhead inherent in adapters. Since the Adapter design pattern itself cannot compensate for all possible refactorings, we discuss API refactorings and their combinations, for which the pattern can be applied in our adaptation context.

To conclude, although focused on API refactorings while designing our adaptation patterns and exhaustive API adaptation, we do not commit our design presented in this chapter to compensate for API refactorings only. Indeed, all our examples are adapting syntactical API changes. However, nothing prevents developers from extending and combining black-box and white-box adapters with means for adapting changes beyond refactorings. For instance, if a modified API method throws an exception, while an old plugin expects a default action to be performed instead (an example of a so-called *protocol mismatch* [YS97]), developers may catch the exception and implement the default action in a method of the corresponding adapter. By doing that, developers reuse all decisions (e.g., adapter instantiation and cache, object wrapping, identity preservation) realized in our adaptation. That is, while our adaptation is designed to serve as an immediate solution for most application-breaking API changes (i.e., refactorings), it is also extensible to add adaptation support for other application-breaking API changes. In particular, in Section 5.2.1 we show how we can combine adapting API refactorings with adapting protocol mismatches.

*"Transforming programs is easy. Preserving behavior is hard."*

–Donald Bradly Roberts
*Practical analysis for refactoring* [Rob99, p. 30]

# 4

# Formalizing Refactoring-Based Adapter Derivation

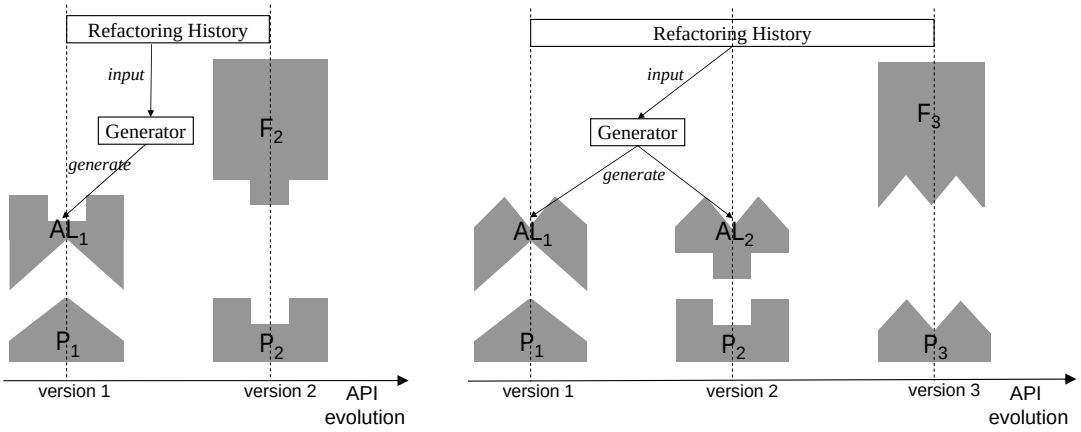
When upgrading software frameworks and libraries, the APIs of which are evolving via refactorings, and where dependent applications are not available for update, Dig and Johnson [DJ05] suggest to treat the history of API refactorings as a specification of component API changes. Furthermore, Dig et al. [DNMJ08] show how to use such specification reconciling the refactored component and its dependent applications. The authors manually specify, for several application-breaking API refactorings, compensating refactorings that inline adaptation code into the API of the upgraded component. The adaptation code reconstructs the syntactic part of the old API affected by refactorings, and either redirects calls of old API methods from existing application code to invoke refactored API methods, or restores deleted API methods [DNMJ08].

This elegant engineering solution is, however, based on the intuition of developers: the adaptation is hand-coded in the Eclipse integrated development environment [Ecl] as mappings from original application-breaking API refactorings to compensating refactorings. The authors do not discuss, why exactly these particular compensating refactorings are chosen and how they are related to the semantics of the original API refactorings. Without such explicit correlation, it is not clear precisely how to systematically define adaptation means for other, possibly more complex API refactorings than the refactorings addressed by Dig et al. [DNMJ08]. Moreover, it is not clear how this correlation should be implemented on platforms other than Eclipse. As another consequence of the lack of rigor in their adaptation definition, the authors do not consider the addition of API types and methods as application-breaking, although such additions may indeed lead to conflicts with existing applications (as we discussed for Unintended Functionality and Type Capture in Section 2.2.3).

Evidently, what is needed is a more rigorous basis for describing refactoring-based adaptation. For an application-breaking API refactoring, at the very least we should be able to say precisely what is the intended meaning of its compensating adaptation action, and how this action is related to the API refactoring. More ambitiously, we should also be able to use this rigor to automate the adaptation in a platform-independent manner. Addressing these issues, this chapter brings the following thesis contribution:

**C6: Rigorous refactoring-based adapter derivation.** To rigorously define refactoring-based adaptation in the context of framework API evolution via refactorings, we use the semantics of application-breaking API refactorings. The key idea is to define an adaptation transformation as an explicit semantic entity, the semantics of which depend on the formal semantics of the original API refactoring, for which this adaptation transformation compensates. More specifically, a compensating transformation is *an inverse* of an original API refactoring with regard to all syntactic program properties affected by the API refactoring. Because we disregard all other program properties but syntactic ones, we weaken the definition of a real inverse transformation, not considering *all* program properties affected by an original transformation. Therefore, we call our compensating transformations *weak refactoring inverses*.

Besides reasoning about the correlation between application-breaking API refactorings and adaptation transformations, the rigor of weak refactoring inverses permits us to automate our adaptation. Since




(a) First framework upgrade. Adaptation layer $AL_1$ is generated for the plugin $P_1$ basing on the trace of application-breaking API refactorings that occurred between the framework versions $F_1$ and $F_2$.

(b) Second framework upgrade. Adaptation layer $AL_1$ is generated for the plugin $P_1$ considering all application-breaking API refactorings in the history, while adaptation layer $AL_2$ is generated for the plugin $P_2$ considering only application-breaking API refactorings occurred between the framework versions $F_2$ and $F_3$.

Figure 4.1: Refactoring-based adaptation. Given the latest framework API and the trace of API refactorings that occurred between previous and the last framework versions, adaptation layers are generated, which translate between existing plugins and the upgraded framework. For plugins developed with the latest framework version, no adaptation layer is required.

we want plugins to use the latest, improved framework, we cannot invert application-breaking API refactorings directly on framework API types. Instead, we use certain weak refactoring inverses—called *comebacks*—to generate adapters. A comeback is a weak refactoring inverse that can be executed on adapters. When generating adapters, we create an adaptation layer that corresponds to the latest framework version and then invert application-breaking API refactorings in the history of framework evolution by executing the corresponding comebacks on adapters of the adaptation layer (Figure 4.1). Since for an old plugin its difference to the latest framework API is expressed as a sequence of refactorings in the refactoring history, by considering that difference we can generate adapters for any previous plugin version (as shown in Figure 4.1(b)). As a consequence, adapted plugins of different versions are provided with their version-specific adaptation layers, which use directly the latest framework version, avoiding the performance penalties otherwise implied by stacking multiple adapters.

Since a comeback is rigorously defined, it is possible to formally prove, that its execution indeed constructs an adapter that compensates for the comeback's corresponding application-breaking API refactoring. Moreover, the semantics of the refactoring-based adaptation as a whole is defined as the execution of a sequence of comebacks that correspond to the application-breaking API refactorings in the refactoring history.

The rest of the chapter is organized as follows: Since we base our definitions on existing work on refactoring formalization [Opd92, Rob99], we overview that work in Section 4.1. To give an intuition of our approach for refactoring-based adapter derivation, we continue then with an informal introduction to the key concepts involved in Section 4.2. Thereafter, we provide formal definitions and show proof examples of weak refactoring inverses and comebacks in Section 4.3, conclude in Section 4.4, while providing a set of further examples and proofs of weak refactoring inverses and comebacks for several common refactorings in Appendix D.

**Terminology Note**

An **analysis function** is a mathematical function with parameters. It yields a result value when applied to arguments. For example, Superclass($X$) is a function.

A logic **predicate** is the unit of composition in FOPL formulæ. It may contain unbound variables and evaluates to a boolean value with the help of an interpretation. Predicates can be built using analysis functions and mathematical relations. For example, Superclass($X$) $= Y$ and IsClass($X$) are predicates.

An **assertion** is a predicate with all variables bound. It serves as a statement and can be used to specify **preconditions**. For example, IsClass($'$*java.lang.Object*$'$) states that *java.lang.Object* is the name of a class.

## 4.1 Background on Refactoring Formalization in First-Order Predicate Logic

Refactorings—behavior-preserving source-to-source program transformations—are program transformations with well-defined semantics [Opd92, Rob99, MEDJ05, Eet07, Tic01, Wer99]. The research on formalizing refactorings has its origins in the works on formalizing the correctness of general program transformations (e.g., [BBP$^+$79, PS83]). In particular, we base our formal definitions on the refactoring formalizations developed by Opdyke [Opd92] and Roberts [Rob99].

### 4.1.1 Refactoring Definitions of Opdyke and Roberts

As a first step in formalizing refactorings, in his PhD thesis Opdyke [Opd92] defines seven program invariants refactorings should preserve (e.g., Distinct Class Names, Type-Safe Assignments, and Unique Superclass). He then defines twenty-three primitive refactorings, such as `AddClass`, `RenameMethod`, and `ChangeType`. Each refactoring definition consists of an informal description of a transformation and a *precondition*—a set of statements that ensure that the execution of the transformation will not change the program behavior (i.e., will maintain the invariants).

Opdyke uses first-order predicate logic (FOPL) [Smu68] to formaly specify refactoring preconditions. FOPL augments the set of standard logic connectives ($\neg$, $\wedge$, $\vee$, $\equiv$, $\Rightarrow$, $\Leftrightarrow$) and quantors ($\exists$, $\forall$) with symbols for specifying functions and predicates. The latter are used to specify preconditions as assertions—predicates with all variables bound. Whether an assertion is true on a program $P$ is determined by evaluating the assertion under the interpretation $I_P = (S, \cdot^I)$, with the program's source code $S$ as the domain of discourse and the mapping $\cdot^I$ specifying the meaning of assertions in the context of $S$. The evaluation of an assertion results exactly in one of the two boolean values: *true* or *false*. In practice, the mapping $\cdot^I$ can be implemented by a tool chain consisting of scanner, lexer, and parser [ASU86], which extracts information about the source code $S$ and represents it in the form of assertions in $P$. Definition 2 reflects the relation between the program $P$ and its source code $S$, as used in the rest of this chapter.

**Definition 2** (Program and Source Code)
An object-oriented program[1] $P$ is the finite set of assertions that represent information about the structure of the source code $S$.

In the following, if a precondition *pre* is true on a program $P$, it is denoted by $\models_{I_P} pre$ or simply by $P \models pre$. Otherwise, if the precondition *pre* is false on a program $P$, it is denoted by $\not\models_{I_P} pre$ or simply by $P \not\models pre$.

[1] In its original sense, the term *program* denotes a sequence of instructions executed or interpreted by the computer. In the remainder of this chapter, we focus on the structure of these instructions, that is, the classes and methods, in which the instructions are grouped.

Based on the work of Opdyke [Opd92], in his PhD thesis Roberts extends the formalization of refactorings with assertion transformations modeling the program transformations caused by the execution of refactorings [Rob99]. From these assertions, postconditions can be derived [KK04]. We slightly extend Roberts' definition of a refactoring [Rob99, page 37] to also reflect the property of a refactoring as a parameterizable transformation.

**Definition 3** (Refactoring)
A refactoring is an ordered triple $\mathtt{R} := (pre, trans, post)$ where *pre* is a set[2] of formulæ that must be true on a program $P$ for R to be legal, *trans* is the program transformation, and *post* is a function from assertions to assertions that transforms legal assertions whenever *trans* transforms programs. All unbound variables in *pre* and *post* are formal parameters of the refactoring and will be bound to concrete values whenever R is applied.

Because we base our work on the works of Opdyke and Roberts, we focus on reasoning about refactorings' program transformations in terms of preconditions and assertion transformations, and not on the transformations themselves. The reason for that is a practical consideration: while there has already been considerable research on formalizing the transformations of refactorings (e.g., as graph rewritings [Eet07, EJ04, HJE06] or term rewriting [Bat07]), the results have barely entered practice yet. Therefore, in the following the program transformation *trans* will only be informally described and the transformation of assertions (i.e., *post*) is explored instead. The application of a postcondition transformation *post* to a program $P$ will be referred to shortly as $post(P)$ hereinafter, while $trans(S)$ denotes the transformation of the source code $S$.

In fact, the assertion transformation of Definition 3 (i.e., *post*) has very much in common with the predicate transformer described by Dijkstra with his algorithm for deriving the weakest precondition of program code [Dij75].[3] However, the notation Roberts uses for specifying a *post* is different and requires a short explanation, since we will heavily use his formalism in our formal definitions and proofs. As a terminology remark, while Roberts uses the $\perp$ symbol for indicating that a function is not defined for a specific input, we will use a less known notation from computability theory [San88, pages 2–3]: $f(a)\downarrow$ means that the function $f$ is defined (or converges) for the input $a$ and $f(a)\uparrow$ indicates that $f$ is undefined for the input $a$. The motivation behind that decision is that the $\perp$ symbol denotes the truth value *false* when interpreting logic formulæ. To avoid ambiguity in case both truth values and convergence need to be expressed in an assertion transformation, we use different symbols for the two concepts.

While Roberts does not formally define the assertion transformation *post*, we introduce its formal definition to concisely use it in the following proofs. Formally, an assertion transformation *post* is a total function from assertions to assertions and is equivalent to the identity function except for a finite number of assertions—only those assertions that are changed are specified. The syntax for transforming a single assertion is

$$assertion' = assertion[x_1/y_1]\cdots[x_n/y_n],$$

where $x_1,\ldots,x_n$ are in the domain of the assertion[4] or will be added to it:

$$\mathrm{dom}(assertion') := \mathrm{dom}(assertion) \cup \bigcup_{i=1}^{n} \{x_i\}\,.$$

Each $y_i$ will be added to the codomain of the assertion if it is not the symbol $\uparrow$:

$$\mathrm{cod}(assertion') := \mathrm{cod}(assertion) \cup \{y_i | y_i \in \{y_1,\ldots,y_n\} \wedge y_i \neq \uparrow\}\,,$$

Otherwise, the corresponding $x_i$ is removed from the domain of the assertion:

$$\mathrm{dom}(assertion') := \mathrm{dom}(assertion) \setminus \{x_i | x_i \in \{x_1,\ldots,x_n\} \wedge y_i \in \{y_1,\ldots,y_n\} \wedge y_i = \uparrow\}$$

[2] Strictly speaking, *pre* is a totally ordered set (a binary, transitive, antisymmetric, and total relation on a set), because assertions are assumed to be evaluated from left to right in order to prevent computing analysis functions on undefined input.

[3] As indicated by its name, a predicate transformer modifies predicates. By transforming predicates, it also transforms all assertions defined in terms of those predicates. By contrast, an assertion transformation *post* modifies assertions directly. The conceptual difference between the two transformations is the level of abstraction used to define the semantics of the transformations (i.e., transforming predicates versus transforming assertions). To the best of our knowledge, there is no existing work formally relating predicate and assertion transformations.

[4] By the domain of the assertion we mean the set of all valid inputs for the assertion.

This helps to undefine assertions for a certain input. To avoid having to specify value assignments for a large (and possibly unknown) set of inputs, quantified expressions are also permitted as in $\forall x \in X.assertion[x/y]$, and quantified expressions can be specified sequentially.

As an example, the assertion transformation $\text{IsClass}' = \text{IsClass}[newClass/true]$ reflects the addition of the class *newClass* to the program, while not changing the rest of the values of the predicate IsClass. As a more complex example, the following assertion transformation reflects the fact, that when a class *class* is removed, its subclasses are connected to its superclass in the class hierarchy.

$\text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[c/\text{Superclass}(class)][class/\uparrow]$

We will formally introduce IsClass, Superclass, and Subclasses as well as other analysis functions used to build predicates in Section 4.1.3.

Roberts also introduces chains of refactorings: "Refactorings are typically applied in sequence. This allows large design changes to be composed of a sequence of smaller, more primitive refactorings. Since each step is a refactoring, and therefore, behavior-preserving, the entire composition is also a refactoring." [Rob99, page 43]

**Definition 4** (Chain of Refactorings [Rob99, page 36])
A chain of refactorings is a sequence of legal refactorings, $\langle \mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n \rangle$, that are applied to a program sequentially and without intervening changes to the program.

As an example, given two refactorings `RenameClass` and `AddMethod`, a chain of the two refactorings is $\langle$`RenameClass`,`AddMethod`$\rangle$. Details on how to derive the composite precondition of a sequence of refactorings can be found in Roberts' PhD thesis [Rob99, pages 39–52].

### 4.1.2 Metamodel of the Language Features Covered by Formal Refactoring Definitions

While Roberts developed his refactoring formalism for Smalltalk, we want to apply it to other object-oriented programming languages, such as Java. Therefore, it is important to precisely delimit the language features supported in Roberts' formalization, so that we can reuse his work for those features in other programming languages. Figure 4.2 summarizes, as an UML model [Obj05], all language features, for which the refactorings are formally defined by Roberts [Rob99]. The annotations in the figure correspond to Roberts' analysis functions listed for the sake of presentation in the following Section 4.1.3.

Figure 4.2 reflects an abstract object-oriented language with single inheritance, which is in fact very similar to the core model of the FAMOOS project [DTS99, page 4]. The model is fairly simple: it ignores concepts such as interfaces, abstract methods, class and member visibility, method return types and parameters (and,

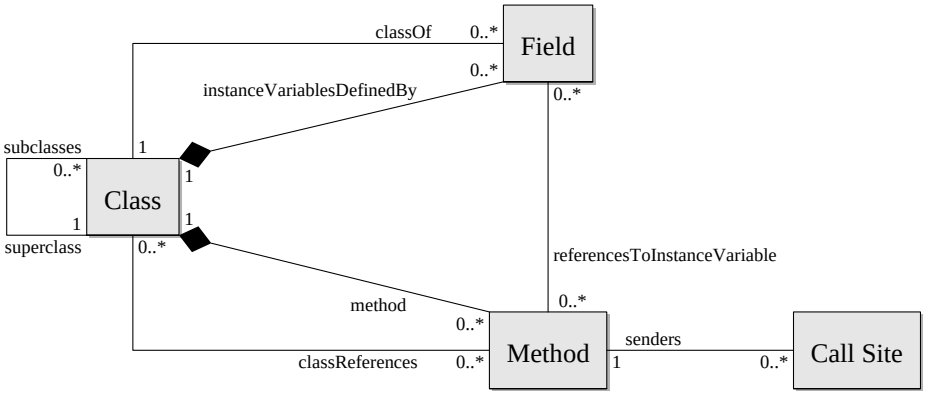



Figure 4.2: UML model of the supported language features. The labels are annotated with analysis functions as defined by Roberts [Rob99].

therefore, also method overloading), exceptions, enumerations, attributes/annotations, and events. Concrete implementations of the technology will need to add support for features specific to their target programming language (e.g., delegates or generic types).

Since we base our work on the refactoring formalization of Roberts [Rob99], our formal refactoring-based adaptation presented in the rest of this chapter is limited to adapting the refactorings applied to the language features of Figure 4.2. Defining the adaptation of refactorings applied to other language features (e.g., interfaces, enumerations, or annotations) would require formalizing the refactorings of those features first. We consider this extension as a future work, which is out of the scope of this thesis. Instead, our main goal is to show how to systematically define a formal refactoring-based adaptation for a given object-oriented programming language. However, in practice we do support other refactorings in our adaptation tool by implementing corresponding compensating transformations operationally (as Prolog rules, as we will show in the tool's implementation described in Section 5.1), without proving them formally in first-order predicate logic.

### 4.1.3 Roberts' Basic and Derived Analysis Functions Adopted for Refactoring-Based Adaptation

This section lists basic and derived analysis functions for expressing program properties and for reasoning about them, as we will need to formalize our adaptation technology. The functions adopt those developed by Roberts in his thesis [Rob99, pages 30–34], sometimes with slight notational modifications. One reason for these modifications has already been mentioned: instead of the $\perp$ symbol used by Roberts for indicating that a function is not defined for a specific input, we will use $f(a)\uparrow$ to indicate that $f$ is undefined for the input $a$. In addition to avoiding fallacious interpretation, this notation has another advantage of allowing to test whether a function converges for a given input. Another reason for the modifications to some of Roberts' ideas is that the Smalltalk programming language is dynamically typed, while our work focuses on the statically-typed Java. As a consequence, these functions do not capture the run-time typing information (e.g., all types of objects assigned to fields of classes) but rather their static counterpart (the static type of a field in a class). As a small remark, due to Smalltalk-specific terminology, Roberts' term for "method" is *selector*.

#### Basic Analysis Functions

**IsClass(*class*)** Total function that is true if a class named *class* exists in the system and false otherwise.

**Superclass(*class*)** Partial function returning the immediate superclass of *class*.

**InstanceVariablesDefinedBy(*class*)** Partial function returning the set of instance variables defined by *class*.

**ClassOf(*class, varName*)** Partial function returning the type of the instance variable *varName* of *class*.

**ReferencesToInstanceVariable(*class, varName*)** Partial function returning the set of class, selector pairs of all of the methods that refer to the instance variable *varName* defined in *class*.

**ClassReferences(*class*)** Partial function returning the set of all class, selector pairs of all methods that reference *class*.

**Method(*class, selector*)** Partial function returning the abstract syntax tree representing the method defined by *selector* in *class*.

**Senders(*class, selector*)** Partial function returning the set of all sites that call the method defined by *selector* in *class*.

**Derived Analysis Functions and Formulæ**

**Subclasses(*class*)** Returns the set of all immediate subclasses of *class*.

$$\text{Subclasses}(class) := \{c | \text{Superclass}(c) = class\}$$

**Subclasses$^*$(*class*)** Returns all of the subclasses of *class* in the inheritance hierarchy. Computes the transitive closure of the Subclasses function.

$$\text{Subclasses}^*(class) := \begin{cases} \emptyset & \text{if Subclasses}(class) = \emptyset \\ \bigcup_{c \in \text{Subclasses}(class)} \{c\} \cup \text{Subclasses}^*(c) & \text{otherwise} \end{cases}$$

**Superclass$^*$(*class*)** Returns all of the ancestors of *class* in the inheritance hierarchy. Computes the transitive closure of the Superclass function.

$$\text{Superclass}^*(class) := \begin{cases} \emptyset & \text{if Superclass}(class)\uparrow \\ \{\text{Superclass}(class)\} \cup \text{Superclass}^*(\text{Superclass}(class)) & \text{otherwise} \end{cases}$$

**DefinesSelector(*class, selector*)** True if *class* locally defines the method *selector*.

$$\text{DefinesSelector}(class, selector) := \text{Method}(class, selector)\downarrow$$

**LookedUpMethod(*class, selector*)** If *class* locally defines the method *selector*, returns *selector* body. Otherwise, returns the body of the most specific *selector* inherited by *class*. If there is no such method, this function is undefined.

$$\text{LookedUpMethod}(class, selector) := \begin{cases} \text{Method}(class, selector) & \text{if Method}(class, selector)\downarrow \\ \text{LookedUpMethod}(\text{Superclass}(class), selector) & \text{otherwise} \end{cases}$$

**UnderstandsSelector(*class, selector*)** True if *class* defines or inherits the method *selector*.

$$\text{UnderstandsSelector}(class, selector) := \exists c \in \{class\} \cup \text{Superclass}^*(class). \text{DefinesSelector}(c, selector)$$

**UnboundVariables(*class, selector*)** Returns the set of all non-local variables referenced from *selector* in *class*.

$$\text{UnboundVariables}(class, selector) := \{v | (class, selector) \in \text{ReferencesToInstanceVariable}(v)\}$$

**DefinesInstanceVariable(*class, variable*)** True if *class* locally defines the instance variable *variable*.

$$\text{DefinesInstanceVariable}(class, variable) := variable \in \text{InstanceVariablesDefinedBy}(class)$$

**HierarchyReferencesInstVariable(*class, variable*)** True if *class*, any of its superclasses, or any of its subclasses references the instance variable *variable*.

$$\text{HierarchyReferencesInstVariable}(class, variable) := \exists c \in \{class\} \cup \text{Superclass}^*(class) \cup \text{Subclasses}^*(class). \text{ReferencesToInstanceVariable}(c, variable) \neq \emptyset$$

**IsEmptyClass(*class*)** True if *class* does not define any methods and instance variables.

$$\text{IsEmptyClass}(class) := \nexists s.\text{DefinesSelector}(class, s) \wedge \nexists v.\text{DefinesInstanceVariable}(class, v)$$

**MostSpecificSuperclass(*class1, class2*)** Returns the most specific common superclass of *class1* and *class2*.

$$\text{MostSpecificSuperclass}(class1, class2) := \begin{cases} class1 & \text{if } class1 = class2 \\ class1 & \text{if } class1 \in \text{Superclass}^*(class2) \\ \text{MostSpecificSuperclass}(\text{Superclass}(class1), class2) & \text{otherwise} \end{cases}$$

### 4.1.4 Examples of Formal Refactoring Definitions: `RenameClass` and `MoveMethod`

To give an impression of formal refactoring definitions, we conclude our background section with two examples of refactoring definitions: `RenameClass` and `MoveMethod`.

#### `RenameClass`

The `RenameClass` refactoring renames an existing class in the program and updates all references. It updates the references in the source code with the help of the *RenameReferences* analysis function, the inner workings of which are not important here. The following definition is adopted from Roberts [Rob99, p. 105].

`RenameClass`(*class*, *newClass*)

$$pre : \text{IsClass}(class) \wedge \tag{1.1}$$
$$\neg\text{IsClass}(newClass) \tag{1.2}$$

---

$$post : \text{IsClass}' = \text{IsClass}[class/false][newClass/true] \tag{1.3}$$
$$\text{ClassReferences}' = \forall c \neq class.\forall s \in \{s|(class, s) \in \text{ClassReferences}(c)\}.$$
$$\text{ClassReferences}[class/\uparrow][newClass/\text{ClassReferences}(class)]$$
$$[c/\text{ClassReferences}(c) \setminus \{(class, s)\} \cup \{(newClass, s)\}] \tag{1.4}$$
$$\text{ReferencesToInstanceVariable}' = \forall (c, v, s) \in$$
$$\{(c, v, s)|(class, s) \in \text{ReferencesToInstanceVariable}(c, v) \wedge c \neq class\}.$$
$$\forall lv.\forall lr \in \{lr|(class, lr) \in \text{ReferencesToInstanceVariable}(class, lv)\}.$$
$$\text{ReferencesToInstanceVariable}[(class, lv)/\uparrow][(newClass, lv)/$$
$$\text{ReferencesToInstanceVariable}(class, lv) \setminus \{(class, lr)\} \cup \{(newClass, lr)\}]$$
$$[(c, v)/\text{ReferencesToInstanceVariable}(c, v) \setminus \{(class, s)\} \cup \{(newClass, s)\}] \tag{1.5}$$
$$\text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[class/\uparrow]$$
$$[newClass/\text{Superclass}(class)][c/newClass] \tag{1.6}$$
$$\text{Method}' = \forall c.\forall s.\text{Method}[(newClass, s)/\text{Method}(class, s)][(class, s)/\uparrow]$$
$$[(c, s)/\text{RenameReferences}(\text{Method}(c, s), class, newClass)] \tag{1.7}$$
$$\text{Senders}' = \forall s.\text{Senders}[(newClass, s)/\text{Senders}(class, s)][(class, s)/\uparrow] \tag{1.8}$$
$$\text{InstanceVariablesDefinedBy}' = \text{InstanceVariablesDefinedBy}[newClass/$$
$$\text{InstanceVariablesDefinedBy}(class)][class/\uparrow] \tag{1.9}$$

**MoveMethod**

The MoveMethod refactoring relocate an existing method from an originating class to a target class and adjust the method correspondingly. Fowler et al. deem this transformation "the bread and butter of refactoring" [FBB+99, p. 142]. This is supported by the many variants of the MoveMethod refactoring in literature [Opd92, Rob99, FBB+99, Eet07, Wer99]: some definitions retain a forwarding method in the originating class, some pass an instance of the originating class as additional parameter to the moved method, some only permit moving to the classes associated via a field in the originating class. Moreover, to account for dependencies among methods and instance variables of the originating class, the refactoring can be extended to move several methods and instance variables at once.

In fact, one can argue, that MoveMethod denotes a family of closely-related refactorings, each having slightly different semantics. Such refactorings differ primarily in the actions they take to make the movied method fit into the target class. For example, if a local variable of the target class was passed as a parameter to the method before the refactoring, the parameter can be removed from the signature of the movied method. As its additional parameter, the refactoring may accept a new method name, which is more appropriate in the target class. Moreover, the moved method may or may not use its originating class. In case the method needs to access a variable of the originating class, the variable can be passed as an additional parameter to the moved method. In case the moved method needs to access a number of members of the source class, an instance of the originating class is passed as additional method parameter, or a reference from the target class to the originating class is created (unless all required members are moved to the target class as well).

To apply our discussion to most such refactorings, we define and use a general version of MoveMethod capturing the intrinsic property of the refactoring: moving a method from one class to another. While based on the definition of Roberts [Rob99, p. 109], our definition abstracts the relation between the originating and the target classes. Moreover, it does not consider additional refactoring actions, such as method renaming, creating forwarding methods, or adding or removing method parameters. More specialized definitions of MoveMethod should be seen as refinements of this general definition.

MoveMethod($class$, $selector$, $newClass$)

$$pre : \text{IsClass}(class) \wedge \tag{1.10}$$
$$\text{DefinesSelector}(class, selector) \wedge \tag{1.11}$$
$$\neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \wedge \tag{1.12}$$
$$\text{IsClass}(newClass) \wedge \tag{1.13}$$
$$\neg\text{UnderstandsSelector}(newClass, selector) \wedge \tag{1.14}$$
$$\text{UnboundVariables}(class, selector) = \emptyset \tag{1.15}$$

---

$$post : \text{Method}' = \text{Method}[(newClass, selector)/\text{Method}(class, selector)]$$
$$[(class, selector)/\uparrow] \tag{1.16}$$
$$\text{Senders}' = \text{Senders}[(newClass, selector)/\text{Senders}(class, selector)]$$
$$[(class, selector)/\uparrow] \tag{1.17}$$
$$\text{ClassReferences}' = \forall c \in \{c | \text{IsClass}(c) \wedge (class, selector) \in \text{ClassReferences}(c)\}.$$
$$\text{ClassReferences}[c/\text{ClassReferences}(c) \setminus$$
$$\{(class, selector)\} \cup \{(newClass, selector)\}] \tag{1.18}$$

## 4.2 Refactoring-Based Adapter Derivation: Informal Introduction

In this section, we informally describe how we derive adapters basing on the information about application-breaking API refactorings.

Listing 4.1: *Node*'s adapter specification

```
1 IsClass(Node) = true
2 Superclass(Node) = Object
3 InstanceVariablesDefinedBy(Node) = {adaptee}
4 ClassOf(Node, adaptee) = refactoredAPI.Node
5 Method(Node, getIdentifier) = "return refactoredAPI.Node.getID()"
6 ReferencesToInstanceVariable(Node, adaptee) = {Node.getIdentifier}
7 ClassReferences(Node) = {}
8 Senders(Node, getIdentifier) = {}
```

### 4.2.1 Adapters as Abstract Specifications and Executable Code

As any program, an adapter can be described directly by executable code (e.g., binaries) or by an abstract specification used to derive the program's executable code (e.g., source code transformed to binary code by a compiler). In our context, the advantage of representing adapters as abstract specifications is abstracting over language details relevant only for a particular implementation platform. Specifically, by using abstract specifications of adapters we can simultaneously derive black-box adapters and white-box adapters, as we discuss in the following.

Recalling our discussion from Section 3.2, in exhaustive API adaptation we derive, for each possibly refactored API type, both its black-box and white-box adapters in the static adaptation layer. For that, we abstract over both kinds of adapters and talk about an adapter specification as an abstract description defined in terms of analysis functions and their values (as introduced in Section 4.1.3), expressing properties of the adapter as a program. Such abstract description can be derived using the information about API refactorings and then employed to generate executable black-box and white-box adapters as source or binary code artifacts (e.g., in Java).

For an example of an adapter specification, let us reconsider the API class *Node* as present in the first framework version of our RefactoringLAN (Appendix A). For simplicity, let us assume that *Node* is a concrete class that has exactly one method *getIdentifier()* returning the unique node identifier. The method is renamed from *getIdentifier()* to *getID()* in the second framework release. Listing 4.1 shows the description of the corresponding adapter in terms of Roberts' analysis functions and their return values. Lines 1–2 of the description specify the names of the adapter class and its superclass (in our case, Java *Object*). Lines 3–6 relate the adapter with the refactored API class *refactoredAPI.Node* via the *adaptee* field of type *refactoredAPI.Node* defined in *Node* and referred to in the adapter method *getIdentifier()*. In particular, the semantics of *Node*'s *getIdentifier()* are specified in terms of the *getID()* method of the *refactoredAPI.Node* class (line 5). Finally, the description says, that there are no senders to the adapter and its method *getIdentifier()* method (lines 7–8). Since we cannot analyze plugins, the last two lines of the specification mean no senders to *getIdentifier()* in the framework, and no *known* senders in plugins.

This abstract description captures all information we need to create *Node*'s black-box and white-box class adapters. That is, we know precisely the syntactic part of the adapter (i.e., its name and method signatures), its relation to the latest framework API, and its supported functionality. Given such description, we consider the particularities of implementing the subtype and use relations in black-box adapters and white-box adapters (discussed in Section 3.1.5) and generate the corresponding executable Java binary adapters. For the adapter specification of Listing 4.1, the source code of the generated black-box and white-box class adapters is shown in Figure 4.3 and Figure 4.4, respectively (with adapter constructors omitted for simplicity).

To summarize, by this abstraction of adapter specifications we shift a set of implementation decisions to the time of generating adapters as executable code, and can focus on how to correctly and automatically derive adapter specifications, as discussed in the following Section 4.2.2.

### 4.2.2 Weak Refactoring Inverses Executed on Adapters: Comebacks

While in the previous section we discussed advantages of abstractly specifying adapters via analysis functions and their values, in this section we informally describe, how we automatically derive abstract

Reconstructed API Class

```
public class Node extends Object {
  refactoredAPI.Node adaptee;

  //forwarding to the framework
  public int getIdentifier() {
    return adaptee.getID();
  }
}
```

Figure 4.3: Black-box class adapter for *Node*. The adapter reconstructs the old API class *Node* and uses *refactoredAPI.Node* as its adaptee.

Reconstructed API Class

```
public class Node extends Object {
  CovertNode adaptee;

  //forwarding to the framework
  public int getIdentifier() {
    return adaptee.upcall_getID();
  }
}
```

Covert Adapter Class

```
public class CovertNode extends refactoredAPI.Node {
  Node adaptee;

  //upcalling method
  public int upcall_getID() {
    super.getID();
  }

  //multiple method dispatch
  public int getID() {
     if (adaptee.getClass().declares("int getIdentifier()"))
      return adaptee.getIdentifier();
    else
      return super.getID();
  }
}
```

Figure 4.4: White-box class adapter for *Node*. The adapter reconstructs the old API class *Node* and realizes delegation by method exchange between the reconstructed *Node* and the covert adapter *CovertNode*.

adapter specifications. Given a trace of application-breaking API refactorings, which occurred between a previous framework version and the upgraded framework, and which are sorted according to the temporal order of their occurrence in the API, we effectively undo all syntactic changes in the trace, as seen from the point of view of the plugins of that previous version. To implement the undo, for an application-breaking API refactoring we specify its weak refactoring inverse—a program transformation that reestablishes all syntactic program properties affected by the API refactoring. In effect, a weak refactoring inverse rolls back its corresponding refactoring only with regard to syntactic program properties (therefore, it is a "weak" inverse transformation as opposed to a "strong" real inverse transformation that considers all affected program properties).[5]

The main reason for introducing weak inverses, and not using real inverses, is that we do not want to invert API refactorings directly on the refactored framework API, because we want plugins to use the refactored, improved framework version. Instead, we want to invert on adapters, the structure of which is different from the structure of API types. In general, due to the structural differences between the API types and the adapters, real refactoring inverses would not be applicable on adapters (i.e., their preconditions would not be satisfied by the adapters to be transformed). Moreover, not all weak refactoring inverses are applicable on adapters. For example, although one can specify a weak refactoring inverse for a refactored public API field, this weak refactoring inverse is not applicable on adapters, because the latter do not support public fields. To denote a special kind of weak refactoring inverses that are applicable on adapters and can be used to automatically derive adapters, we call them *comebacks*.

Technically, a comeback is realized in terms of refactoring operators executed on adapters. Similar to ordinary refactorings, it is defined as a template transformation, parameterized by, for instance, type and

[5]The set of all possible weak refactoring inverses is a superset of the set of strong inverses.

method names, parameter and return value types, or default argument values. However, a comeback is parameterized by the names and types of *adapters*, and is instantiated to an executable transformation by reusing parameters of its corresponding API refactoring. Since we operate on the level of adapter specifications, comebacks can be seen as specification transformations, although we could implement them also as source or binary transformations (to derive concrete adapters directly, and not their abstract specifications). In the following, by a comeback transforming an adapter we will mean transforming an adapter specification, which will effectively impact the structure of resulting (black-box and white-box) binary adapters.

For some refactorings, the corresponding comebacks are implemented by a single refactoring. For example, the comeback corresponding to the API refactoring `RenameClass`(*oldName*, *newName*) renames the adapter from *newName* to *oldName*. As another example, the comeback of `AddEmptyClass` is defined by `RemoveEmptyClass`, removing an adapter. Similarly, the comeback of `AddMethod` is defined by `RemoveMethod`, removing an adapter method. The latter two comebacks lead to newly introduced API types and API methods not being exposed (via adapters) to old plugins, which is required to protect from Unintended Functionality and Type Capture (as we discussed for the former in Section 3.1.3 and for the latter in Section 2.2.4).

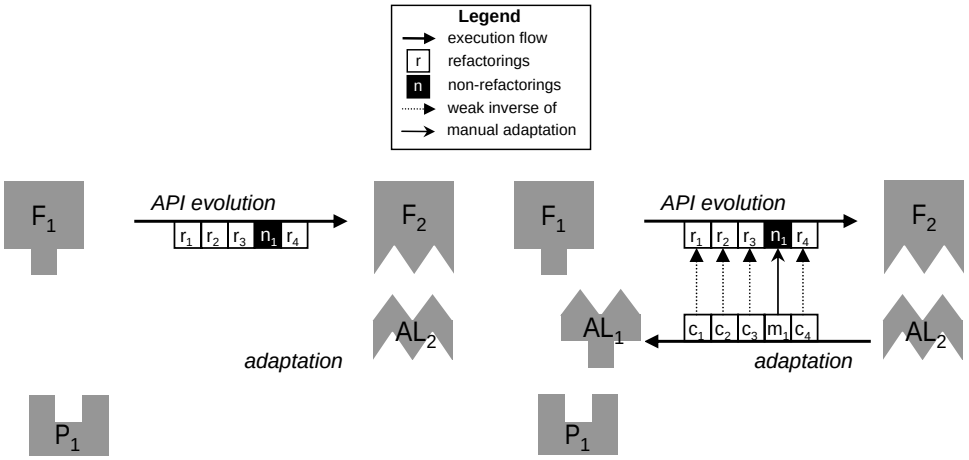
The comebacks of other refactorings consist of sequences of refactorings. For instance, the comeback of `PushDownMethod` is defined by the `DeleteMethod` and `AddMethod` refactorings, the sequential execution of which effectively moves (pushes up) the method between adapters in the adapter class hierarchy. Moreover, complex comebacks may be defined by composing other, more primitive comebacks. For example, the comeback of `ExtractSubclass` is defined by combining the comebacks of `PushDownMethod` and `AddClass`. This definition directly depends on how a complex refactoring is modeled in terms of more primitive refactorings (as we discussed in Section 3.3.4 on page 96).

Given the framework of the latest version, the version of an old plugin, and the history of API refactorings between the two versions sorted in the temporal order of their application to the API, we perform the following two adaptation steps to semi-automatically construct adapters before upgrading the framework (Figure 4.5):

1. As the first step of adapter generation, we automatically create a *syntactic replica* of the latest framework API—an adaptation layer that mimics the API of the latest framework version and uses that version's API types as adaptees (Figure 4.5(a)). At this stage, the adaptation layer corresponds to the latest API of the framework (i.e., represents exactly the same set of API types), and is not yet useful for the old plugin.

2. As the second step of adapter generation, we inverse the syntactic changes introduced by application-breaking API refactorings by executing the corresponding comebacks on the adapters of the adaptation layer (Figure 4.5(b)). The execution of comebacks yields the adaptation layer that reconstructs the API of the framework version, with which the old plugin in question was developed, and translates between the old plugin and the upgraded framework. API changes other than refactorings are addressed manually (as indicated in Figure 4.5) or using other adaptation technologies (e.g., protocol adaptation [YS97], as we will discuss in Section 5.2), by inserting the corresponding adaptation code into adapters of the generated adaptation layer.

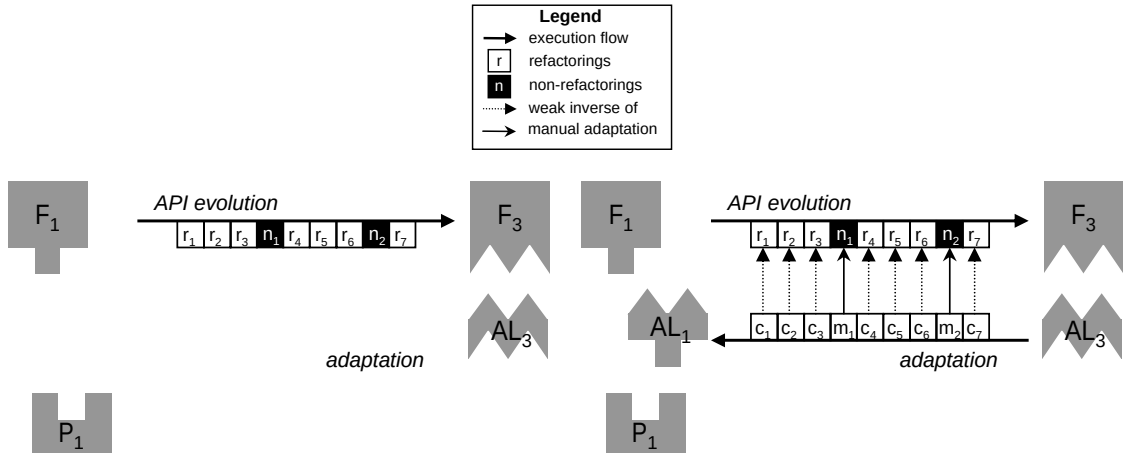
When deriving adapters, the versions of the plugin in question and of the framework need not be consecutive: what is important is the difference between the framework's API versions expressed in terms of API refactorings. For example, if the framework API is again refactored and the framework upgraded to version $F_3$, for the plugin $P_1$ the corresponding adaptation layer $AL_1$ is regenerated using the refactoring history between $F_1$ and $F_3$ (Figure 4.6). For the plugins developed with the framework version $F_2$, the corresponding adaptation layer $AL_2$ is derived similarly.




(a) First step. Adaptation layer $AL_2$ is generated as a syntactic replica of the latest framework API of version $F_2$.

(b) Second step. The execution of comebacks on adapters yields the adaptation layer $AL_1$.

Figure 4.5: Comeback-based adapter derivation (first framework upgrade). To a set of refactorings ($r_1$–$r_4$) between two framework versions ($F_1$, $F_2$) correspond comebacks ($c_1$–$c_4$) executed on the adapters of the adaptation layer $AL_2$. The execution order of comebacks is the inverse of the order of the original API refactorings in the history. Changes other than refactorings ($n_1$) are adapted manually (transformation $m_1$). The resulting adaptation layer $AL_1$ reconciles the upgraded framework and the old plugin $P_1$.




(a) First step. Adaptation layer $AL_3$ is generated as a syntactic replica of the latest framework API of version $F_3$.

(b) Second step. The execution of comebacks yields the adaptation layer $AL_1$.

Figure 4.6: Comeback-based adapter derivation (second framework upgrade). Considering the history of the application-breaking API refactorings ($r_1$–$r_7$) that occurred between the framework versions $F_1$ and $F_3$, the adaptation layer $AL_1$ translating between the upgraded framework and the old plugin $P_1$ is regenerated by the execution of the corresponding comebacks ($c_1$–$c_7$).

## 4.3 Formalizing Refactoring-Based Adapter Derivation

While in the previous section we informally introduced the concepts of a weak refactoring inverses and a comeback, in this section we formalize these concepts. To ensure the soundness of our adaptation, we furthermore show that the introduction of adapters does not change the semantics of the upgraded framework. We conclude with example proofs of weak refactoring inverses and comebacks for two

commonly used refactorings (`RenameClass` and `MoveMethod`), while leaving such proofs for a set of other common refactorings for Appendix D.

### 4.3.1 Formalizing Weak Refactoring Inverses

Ever since refactoring has become a core technique of various evolutionary and agile software development processes, such as Rational Unified Process (RUP) [Kru03] or eXtreme Programming (XP) [BA04], tool support has been growing continuously. Most of the available tools are interactive: the developer applies transformations while exploring program sources. Sometimes there is a need to *undo* a successfully applied refactoring. Intuitively, to undo a behavior-preserving transformation (a refactoring) means to execute its inverse transformation (that is then a refactoring, too). However, as Roberts points out [Rob99, p. 56], "[an] inverse [...] may be difficult to implement as a refactoring." This stems from the fact that in practice refactoring preconditions are often formalized, while the actual transformations are not. Thus, the behavior of the transformations are specified indirectly via the preconditions. Furthermore, on large programs it is often expensive to evaluate preconditions [Rob99].

However, with regard to the costs of evaluating preconditions, an important consequence of weak refactoring inverses (that invert only syntactic changes caused by refactorings and disregard other program properties) is the reduced complexity of proving the inversion property of the transformation. Furthermore, once a transformation is proved to be a weak inverse of a refactoring, its precondition does not need to be evaluated at run time.

**Definition 5** (Weak Refactoring Inverse)
Given a refactoring $\mathtt{R} = (pre_{\mathtt{R}}, trans_{\mathtt{R}}, post_{\mathtt{R}})$ that is legal on program $P$ (i.e., $P \models pre_{\mathtt{R}}$), a weak inverse refactoring of $\mathtt{R}$ is a refactoring $\mathtt{C} := (pre_{\mathtt{C}}, trans_{\mathtt{C}}, post_{\mathtt{C}})$, such that $post_{\mathtt{R}}(P) \models pre_{\mathtt{C}}$ and $post_{\mathtt{C}}(post_{\mathtt{R}}(P)) \models pre_{\mathtt{R}}$. Note that $post_{\mathtt{R}}(P)$ is the changed program $P$ after the application of the assertion transformations specified in $post_{\mathtt{R}}$, while $post_{\mathtt{C}}(post_{\mathtt{R}}(P))$ is the changed program $P$ after the application of the assertion transformations specified in $post_{\mathtt{R}}$ and then in $post_{\mathtt{C}}$, respectively.

From Definition 5 follows that to construct a proof of a weak refactoring inverse, one needs to evaluate the effect of an assertion transformation *post*. For that, we use the notion of the *strongest postcondition*. In his seminal article, Hoare [Hoa69] presents a calculus for proving program correctness using preconditions and postconditions written as a triple $\{pre\}\,\mathtt{S}\,\{Q\}$, where $\mathtt{S}$ is a program and *pre* and $Q$ are predicate sets.[6] The calculus expresses the idea that, starting in a state where *pre* is true, the execution of $\mathtt{S}$ will terminate in a state where $Q$ is true. For deriving sensible values for *pre* and $Q$ Hoare introduced two other concepts:

**Definition 6** (Strongest Postcondition)
Given a Hoare triple $\{pre\}\,\mathtt{S}\,\{Q\}$, then the following holds:

$$(\forall Q')((\{pre\}\,\mathtt{S}\,\{Q'\}) \wedge (Q \Rightarrow Q')) \Longleftrightarrow Q \text{ is the strongest postcondition.}$$

**Definition 7** (Weakest Precondition)
Given a Hoare triple $\{pre\}\,\mathtt{S}\,\{Q\}$, then the following holds:

$$(\forall pre')((\{pre'\}\,\mathtt{S}\,\{Q\}) \wedge (pre' \Rightarrow pre)) \Longleftrightarrow pre \text{ is the weakest precondition.}$$

From the definition of the implication and the interpretation of sets of formulæ it is trivial to see, that for two sets of formulæ $A$ and $B$ the following holds: $(A \supseteq B) \Longrightarrow (A \Rightarrow B)$, that is, a superset always implies a subset. Informally, the strongest postcondition could be seen as the superset of all valid postconditions, while the weakest precondition as the smallest (sensible, so not empty) subset of all valid preconditions.

The strongest postcondition $Q$ of an assertion transformation *post* (the latter defined by Definition 3 on page 106) is then a subset of the program $P$ containing exactly those assertions that were modified by the *post*. Using this notation and assuming that a refactoring $\mathtt{R}$ and its corresponding weak refactoring inverse $\mathtt{C}$ have been specified using the weakest precondition, Figure 4.7 depicts the relationship between

[6] We use different identifiers than Hoare to avoid terminology confusion with refactorings and inverses.

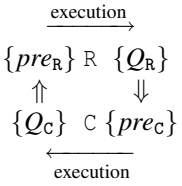



Figure 4.7: Relationship between the preconditions and postconditions of a refactoring R and its weak inverse C. Both transformations are defined as Hoare triples [Hoa69].

the preconditions and postconditions of the refactoring and its weak inverse. This can also be used for constructively proving weak refactoring inverses in accordance with Definition 5. The following steps form the basic outline for our proofs shown in this section and in Appendix D.

**Corollary 1** (Proving Weak Refactoring Inverses)
Given a refactoring $\mathtt{R} = (pre_{\mathtt{R}}, trans_{\mathtt{R}}, post_{\mathtt{R}})$ and a transformation $\mathtt{C} = (pre_{\mathtt{C}}, trans_{\mathtt{C}}, post_{\mathtt{C}})$, to prove that C is a weak inverse of R, one needs to prove the following properties of C:

1. C is behavior-preserving, that is, it is a refactoring on its own. This follows immediately from the definition of a weak refactoring inverse (Definition 5, page 116).
2. The precondition of C is satisfied by a program $P$ after the execution of R, that is, $post_{\mathtt{R}}(P) \models pre_{\mathtt{C}}$ or, equivalently, $Q_{\mathtt{R}} \Rightarrow pre_{\mathtt{C}}$.
3. After the execution of C, the program satisfies the precondition of R, that is, $post_{\mathtt{C}}(post_{\mathtt{R}}(P)) \models pre_{\mathtt{R}}$ or, equivalently, $Q_{\mathtt{C}} \Rightarrow pre_{\mathtt{R}}$.

As weak refactoring inverses are refactorings by themselves, we extend Roberts' definition of refactoring chains (Definition 4 on page 107) accordingly.

**Definition 8** (Weak Refactoring Inverse Chain)
Given a refactoring chain $\mathtt{R} = \langle \mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n \rangle$ that is legal on program $P$ (i.e., $P \models pre_{\mathtt{R}}$), the corresponding weak refactoring inverse chain is $\mathtt{C} := \langle \mathtt{C}_n, \mathtt{C}_{n-1}, \ldots, \mathtt{C}_1 \rangle$, such that for each refactoring $\mathtt{R}_i$ in R the corresponding weak inverse is $\mathtt{C}_i$ in C.

For instance, given the chain of the two refactorings ⟨RenameClass, AddMethod⟩, the corresponding weak refactoring inverse chain is ⟨RemoveMethod, RenameClass⟩. However, C is not *automatically* a weak inverse transformation of R, because (by the definition of a weak refactoring inverse) we cannot guarantee, that preceding transformations in the chain do not have side program effects invalidating (i.e., causing to evaluate to *false*) the preconditions of subsequent transformations in the chain. We will discuss this limitation in Section 4.3.4.

### 4.3.2 Formalizing the Adaptation Layer and Comebacks

Now that we formally defined the concept of a weak refactoring inverse, we can apply it in the context of creating adapters. Recalling our informal discussion from Section 4.2, in the rest of this chapter by an adapter we mean an abstract adapter specification that can be used to derive binary adapters.

**Definition 9** (Adaptation Layer Creation)
For making explicit the link between adapter names and the names of the corresponding adaptees, we introduce a new binary predicate *IsAdapter*:

**IsAdapter(*class, adapter*)** True if *adapter* is the name of the adapter for *class*.

It is modified during the first step of adaptation layer creation, when framework classes are renamed (using a predefined prefix) by the `RenameFrameworkClass` refactoring to make room for adapters

$$\texttt{RenameFrameworkClass}(class) := \texttt{RenameClass}(class, \text{prefix} + class),$$

which extends the `RenameClass` refactoring (the formal definition of which is provided in Section D.3) with the additional assertion transformation

$$\text{IsAdapter}' = \text{IsAdapter}[(newClass, class)/true].$$

The names *newClass* and *class* refer to the formal parameters of the `RenameClass` refactoring and will be bound to the concrete arguments prefix + *class* and *class*, respectively.

This conceptual renaming is necessary for introducing name spaces to avoid collisions of the names of API types, plugin types, and adapter types. By renaming, we create two name spaces: (1) the name space of the refactored framework, and (2) the name space of the reconstructed API types and old plugin types. In thesis examples (e.g., in code listings of Figure 4.3 and Figure 4.4 presenting the adapters for *Node* in Section 4.2.1), as the prefix we use the *refactoredAPI.* string.[7]

After the renaming, the second step is executing the `CreateAdapter` refactoring chain, in which we use Roberts' notation of refactoring chains (Definition 4 on page 107):[8]

$$\begin{aligned}&\texttt{CreateAdapter}(class, adapter) := \langle \\ &\quad \exists s : \text{IsAdapter}(\text{Superclass}(class), s).\texttt{AddEmptyClass}(adapter, s, \emptyset), \\ &\quad \texttt{AddInstanceVariable}(adapter, \text{“adaptee”}, class), \\ &\quad \forall m \in \{m | \text{Method}(class, m)\downarrow\}.\texttt{AddAdapterMethod}(adapter, m)\rangle.\end{aligned}$$

The `AddAdapterMethod` refactoring uses the `AddMethod` refactoring (the formal definition of which is provided in Section D.2) to create a method that forwards calls to the adapter's adaptee, using the instance variable "adaptee".

Bringing everything together, the adaptation layer for a program (i.e., the framework API, in our case) is created using the following refactoring chain:

$$\begin{aligned}&\texttt{CreateAdapterLayer}(program) := \langle \\ &\quad \forall class \in program.\texttt{RenameFrameworkClass}(class), \\ &\quad \forall (class, adapter) \in \{(c,a) | \text{IsAdapter}(c,a)\}.\texttt{CreateAdapter}(class, adapter)\rangle.\end{aligned}$$

**Corollary 2** (Adaptation Layer Properties)
From the above definition of the adaptation layer creation that transforms a program $P$ into an adapted program $A$ one can deduce the following properties (with interpretations $I_P$ and $I_A$ having $P$ and $A$ as their domains of discourse, respectively):

$$\begin{aligned}\forall class.\, I_P \models \text{IsClass}(class) &\Rightarrow I_A \models \text{IsClass}(class), \\ \forall super, sub.\, I_P \models \text{Superclass}(super) = sub &\Rightarrow I_A \models \text{Superclass}(super) = sub, \\ \forall class, sel.\, I_P \models \text{DefinesSelector}(class, sel) &\Rightarrow I_A \models \text{DefinesSelector}(class, sel), \\ \forall class.\, I_P \models \text{IsClass}(class) &\Rightarrow I_A \models \text{ClassReferences}(class) = \emptyset, \text{ and} \\ \forall class, sel.\, I_A \models \text{Senders}(class, sel) = \emptyset.&\end{aligned}$$

[7] In practice, as we discussed at the end of Section 2.2.4, in Java the prefix can be implemented using a class loader for loading framework types distinct from the class loader for loading plugin and reconstructed adapter types. As a consequence, no actual renaming of framework API types is needed.

[8] Formal definitions of the `AddEmptyClass` and `AddInstanceVariable` refactorings are provided in the appendices D.1 and E.1, respectively.

In other words, the adaptation layer first renames all API types[9] and then recreates the program API by putting adapters in the place of program classes. The only exception are instance variables, which are not replicated. In practice, however, all instance variables should be encapsulated by accessor methods to implement information hiding as proposed by Parnas [Par72]. If we assume that the program under adaptation is a valid program (i.e., accepted by the compiler of the programming language), then it holds that it does not contain any references to the newly created adaptation layer, because (1) the adaptation layer did not exist at compilation time, and (2) the process of creating the adaptation layer does not insert any references into the program classes. For the same reason, no method in the adaptation layer has any call sites. As the program API is fully recreated, the following structural derived analysis functions behave equally on the adaptation layer as they behaved on the API types before the creation of the adaptation layer: *Subclasses*, *Subclasses*$^*$, *Superclass*$^*$, *UnderstandsSelector*, and *MostSpecificSuperclass*.

The interpretation of the formulæ variables is different in $I_P$ and $I_A$: while a class or method name in $I_P$ references an artifact in the program, in $I_A$ the same name points into the adaptation layer. Informally, $I_P$ is like a snapshot of the program before the adaptation layer creation, while $I_A$ captures the situation afterwards.

After having examined the syntactic part of the adaptation layer generation process, we need to show that adding the adaptation layer is behavior-preserving, that is, the adapter methods forwarding to the actual program methods do not change the semantics of the latter. Informally, this behavior preservation is implied by the construction of the adaptation layer: since the definition of `CreateAdapterLayer` is expressed solely in terms of refactorings, `CreateAdapterLayer` is also a refactoring, hence, behavior-preserving. Formally, we prove behavior preservation using Java denotational semantics [AFL99]. Since our proof requires presenting a fairly lengthy background on Java denotational semantics, while the proof is not required in the rest of this chapter, we enclose it in Appendix C.

Now that the derivation of the adaptation layer has been detailed, weak refactoring inverses can be executed on adapters of the adaptation layer instead of the actual program (i.e., the framework, in our case). This shift permits us to transform the adaptation layer and leave the actual program untouched.

**Definition 10** (Comeback)
Given a program $P$ modified by a refactoring `R` and an adaptation layer $A$ derived for $P$, a comeback for `R` is a weak refactoring inverse $\texttt{C} := (pre, trans, post)$ that can be executed on $A$, whenever it can be executed on $P$ itself:

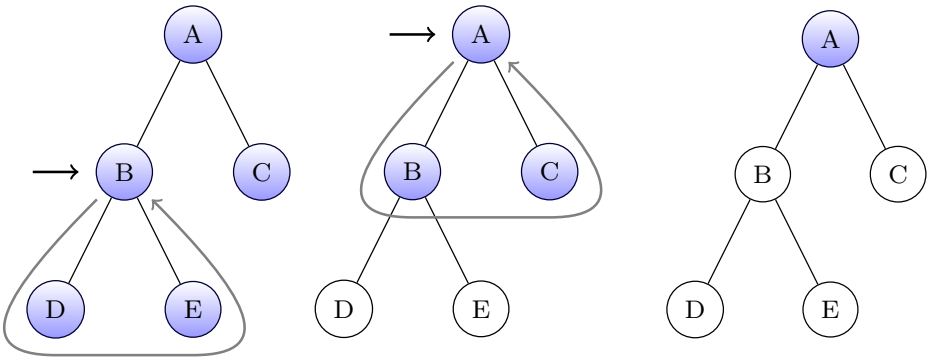
$$P \models pre \wedge A \models pre, \text{ where } A = post_{\texttt{CreateAdapterLayer}}(P).$$

### 4.3.3 Pulling Up Adaptee Fields to Avoid Split States

As discussed in Section 3.3.1 (Figure 3.37 on page 88), to avoid split states in the presence of adapters we must make sure that a wrapper always uses the same adaptee object. Thereby we ensure that, when the message forwarding is performed in an adapter method, the self reference is always bound to the same message receiver (i.e., an adaptee object). Our adopted solution sketched in Section 3.3.1 is to declare the adaptee field protected and permit adapters to inherit it. Only the top-most adapter in the hierarchy of reconstructed API classes defines the adaptee field, which is inherited by the adapters down the hierarchy. In this section, we show how we realized this design decision.

This solution is the final step in adapter generation (i.e., after all corresponding comebacks are executed), realized by refactorings that pull up adaptee fields in the adapter hierarchy. Namely, the `Recursively-PullUpInstanceVariable` refactoring makes recursive use of the `PullUpInstanceVariable` refactoring (the formal definition of which is provided in Appendix E.2) to remove redundant adaptee fields from

[9] Recalling our discussion from Section 3.2, in our adaptation we consider only user-defined API types—types that could be refactored by framework developers. Library types, such as language types and component types of third parties, are considered to remain unchanged between API releases. For such types, no adapters are created.




(a) Recursion from adapter B. (b) Recursion from adapter A. (c) Iteration completed.

Figure 4.8: Termination of the iterative application of the `RecursivelyPullUpInstanceVariable` refactoring when creating the adaptation layer. Colored and transparent nodes stand for adapters with and without an adaptee field, respectively. The straight arrow points to the start node of the current iteration step.

subclasses, so that the one inherited from the base adapter is used instead:

$$\begin{aligned}
&\texttt{RecursivelyPullUpInstanceVariable}(\mathit{class}, \mathit{varName}) := \langle \\
&\quad \forall c \in \{c | c \in \text{Subclasses}(\mathit{class}) \wedge \mathit{varName} \in \text{InstanceVariablesDefinedBy}(c)\}. \\
&\qquad \texttt{RecursivelyPullUpInstanceVariable}(c, \mathit{varName}), \\
&\quad \forall c \in \{c | c \in \text{Subclasses}(\mathit{class}) \wedge \mathit{varName} \in \text{InstanceVariablesDefinedBy}(c)\}. \\
&\qquad \texttt{PullUpInstanceVariable}(\{c\}, \mathit{varName})\rangle
\end{aligned}$$

This refactoring is applied to all adapter classes in the adaptation layer with "adaptee" as an actual value for *varName*.

The `RecursivelyPullUpInstanceVariable` refactoring is head-recursive (i.e., it performs a postorder traversal): it first descends the class hierarchy and pulls up instance variables starting from the leaves of the adapter tree working its way up to the topmost adapter. This is necessary, because `PullUpInstanceVariable` is only defined for two directly connected hierarchy levels (i.e., an adapter class and its subclasses). Pulling up instance variables with the same name from several subclasses to a single superclass works, because the precondition of the `PullUpInstanceVariable` refactoring does not check for an existing instance variable in the superclass, and it uses set union, which will eliminate duplicates. Due to the use of the `MostSpecificSuperclass` analysis function, the final type of the adaptee field in the base adapter will be the most specific supertype of all adaptee fields, so that instances of all subclasses can be assigned to the adaptee field.

This recursive algorithm terminates for two reasons: First, it is applied to a finite set of adapters, because the set of program classes is finite (Definition 2 on page 105) and for every program class exactly one adapter is created. Second, each adapter only has a finite set of subclasses. An exemplary application of the algorithm is depicted in Figure 4.8, where the iteration starts with adapter B and pulls up the adaptee fields from the subclasses D and E. The iteration continues with adapter A and the recursion stops at the subclasses B and C, from which the adaptee fields are pulled up in turn. The following iteration steps over C, D, and E have no effect, as no recursion can be performed due to the absence of adaptee fields in the subclasses.

The use of the universal quantifications in the definition of `RecursivelyPullUpInstanceVariable` does not only serve the obvious purpose of iterating over all classes fulfilling certain properties. In contrast to the existential quantifier, a universal quantification is successful in case there are no items to quantify, because the definition of its evaluation under an interpretation includes the empty set. As a consequence, `RecursivelyPullUpInstanceVariable` will be successful, even if applied to a single adapter without subclasses.

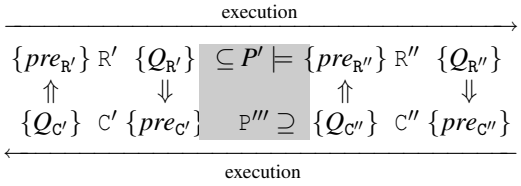



Figure 4.9: No automatic proof for chaining weak refactoring inverses. Whereas for the refactorings R′ and R″ their automating chaining is possible (the upper part of the grey box), for the corresponding weak refactoring inverses C′ and C″ the feasibility of chaining has to be proved manually (the lower part of the grey box).

### 4.3.4 Limitation: No Automatic Proof of Chaining Weak Refactoring Inverses

The formalism introduced by Roberts, on which we base the formalization of comebacks, permits chaining of refactorings (Definition 4 on page 107). Similarly, because they are also refactorings, weak inverses can be chained (Definition 8 on page 117). For instance, given the chain of the two refactorings `RenameClass` and `AddMethod`, the corresponding weak refactoring inverse chain is ⟨`RemoveMethod`, `RenameClass`⟩. Although intuitive, this whole chain cannot be automatically proved to be a weak inverse transformation for the chain of the two refactorings. To automatically prove the chaining, one needs to automatically prove that the `RenameClass` refactoring as the last element of the inverse chain can be executed after the `RemoveMethod` refactoring.

Unfortunately, there is no way to provide a general proof that is independent from the actual transformations, because our comeback formalism is built on assertions and subsumption rather than on descriptions of the actual program transformations. This choice was intended, it simplified reasoning dramatically and allowed to extensively reuse existing refactoring formalization defined by Opdyke [Opd92] and Roberts [Rob99]. However, the genericity of the definition of weak refactoring inverses permits them to have side effects. As a consequence, for each given chain of weak refactoring inverses we need to prove manually, whether the transformations can be executed in the specified sequence so that it qualifies as a chain of weak refactoring inverses.

Figure 4.9 details this limitation: it shows two refactorings R′ and R″ executed in a chain together with their corresponding weak inverses C′ and C″. The upper part of the grey area in the middle of the figure is given by Roberts' definition of refactoring chains (Definition 4, page 107). To prove that C′ can be executed, which corresponds to the missing $=\!|$ symbol in the lower half of the grey area, either of the following two relationships needs to be shown: $P'$ (the program after the execution of R′) is equivalent to $P'''$ (the program after the execution of C″), or $P'''$ enables the execution of C′. The former approach requires formal descriptions of both R″ and C″ (and not only of their preconditions and postconditions) as program transformations, and such formal descriptions are not available in the current state of the art on refactoring formalization using first-order predicate logic. Therefore, we use the latter approach for manually proving the feasibility of the chain.

### 4.3.5 Example Proofs of Weak Refactoring Inverses and Comebacks for `RenameClass` and `MoveMethod`

We conclude this section with the definitions and proofs of weak refactoring inverses and comebacks for `RenameClass` and `MoveMethod` refactorings, which formal definitions we repeat in this section for convenience. Consistently with the previous presentation, we use Roberts' notation adopted to reflect the properties of the Java programming language (as we discussed in Section 4.1.3).

### Notation and Invariants

While the logic connectives $\wedge$, $\vee$, and $\neg$ are used in FOPL formulæ, their corresponding boolean functions $\wedge^*$, $\vee^*$, and $\neg^*$ are used when interpreting them. They combine and transform the boolean values $\top$ and $\bot$ obtained by interpreting the different parts of a formula. There are more logic connectives, such as $\Rightarrow$ and $\Leftrightarrow$, which can be composed from the three primitive ones.

Interpreting a FOPL formula also requires the specification of a *variable mapping*, which assigns values to all *unbound variables* in the formula. A variable is unbound, if it is neither bound by the existential quantifier $\exists$ nor by the universal quantifier $\forall$. Such a variable mapping for an interpretation $I = (\mathcal{D}, \cdot^I)$ on a domain $\mathcal{D}$ is a function $Z : \mathcal{V} \to \mathcal{D}$, and it is usually specified by enumerating the variables and their assigned values, as in

$$Z = \{a \mapsto x, b \mapsto y\}.$$

The $*$ symbol denotes a *don't care* condition in both FOPL formulæ and truth value computations, and is used as a short-hand notation for negligible parts. For example, in the truth value computation $(\top \vee^* *)$ only the left part of the computation is important, because it determines the result.

For the sake of readability, in the following transformation definitions we will number the assertions in preconditions and assertion transformations. This eases referring to them in proofs, as, for example, in 3.3 : IsClass$[class/false]$ $[newClass/true]$ referring to the assertion transformation part numbered 3.3. These references will be used to indicate the effect of an assertion transformation of an interpretation domain with regard to a specific assertion as follows: Consider the interpretation $I$ defined over a domain $P$ and the assertion transformation *post* that was applied to $P$ yielding $P'$. Then $I'$ defined over $P'$ can be reduced to $I^{\{3.3:\text{IsClass}[class/false][newClass/true]\}}$ whenever the predicate of interest is IsClass. In other words, by this notation we show the difference between $I$ and $I'$, where the difference is expressed in terms of certain assertions transformed by *post*.

The following invariants formalize the most general relations and always hold for every object-oriented program. They will be used throughout the proofs in the following sections.

**Invariant 1**
Only existing classes can be referenced:

$$\forall class.\text{ClassReferences}(class) \neq \emptyset \Longrightarrow \text{IsClass}(class).$$

**Invariant 2**
A method can only exist within a class:

$$\exists class.\exists method.\text{Method}(class, method)\downarrow \Longrightarrow \text{IsClass}(class).$$

**Invariant 3**
Only existing methods can be called:

$$\exists class.\exists m.\text{Senders}(class, m) \neq \emptyset \Longrightarrow \text{Method}(class, m)\downarrow.$$

**Invariant 4**
An instance variable can only exist within a class:

$$\forall class.\text{InstanceVariablesDefinedByClass}(class) \neq \emptyset \Longrightarrow \text{IsClass}(class).$$

**Invariant 5**
Only classes can have and can be superclasses:

$$\exists super.\exists sub.\text{Superclass}(sub) = super \Longrightarrow \text{IsClass}(sub) \wedge \text{IsClass}(super).$$

By definition this implies

$$\exists super.\text{IsClass}(super) \Longrightarrow \forall sub \in \text{Subclasses}(super) : \text{IsClass}(sub).$$

In addition to these global invariants, for certain refactorings we define refactoring-specific invariants extracted from the precondition. The extraction operates on sets of assertions, so assertion equality must be defined first.

**Definition 11** (Assertion Equality)
Two assertions $p$ and $q$ are equal iff they use the same input to the same predicate:

$$p(x_1,\ldots,x_m) = q(y_1,\ldots,y_n) \Leftrightarrow p = q \wedge m = n \wedge \forall (x,y) \in \{(x_1,y_1),\ldots,(x_m,y_n)\} : x = y.$$

**Definition 12** (Refactoring Invariant)
Given a refactoring $\mathtt{R} := (pre, trans, post)$, the *refactoring invariant* is the subset of the precondition *pre* containing only those assertions that are not changed by *post*, where assertions are compared according to Definition 11:

$$inv(\mathtt{R}) := pre \cap post(pre).$$

**Corollary 3** (Refactoring Chain Invariant)
Given a refactoring chain $\mathtt{R} := \langle \mathtt{R}_1, \mathtt{R}_2 \rangle$ consisting of refactorings $\mathtt{R}_1 := (pre_1, trans_1, post_1)$ and $\mathtt{R}_2 := (pre_2, trans_2, post_2)$, the refactoring chain invariant can be composed from the refactoring invariants as follows:

$$inv(\mathtt{R}) := (inv(\mathtt{R}_1) \cup pre_2) \cap post_2(inv(\mathtt{R}_1) \cup pre_2).$$

As a final notation remark, in certain cases we need to indicate that an interpretation $I'$ does not deviate from an interpretation $I$ with respect to a specific assertion due to a (global or refactoring) invariant. In other words, we need to show that with regard to a certain assertion certain program properties under $I$ and under $I'$ are identical, due to a certain invariant respected in both $I$ and $I'$. For that, for the global invariants, their number will be included as in $I^{\{\text{Inv } 1\}}$, while refactoring invariants will be referred to only by the string *inv* without a number.

#### `RenameClass`

The `RenameClass` refactoring (Section 4.1.4) renames an existing class in the program.

`RenameClass`(*class*, *newClass*)

$$pre : \text{IsClass}(class) \wedge \tag{3.1}$$
$$\neg \text{IsClass}(newClass) \tag{3.2}$$

---

$$post : \text{IsClass}' = \text{IsClass}[class/false][newClass/true] \tag{3.3}$$
$$\text{ClassReferences}' = \forall c \neq class. \forall s \in \{s | (class,s) \in \text{ClassReferences}(c)\}.$$
$$\text{ClassReferences}[class/\uparrow][newClass/\text{ClassReferences}(class)]$$
$$[c/\text{ClassReferences}(c) \setminus \{(class,s)\} \cup \{(newClass,s)\}] \tag{3.4}$$
$$\text{ReferencesToInstanceVariable}' = \forall (c,v,s) \in$$
$$\{(c,v,s) | (class,s) \in \text{ReferencesToInstanceVariable}(c,v) \wedge c \neq class\}.$$
$$\forall lv. \forall lr \in \{lr | (class,lr) \in \text{ReferencesToInstanceVariable}(class,lv)\}.$$
$$\text{ReferencesToInstanceVariable}[(class,lv)/\uparrow][(newClass,lv)/$$
$$\text{ReferencesToInstanceVariable}(class,lv) \setminus \{(class,lr)\} \cup \{(newClass,lr)\}]$$
$$[(c,v)/\text{ReferencesToInstanceVariable}(c,v) \setminus \{(class,s)\} \cup \{(newClass,s)\}] \tag{3.5}$$
$$\text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[class/\uparrow]$$
$$[newClass/\text{Superclass}(class)][c/newClass] \tag{3.6}$$
$$\text{Method}' = \forall c. \forall s. \text{Method}[(newClass,s)/\text{Method}(class,s)][(class,s)/\uparrow]$$
$$[(c,s)/\text{RenameReferences}(\text{Method}(c,s), class, newClass)] \tag{3.7}$$
$$\text{Senders}' = \forall s. \text{Senders}[(newClass,s)/\text{Senders}(class,s)][(class,s)/\uparrow] \tag{3.8}$$
$$\text{InstanceVariablesDefinedBy}' = \text{InstanceVariablesDefinedBy}[newClass/$$
$$\text{InstanceVariablesDefinedBy}(class)][class/\uparrow] \tag{3.9}$$

**Theorem 1** (`InvRenameClass` I)
The weak refactoring inverse for the `RenameClass` refactoring, `InvRenameClass`, is the `RenameClass` refactoring:

$$\texttt{InvRenameClass}(\mathit{class}, \mathit{newClass}) := \texttt{RenameClass}(\mathit{newClass}, \mathit{class}).$$

*Proof.*

1. `InvRenameClass` is behavior-preserving by definition, because it consists of the single refactoring `RenameClass`.

2. $\mathit{post}_{\texttt{RenameClass}}(P) \models \mathit{pre}_{\texttt{InvRenameClass}} \Leftrightarrow \mathit{post}_{\texttt{RenameClass}}(P) \models \mathit{pre}_{\texttt{RenameClass}}$:
   Let $I'$ be the interpretation gained by applying the `RenameClass` refactoring to the source code $S$ of the program $P$ reflected by $P' = \mathit{post}_{\texttt{RenameClass}}(P)$

   $$I' = \left(\mathit{trans}_{\texttt{RenameClass}}(S), \cdot^{I}\right),$$

   and let $Z$ be the variable assignment induced by theorem 1

   $$Z = \{\mathit{class} \mapsto \mathit{newClass}, \mathit{newClass} \mapsto \mathit{class}\}.$$

   Then $\mathit{pre}_{\texttt{RenameClass}}$ evaluates to true under $I'$ and $Z$:

   $$\begin{aligned}
   &[\text{IsClass}(\mathit{class})]^{I',Z} \wedge^* \neg^* [\text{IsClass}(\mathit{newClass})]^{I',Z} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I'} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I'} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I\{\text{3.3:IsClass}[\mathit{newClass}/\mathit{true}]\}} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I\{\text{3.3:IsClass}[\mathit{class}/\mathit{false}]\}} \\
   &= \top \wedge^* \neg^* \bot = \top
   \end{aligned}$$

3. $\mathit{post}_{\texttt{InvRenameClass}}(\mathit{post}_{\texttt{RenameClass}}(P)) \models \mathit{pre}_{\texttt{RenameClass}}$:
   Let $I''$ be the interpretation gained by applying the `RenameClass` refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = \mathit{post}_{\texttt{RenameClass}}(P')$

   $$I'' = \left(\mathit{trans}_{\texttt{RenameClass}}(S'), \cdot^{I}\right),$$

   and let $Z'$ be the variable assignment induced by theorem 1

   $$Z' = \{\mathit{class} \mapsto \mathit{newClass}, \mathit{newClass} \mapsto \mathit{class}\}.$$

   Then $\mathit{pre}_{\texttt{RenameClass}}$ evaluates to true under $I''$ and $Z'$:

   $$\begin{aligned}
   &[\text{IsClass}(\mathit{class})]^{I'',Z'} \wedge^* \neg^* [\text{IsClass}(\mathit{newClass})]^{I'',Z'} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I''} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I''} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I'\{\text{3.3:IsClass}[\mathit{newClass}/\mathit{true}]\}} \wedge^* \\
   &\quad \neg^* [\text{IsClass}(\mathit{class})]^{I'\{\text{3.3:IsClass}[\mathit{class}/\mathit{false}]\}} \\
   &= \top \wedge^* \neg^* \bot = \top
   \end{aligned}$$

□

**Theorem 2** (`InvRenameClass` II)
`InvRenameClass` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `AddEmptyClass` refactoring precondition it holds that $I_P \models a \Longrightarrow I_A \models a$:

$$I_P \models \text{IsClass}(\mathit{class}) \overset{\mathit{Corollary}\ 2}{\Longrightarrow} I_A \models \text{IsClass}(\mathit{class})$$

$$I_P \models \neg\text{IsClass}(\mathit{newClass}) \Leftrightarrow I_P \not\models \text{IsClass}(\mathit{newClass}) \overset{\mathit{Corollary}\ 2}{\Longrightarrow} I_A \not\models \text{IsClass}(\mathit{newClass}) \Leftrightarrow I_A \models \neg\text{IsClass}(\mathit{newClass})$$

□

**MoveMethod**

The MoveMethod refactoring moves an existing method from one class to another and adjust the method to its new location. Its general formal definition was introduced in Section 4.1.4.

MoveMethod(*class*, *selector*, *newClass*)

$$
\begin{aligned}
&\mathit{pre} : \text{IsClass}(\mathit{class}) \wedge && (3.10)\\
&\quad \text{DefinesSelector}(\mathit{class}, \mathit{selector}) \wedge && (3.11)\\
&\quad \neg\text{UnderstandsSelector}(\text{Superclass}(\mathit{class}), \mathit{selector}) \wedge && (3.12)\\
&\quad \text{IsClass}(\mathit{newClass}) \wedge && (3.13)\\
&\quad \neg\text{UnderstandsSelector}(\mathit{newClass}, \mathit{selector}) \wedge && (3.14)\\
&\quad \text{UnboundVariables}(\mathit{class}, \mathit{selector}) = \emptyset && (3.15)
\end{aligned}
$$

---

$$
\begin{aligned}
&\mathit{post} : \text{Method}' = \text{Method}[(\mathit{newClass}, \mathit{selector})/\text{Method}(\mathit{class}, \mathit{selector})]\\
&\quad [(\mathit{class}, \mathit{selector})/\uparrow] && (3.16)\\
&\quad \text{Senders}' = \text{Senders}[(\mathit{newClass}, \mathit{selector})/\text{Senders}(\mathit{class}, \mathit{selector})]\\
&\quad [(\mathit{class}, \mathit{selector})/\uparrow] && (3.17)\\
&\quad \text{ClassReferences}' = \forall c \in \{c | \text{IsClass}(c) \wedge (\mathit{class}, \mathit{selector}) \in \text{ClassReferences}(c)\}.\\
&\quad \text{ClassReferences}[c/\text{ClassReferences}(c) \setminus\\
&\quad \{(\mathit{class}, \mathit{selector})\} \cup \{(\mathit{newClass}, \mathit{selector})\}] && (3.18)
\end{aligned}
$$

**Theorem 3** (InvMoveMethod I)
The weak refactoring inverse for the MoveMethod refactoring, InvMoveMethod, is the MoveMethod refactoring:

$$
\begin{aligned}
&\texttt{InvMoveMethod}(\mathit{class}, \mathit{selector}, \mathit{newClass}) :=\\
&\quad \texttt{MoveMethod}(\mathit{newClass}, \mathit{selector}, \mathit{class}).
\end{aligned}
$$

*Proof.*

1. InvMoveMethod is behavior-preserving by definition, because it consists of the single refactoring MoveMethod.

2. $\mathit{post}_{\texttt{MoveMethod}}(P) \models \mathit{pre}_{\texttt{InvMoveMethod}} \Leftrightarrow \mathit{post}_{\texttt{MoveMethod}}(P) \models \mathit{pre}_{\texttt{MoveMethod}}$:
   Let $I'$ be the interpretation gained by applying the MoveMethod refactoring to the source code $S$ of the program $P$ reflected by $P' = \mathit{post}_{\texttt{MoveMethod}}(P)$

   $$I' = \left(\mathit{trans}_{\texttt{MoveMethod}}(S), \cdot^{I}\right),$$

   and let $Z$ be the variable assignment induced by theorem 3

   $$Z = \{\mathit{class} \mapsto \mathit{newClass}, \mathit{selector} \mapsto \mathit{selector}, \mathit{newClass} \mapsto \mathit{class}\}.$$

   The refactoring invariant of the MoveMethod refactoring is

   $$
   \begin{aligned}
   \mathit{inv}(\texttt{MoveMethod}) \quad = \quad & \{\text{IsClass}(\mathit{class}) \wedge\\
   & \neg\text{UnderstandsSelector}(\text{Superclass}(\mathit{class}), \mathit{selector}) \wedge\\
   & \text{IsClass}(\mathit{newClass})\}.
   \end{aligned}
   $$

Then $pre_{\texttt{MoveMethod}}$ evaluates to true under $I', Z$, and the invariant:

$$
\begin{aligned}
&[\text{IsClass}(class)]^{I',Z} \wedge^* [\text{DefinesSelector}(class, selector)]^{I',Z} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I',Z} \wedge^* \\
&[\text{IsClass}(newClass)]^{I',Z} \wedge^* \neg^* [\text{UnderstandsSelector}(newClass, selector)]^{I',Z} \wedge^* \\
&[\text{UnboundVariables}(class, selector)]^{I',Z} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I'} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I'} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I'} \wedge^* \\
&[\text{IsClass}(class)]^{I'} \wedge^* \neg^* [\text{UnderstandsSelector}(class, selector)]^{I'} \wedge^* \\
&[\text{UnboundVariables}(newClass, selector)]^{I'} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I'} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I'} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I'} \wedge^* \\
&[\text{IsClass}(class)]^{I'} \wedge^* \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'} \wedge^* \\
&\neg^* [\text{DefinesSelector}(class, selector)]^{I'} \wedge^* [\text{UnboundVariables}(newClass, selector)]^{I'} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I\{\text{Inv}\}} \wedge^* \\
&[\text{Method}(newClass, selector)]^{I\{3.16:\text{Method}[(newClass,selector)/\text{Method}(class,selector)]\}} \downarrow \wedge^* \\
&\top \wedge^* [\text{IsClass}(class)]^{I\{\text{Inv}\}} \wedge^* \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I\{\text{Inv}\}} \wedge^* \\
&[\text{Method}(class, selector)]^{I\{3.16:\text{Method}[(class,selector)/\uparrow]\}} \uparrow \wedge^* \top \\
= &\top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top = \top
\end{aligned}
$$

The last truth value for the *UnboundVariables* function can be deduced from the fact that the abstract syntax tree of the moved method is not changed and so no references to non-local variables are introduced.

3. $post_{\texttt{InvMoveMethod}}(post_{\texttt{MoveMethod}}(P)) \models pre_{\texttt{MoveMethod}}$:
Let $I''$ be the interpretation gained by applying the `MoveMethod` refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = post_{\texttt{MoveMethod}}(P')$

$$I'' = \left(trans_{\texttt{MoveMethod}}(S'), \cdot^I\right),$$

and let $Z'$ be the variable assignment induced by theorem 3

$$Z' = \{class \mapsto newClass, selector \mapsto selector, newClass \mapsto class\}.$$

Then $pre_{\texttt{MoveMethod}}$ evaluates to true under $I'', Z'$, and the invariant:

$$
\begin{aligned}
&[\text{IsClass}(class)]^{I'',Z'} \wedge^* [\text{DefinesSelector}(class, selector)]^{I'',Z'} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'',Z} \wedge^* \\
&[\text{IsClass}(newClass)]^{I'',Z'} \wedge^* \neg^* [\text{UnderstandsSelector}(newClass, selector)]^{I'',Z'} \wedge^* \\
&[\text{UnboundVariables}(class, selector)]^{I'',Z'} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I''} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I''} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I''} \wedge^* \\
&[\text{IsClass}(class)]^{I''} \wedge^* \neg^* [\text{UnderstandsSelector}(class, selector)]^{I''} \wedge^* \\
&[\text{UnboundVariables}(newClass, selector)]^{I''} = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
&= [\text{IsClass}(newClass)]^{I''} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I''} \wedge^* \\
&\quad \neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I''} \wedge^* \\
&\quad [\text{IsClass}(class)]^{I''} \wedge^* \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I''} \wedge^* \\
&\quad \neg^* [\text{DefinesSelector}(class, selector)]^{I''} \wedge^* [\text{UnboundVariables}(newClass, selector)]^{I''} = \emptyset \\
&= [\text{IsClass}(newClass)]^{I'\{\text{Inv}\}} \wedge^* \\
&\quad [\text{Method}(newClass, selector)]^{I'\{3.16:\text{Method}[(newClass,selector)/\text{Method}(class,selector)]\}} \downarrow \wedge^* \\
&\quad \top \wedge^* [\text{IsClass}(class)]^{I'\{\text{Inv}\}} \wedge^* \\
&\quad \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'\{\text{Inv}\}} \wedge^* \\
&\quad [\text{Method}(class, selector)]^{I'\{3.16:\text{Method}[(class,selector)/\uparrow]\}} \uparrow \wedge^* \top \\
&= \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top = \top
\end{aligned}
$$

Again, the last truth value is implied by not changing the method's abstract syntax tree when moving it to the new class.

□

**Theorem 4** (`InvMoveMethod` II)
`InvMoveMethod` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `MoveMethod` refactoring precondition it holds that $I_P \models a \implies I_A \models a$:

$$
\begin{aligned}
&I_P \models \text{IsClass}(class) \overset{Corollary\ 2}{\implies} I_A \models \text{IsClass}(class) \\
&I_P \models \text{DefinesSelector}(class, selector) \overset{Corollary\ 2}{\implies} I_A \models \text{DefinesSelector}(class, selector) \\
&I_P \models \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \overset{Corollary\ 2}{\implies} \\
&\quad I_A \models \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \\
&I_P \models \text{IsClass}(newClass) \overset{Corollary\ 2}{\implies} I_A \models \text{IsClass}(newClass) \\
&I_P \models \neg\text{UnderstandsSelector}(newClass, selector) \overset{Corollary\ 2}{\implies} \\
&\quad I_A \models \neg\text{UnderstandsSelector}(newClass, selector) \\
&I_P \models \text{UnboundVariables}(class, selector) = \emptyset \overset{Corollary\ 2}{\implies} \\
&\quad I_A \models \text{UnboundVariables}(class, selector) = \emptyset
\end{aligned}
$$

The last implication only holds true, if access to the adaptee field in adapters is encapsulated, i.e., performed via a method call. If it were accessed directly, the adaptee field would be in the set of unbound variables of the adapter method. □

## 4.4 Summary

In this chapter, we argue for a need of rigorously defining adaptation actions that compensate for application-breaking API refactorings. There are at least two immediate benefits of such rigorous definition. First, it fosters better understanding of how to define compensating actions and reasoning about the intended meaning of these actions. Second, arguably even more important, it helps to automate the adaptation and avoid hand-coded, platform-specific implementations, which are tedious and error-prone to provide.

Our definitions of weak refactoring inverses and comebacks are an important step towards a systematic adaptation of evolutionary, yet application-breaking, API refactorings. The weakness of the definition of a weak refactoring inverse is in fact its main power: by focusing only on certain (syntactic) program properties important for proper adaptation, for a given application-breaking API refactoring we can define its syntactic undo, while avoiding the complexity of reasoning about other program properties possibly affected by the refactoring. In our case, some of these inverses—comebacks—are executable on adapters, and can be used to systematically and automatically derive the latter. Because we stay at a formal and platform-independent level in our adaptation definitions, our formal technology can be realized in a large number of platforms and implementation languages.

It is important to emphasize, that we do not deliver proof support for all comebacks supported in our logic-based adaptation tool ComeBack! (discussed in detail in Chapter 5). While for the comebacks formally proven in Section 4.3.5 and Appendix D we translate their definitions to the tool's input format (i.e, Prolog rules), for other compensated API refactorings we implement the corresponding comebacks directly in the tool. To provide proof support for refactorings applied to such language features as interfaces, attributes, or generic types, would require formalizing the refactorings of those features first. We consider this extension a future work, which is out of the scope of this thesis. Instead, addressing a number of refactorings of important language features ubiquitous in object-oriented programming languages, we believe to pave a way to consistently constructing refactoring-based adaptation for a large number of languages. Our definitions should be seen as a guideline for developers on how to systematically define a formal refactoring-based adaptation for a given object-oriented programming language. Concrete implementations of the technology will need to add support for features specific to their target programming language.

Currently, for each considered API refactoring we define and prove its weak refactoring inverse and comeback manually. As a future, fairly ambitious work we envisage automatically deriving such specifications. Given a refactoring specification as a set of preconditions and assertion transformations, a deductive engine could derive the assertions of the precondition of the corresponding weak refactoring inverse, as well as the assertions transformed by the assertion transformation of that inverse. Moreover, the deductive engine could also consider the execution context of the inverse (e.g., adapters) to derive the specification of a weak refactoring inverse executable in the given adaptation context (e.g., a comeback specification), if such specification may be defined.

An important limitation of our work—no automatic chaining of weak refactoring inverses—can be solved by providing more rigor in the descriptions of refactorings as program transformations. Instead of reasoning about refactorings in terms of preconditions and assertion transformations, one could use a formalism, such as graph rewriting [HJE06, EJ04, Eet07] or term rewriting [Bat07], to explicitly specify the refactoring transformations and, hence, their inverses. In this case, it would be possible to formally prove the absence of undesirable side effects of refactoring inverses, enabling automatic chaining of inverses. The feasibility of this future work depends directly on the progress in the area of refactoring formalization.

*"Evolution of software components is a complex and large topic. No single tool could handle all its aspects."*

–Danny Dig
*Automated Upgrading of Component-Based Applications* [Dig07, p. 133]

# 5

# Realizing, Evaluating, and Empowering a Refactoring-Based Adaptation Technology

In the previous two chapters, we denote precisely the design of the adapters compensating for application-breaking API refactorings, exhaustive API adaptation, and formal refactoring-based adapter derivation. In practice, we should find a straightforward way to realize these adaptation decisions in a tool, which uses the information about API refactorings to provide binary backward compatibility of the upgraded framework and old plugins. In particular, the tool should be able to create binary adapters according to the Black-Box Class and Interface Adapter patterns, and the White-Box Class and Interface Adapter patterns. In addition, the tool should support all adaptation decisions of exhaustive API adaptation, such as the adapter cache and dynamic adapter creation. At the same time, there should be no conceptual gap between our formal definition of rigorous adapter derivation and its realization in the tool for automatically creating binary adapters.

A practical tool should also provide transparent framework upgrade, in the sense that the refactored framework should be deployed and combined with existing plugins without additional interventions required from application developers. For our tool, furthermore, we want to estimate, to which extent it can support adaptation of application-breaking API refactorings discovered by other researches [DJ05] and us (Section 2.2.3). In addition, we should evaluate the performance overhead implied by adapters. Finally, since a single tool cannot cope with all aspects of component evolution [Dig07], we want also to assess the feasibility of combining refactoring-based adaptation with adaptation technologies compensating for application-breaking API changes other than refactorings.

This chapter brings the following thesis contribution:

**C7: Practical tool for refactoring-based API adaptation.** We discuss the development and evaluation of our logic-based tool ComeBack! that generates Java binary adapters using a history of API refactorings and a library of comebacks. We decided for Prolog [DCED96], mainly because the first-order logic definitions of adapters as abstract specifications and comebacks as specification transformations could be straightforwardly translated into Prolog facts and rules. Internally, adapter specifications are represented by a set of Prolog facts, while comebacks are implemented as Prolog rules, which operate on and modify facts about adapters. The facts transformed by comebacks are then serialized into executable binary adapters.

Our adaptation is transparent for existing applications, because the adaptation layers are generated at the framework site and are deployed (as separate *adapter components*) together with the refactored framework. To avoid name collisions of refactored API types and existing plugin types we separate name spaces of existing plugins and the upgraded framework using (in Java) separate class loaders. This name separation also permits us to support plugins' *side-by-side execution* [Ang02]: within the same application, adapted plugins of different versions may coexist and simultaneously use, together with newly developed plugins, the latest framework version.

We did not explicitly design a library of comebacks right from the start of ComeBack! development. Instead, we evolved the library during our case study (reported in Chapter 2) to compensate for

discovered application-breaking API refactorings. In addition, we extended the library to compensate for application-breaking API refactorings reported by Dig and Johnson [DJ05]. As a consequence, the functionality of ComeBack! permits us to adapt all application-breaking API refactorings detected in our case study and most of such refactorings reported by Dig and Johnson [DJ05] (except for refactorings of instance variables declared as public in APIs). Using the information about the application-breaking API refactorings obtained in our case study, we adapted two selected applications. In addition, we provided adapters for all application-breaking API refactorings used in RefactoringLAN (Appendix A). Our performance tests of the adapted applications show that, comparing to the execution time of original applications, in the worst case the performance overhead in adapted applications reached 6.5%, while in the best case the overhead was about 1%. In general, we consider performance pentalties implied by our adapters acceptable in a large number of adaptation scenarios.

ComeBack! is designed to be independent from a particular Integrated Development Environment (IDE) and can be combined with other adaptation tools and technologies. Its declarativeness, intrinsic to a logic-based implementation, allows us to combine the tool with the Eclipse refactoring log to reuse the refactoring history automatically recorded in Eclipse. In addition, ComeBack! is flexible enough to allow the combination of refactoring-based adaptation with other adaptation approaches. In particular, we estimated the feasibility of combining ComeBack! with *protocol adaptation* [YS97]—adaptation of improper message exchange among functionally compatible components. Integrating such adaptation in ComeBack! alleviates the task of writing and maintaining adaptation specifications used for adapting API changes other than refactorings.

The rest of the chapter is organized as follows: Section 5.1 presents the general architecture of ComeBack! and its implementation modules (Section 5.1.1), followed by the description of class loading implemented to separate framework and plugin name spaces, and to support side-by-side plugin execution (Section 5.1.2). Thereafter, the section reports on the current tool functionality and the performance of applications adapted by ComeBack! (Section 5.1.3). Section 5.2 discusses how ComeBack! can be empowered to cope with protocol adaptation by introducing the notions of protocol adapters and the required adaptation specifications (Section 5.2.1), showing the problems in maintaining adaptation specifications in the presence of API refactorings (Section 5.2.2), and solving these problems by using comeback-based adaptation (Section 5.2.3). Section 5.3 concludes the chapter.

## 5.1 ComeBack!: a Generative Tool for Refactoring-Based Adaptation

Because we aimed for developing a program that synthesizes other programs (i.e., adapters), it was natural to use generative techniques [CE00] to develop ComeBack!.

### 5.1.1 General Architecture and Implementation Modules

Figure 5.1 shows the general architecture of ComeBack! and the data flow among internal and external modules as performed during the adaptation. For a number of common refactorings we provide a comeback library consisting of the corresponding 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. Once all comebacks have been executed, the fact base contains the information necessary for generating adapters. More precisely, the fact base describes the adapters as abstract adapter specifications (as we discussed in Section 4.2.1) and can be used to create the corresponding binary adapters. That is, in ComeBack! we extract and transform the information about the program and not the program itself; the generation of executable adapters using this information is then the final step in adapter derivation.

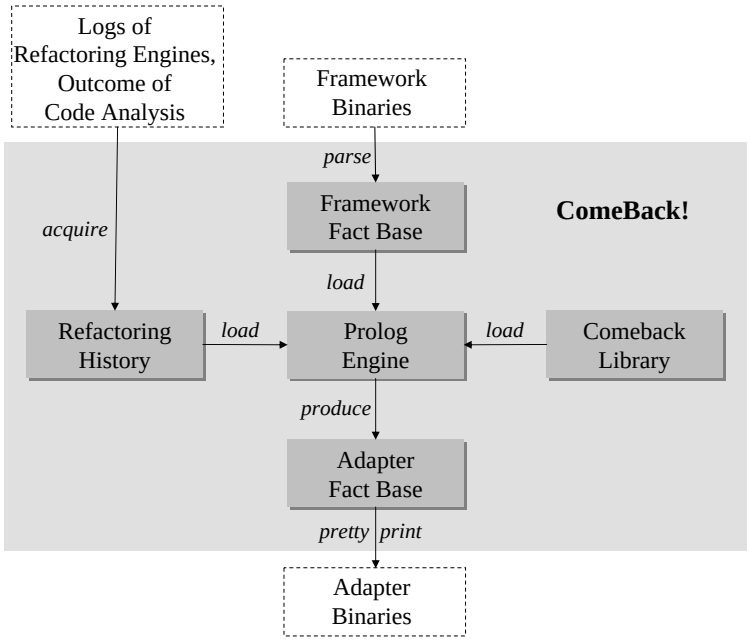



Figure 5.1: General architecture of ComeBack!. Grey boxes denote the input and output modules used by the Prolog engine, while dashed boxes denote the input and output modules external to the tool.

Although by sticking to the refactoring engine of one of the open-source development environments we could have partially reused its features (e.g., a parser), we would have been bound to the particularities of the engine's implementation. Since we envisage extending our technology to support languages other than Java (e.g., .NET), for which no open-source refactoring engines may exist, we decided to stay independent from such implementation particularities.

Figure 5.2 provides details on the internal implementation modules of ComeBack! and relates them to the process of creating adapters statically (i.e., before upgrading the framework) and dynamically (i.e., at application execution time), as we discussed in Section 3.2. In the top part of the figure (reading from left to write), the API binaries are parsed by a Java-to-Prolog parser, obtained facts are transformed by the corresponding comebacks, and the resulting adapter facts are serialized to the Java binary adapters by the tool's adapter generator (using the ASM code generation library [ASM]). The adapter generator packs all adapters and the adapter cache of the static adaptation layer into an adapter component to be deployed together with the refactored framework. In addition, the adapter generator serializes itself as a part of the adapter component; as a consequence, we can reuse the very same adapter generation logic for dynamically creating adapters whenever needed (bottom right part of Figure 5.2).

Since Prolog declarativeness permits us to abstract from irrelevant syntactical issues in the comeback implementations, the Java-specific details of program transformations are not included in the comeback implementations, and are addressed when emitting binary adapters. Listing 5.1 shows the comeback for the `MoveMethod` refactoring as implemented in ComeBack!. (The formal definitions of the `MoveMethod` refactoring and its comeback can be found in Section 4.3.5.) For convenience, we implemented the `MoveMethod` refactoring as a Prolog rule (lines 7–11 of Listing 5.1) and then reused its implementation to implement the corresponding comeback `CbMoveMethod` (lines 16–17).

We are able to combine ComeBack! with tools for acquiring and storing refactoring information. The latter is important in addressing the main prerequisite of our approach—the availability of the totally ordered API refactoring history. For Java, such history comes "for free" in the Eclipse and JBuilder IDEs, in which refactorings applied via the IDE facilities are automatically recorded in a refactoring log. For instance, the Eclipse environment provides a set of refactorings that, when applied, are automatically registered in a refactoring log as textual descriptors of the applied changes. We developed a query facility that reads in and

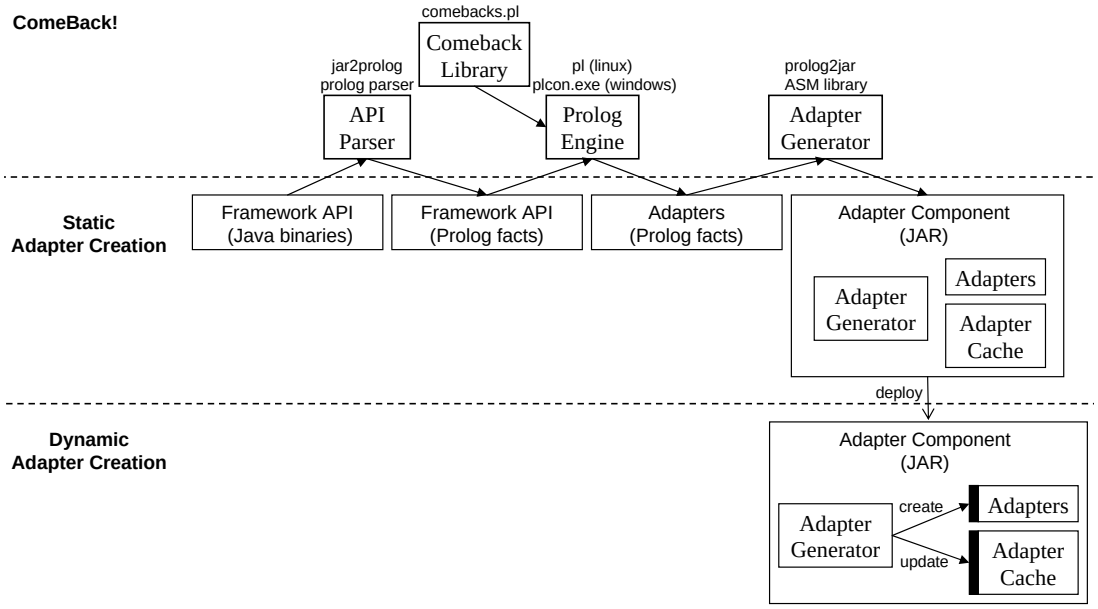



Figure 5.2: Internal ComeBack! modules related to static and dynamic adapter creation. The arrows in the top part of the figure shows the input to and output from the modules of ComeBack!. Above each module its implementation details (file or package names) are provided. Since the adapter generator is bundled together with the statically created adapters and the adapter cache, it can be reused for dynamically creating adapters at application execution time (shown as black extensions to the adapter set and adapter cache in the bottom right part of the figure).

Listing 5.1: The `MoveMethod` refactoring and its comeback as Prolog rules

```
%
% Moves the method Selector with the given Parameters from Class to NewClass
% and updates existing Senders.
%
% AST and Exceptions are parsed method implementation and exceptions, respectively
%
moveMethod(Class, Selector, Parameters, NewClass) :
        retract(method(Class, Selector, AST, Parameters, Exceptions)),
        retract(senders(Class, Selector, Parameters, Senders)),
        asserta(method(NewClass, Selector, AST, Parameters, Exceptions)),
        asserta(senders(NewClass, Selector, Parameters, Senders)).

%
% The comeback for MoveMethod moves the method back to its originating class
%
cbMoveMethod(Class, Selector, Parameters, NewClass) :
        moveMethod(NewClass, Selector, Parameters, Class).
```

serializes the Eclipse log into a set of corresponding Prolog facts to be reused in ComeBack!. According to these facts, the corresponding comebacks of the comeback library are selected and instantiated by the Prolog engine. In case of framework branching, the total order of the refactoring trace can be achieved by merging separately recorded refactorings [DMJN07].[1]

[1] We did not investigate applying our technology in the presence of branching.

### 5.1.2 Assembling Refactored Framework and Existing Plugins

As we have shown in Figure 5.2, once the binary adapters of the static adaptation layer are generated, they are packed (together with the adapter cache and adapter generator) into an adapter component to be deployed together with the refactored framework. From the point of view of the plugin, the framework upgrade is replacing one component by another component with the same name: an old version of either the framework itself or a previously deployed adapter component is replaced with the newest adapter component that uses the latest framework version. In this regard, an important characteristic of our technology is that it does not suffer from the so-called DLL Hell—a set of problems that may appear when a component used by multiple applications is replaced with another version [ESJ02]. Existing applications may stop working if the two component versions are not binary compatible (upgrade problem) or if the replacing version is older than the initial one (downgrade problem). In our case, the former problem is eliminated by the comeback definition and construction of binary-compatible adapters, and the latter by making adapters always use the latest framework version.

However, with regard to the safe deployment of the refactored framework, we still need to instruct the JVM about how to assemble the refactored framework and existing plugins via adapter components. In particular, we should address the following two issues:

- Name collisions of the reconstructed API and the refactored API. Since in exhaustive API adaptation the static adaptation layer completely reconstructs the API as expected by the old plugin, the names of the reconstructed API types and the refactored API types are the same (except for renamed API types). At run time, we should be able to differentiate between the types of the two APIs and selectively load a refactored API type or a reconstructed API type, depending on the calling site (i.e., the plugin or the framework).

- The Type Capture problem. As we discussed in Section 2.2.3, newly introduced or renamed framework types may accidentally have the same fully qualified names as old plugin types. In case a type defined at one site (i.e., the framework or the plugin) has already been loaded, while a type with the same fully qualified named defined at the other site is expected, the JVM would attempt to use the loaded type, possibly leading to application malfunction (Type Capture, Figure 2.11 on page 34). At run time, we should be able to load the type as expected by the caller, even if some other type with the same fully qualified name has already been loaded during the preceding application execution.

Addressing the two issues, we separate the name spaces of the framework and the plugin so that even if two types have the same fully qualified names, they are distinguished by the language execution environment. In Java, our solution is based on generating custom class loaders and uses the fact that, since classes are identified in the JVM by their names *and* the class loader that loaded them, classes with the same names loaded by separate class loaders can be distinguished effectively. Therefore, framework and plugin name spaces can be effectively separated by using separate class loaders to load the types of the old plugin and the refactored framework.

Technically, for the upgraded framework and an old plugin, we need two class loaders: a *framework class loader* to load framework types, and a *plugin class loader* to load plugin types. In addition, we need to load adapter types (classes and interfaces of the static adaptation layer), which also belong to one of the two name spaces. More exactly, the framework class loader is responsible for loading framework types and their adapter subtypes, while the plugin class loader is responsible for loading plugin types and all other adapter types:

- Certain adapter types of the static adaptation layer are subtypes of the refactored API types; therefore, these adapter types belong to the name space of the upgraded framework. Namely, such types are the classes realizing Covert Plugin Adapter of the White-Box Class Adapter pattern (Figure 3.7 on page 51) and Covert Plugin Adapter of the White-Box Interface Adapter pattern (Figure 3.10 on page 57).

- All other adapter types of the static adaptation layer belong to the plugin name space. Namely, such types are all reconstructed API types of the black-box and white-box adaptation patterns, and the classes realizing Covert Framework Adapter of the Black-Box Interface Adapter pattern (Figure 3.5 on page 47).

As a consequence of using these two class loaders, at run time we create two different name spaces: the name space of framework types and the name space of plugin types. Adapter types are present in both name spaces, but because they are named as expected by the plugin (in case of reconstructed API types) or their names are obfuscated (in case of covert adapter classes), adapter types cannot introduce any name collisions.

The particular implementation of the separate class loading depends on the launching site: either the plugin or the framework starting the application (in other words, defining the *main()* method). At the launching site, we assume the Java (default) system class loader,[2] while at the other site we use a custom class loader automatically created at adapter generation time to load types reflectively.[3]

#### Plugin Launching the Application

In this scenario, the plugin contains the code to launch the application and provides the value of the CLASSPATH environment, referring to the framework's JAR file. While upgrading the framework, we replace the old framework JAR file by the generated adapter JAR file, so that the plugin's class path does not need to be adjusted. At the same time, the JAR file of the upgraded framework is placed in some other directory (in our case, named according to the framework's version). Since the adapter component is named exactly as the framework, while its adapters reconstruct the API expected by the old plugin, the fully qualified names of the expected framework types and the reconstructed types of the adapter component are identical. Therefore, although the (old) framework type names are embedded in the plugin binaries, the dynamic linking mechanism of Java will locate the corresponding adapter types required from the plugin. As a consequence, after the framework upgrade the plugin will load adapter types whenever it was loading API types before the upgrade.

The adapters being loaded by the plugin are equipped, in turn, with a custom framework class loader used whenever a framework type (or an adapter type subtyping a framework type), which should be used as an adaptee, is to be loaded. In such cases, we effectively switch from the system class loader used for loading plugin types to the custom framework class loader. In all other cases (when a plugin type is loaded from the plugin, or a framework type is loaded from the framework), the type is loaded by the corresponding class loader (the system class loader or the framework class loader, respectively).

#### Framework Launching the Application

In many big frameworks, such as Eclipse [Ecl] and NetBeans [Net], the framework is usually responsible for launching its applications. In such cases, the plugin's JAR file is deployed in a prescribed directory, while the framework provides a dedicated plugin class loader to fetch plugin types from the provided JAR file. The actual loading of plugin types may occur at the system startup or later on demand.

In contrast to the plugin launching the application, in this scenario the name spaces of the framework and plugin are already separated: framework types are loaded by the framework class loader (e.g., the system class loader assumed in our discussion), while plugin types are loaded by the plugin class loader. However, after the framework upgrade the old plugin class loader is not aware of the introduced adapters of the plugin's adapter component. What we need is to make the plugin work with adapter types instead of framework types (as before the framework upgrade). Therefore, we generate a new plugin class loader that makes plugin types access adapters of the plugin's adapter component, instead of framework types. The old plugin class loader is then replaced by the generated plugin class loader.

[2] The following discussion also applies to the cases, when the launching site uses a custom class loader. In our discussion, we omit such cases for simplicity.

[3] In contrast to Java, in .NET such solution can be based on installing shared components in the global assembly cache (GAC) and does not require using reflection.

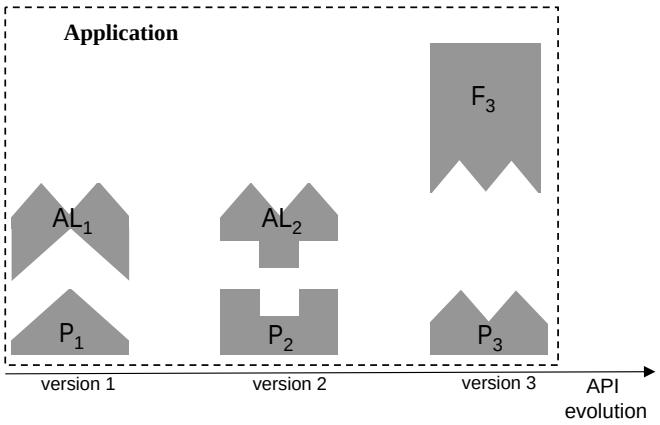



Figure 5.3: Side-by-side plugin execution. Within the same application, plugins of different versions execute simultaneously with the upgraded framework. Adapter components containing the adaptation layers $AL_1$ and $AL_2$ are named as the upgraded framework of version $F_3$.

**Supporting Side-by-Side Plugin Execution.** In case an application evolves together with the framework and consists of plugins of different versions, we support plugins of different versions of the same application to simultaneously use the upgraded framework, either directly (for newly created plugins) or through the corresponding adapter components (for plugins of old versions). To achieve this, because all adapter components and the framework have the same name (say, $F$) differing only in version numbers, we must support *side-by-side execution* [Ang02]—when multiple versions of $F$ are deployed, any existing component has to find its "right" version of $F$ (Figure 5.3).

To achieve side-by-side execution of plugins with the upgraded framework, the run-time system must differentiate among versions of $F$ (i.e., the upgraded framework and adapter components).

- Plugins have to use their corresponding adapter components. In our solution, each provided plugin class loader uses exactly the adapter component, which corresponds to the particular plugin version.
- Adapter components have to use the latest version of $F$ (i.e., the framework itself). In our solution, each adapter component gets the framework class loader used to load types of the upgraded framework (e.g., the system class loader) and uses this class loader whenever a framework type needs to be loaded.

In effect, even in presence of multiple plugins developed against different framework versions and used within the same application, the framework upgrade is completely transparent. Moreover, new plugins can be developed with the upgraded framework and integrated into the application.

### 5.1.3 Evaluation of ComeBack! Functionality and Adapters' Performance Overhead

We evolved the functionality of ComeBack! during our case study. Two applications adapted using the results of the case study as well as the adapted API types of RefactoringLAN were used to compare the performance of the original and the adapted applications.

#### Functionality

Table 5.1 lists the functionality of ComeBack! at the time of writing. In the table, we group supported refactorings according to our discussion of adaptable API refactorings from Section 3.3.4. For the refactorings requiring additional input from developers, such as `RemoveMethod`, we provide default (e.g., empty)

| Group | Refactorings |
|---|---|
| Affecting API method signatures and locations | RenameMethod AddParameter RemoveParameter PushDownMethod MoveMethod ReplaceDataValueWithObject |
| Affecting API type names and locations | RenamePackage MovePackage RenameClass MoveClass RenameInterface MoveInterface |
| Adding API types and methods | AddEmptyClass AddInterface AddException AddMethod ExtractMethod AddEncapsulatedField |
| Removing API types and methods | RemoveEmptyClass RemoveInterface RemoveException RemoveMethod InlineMethod RemoveEncapsulatedField |
| Composite API refactorings | AddClass RemoveClass ExtractClass InlineClass ExtractSubclass ExtractInterface |

Table 5.1: Application-breaking API refactorings adapted in ComeBack!. The adaptation of the composite API refactorings (the last group in the table) is achieved via adapting their constituents—refactorings of the first four groups in the table.

implementations, which can be changed on demand for a particular application of the refactoring. Moreover, for `InlineMethod` we assume that the inlined method was not used as a (non-abstract) hook method in the old API (as discussed for the comeback of this refactoring in Appendix D.6, page 190). The current comeback library can adapt all refactorings discovered by Dig and Johnson [DJ05] (reported in detail in Dig's dissertation [Dig07, p. 21]), except for refactorings moving and renaming public API fields.[4] It also can adapt all refactorings discovered in our case study, because all API variables of JHotDraw and SalesPoint are encapsulated.

### Efficiency

Concerned about the performance penalties inevitable when using adapters, we implementing a number of static and run-time optimizations discussed in Section 3.3.3. To test the performance of the adapted applications, by considering the application-breaking API refactorings discovered in our critical case study of API evolution (Section 2.1) we created adapters for the most advanced sample application of JHotDraw and one of the student applications of SalesPoint. Complementary to the adapters created in the case study, we also created and tested adapters for the examples of application-breaking API refactorings used in our RefactoringLAN (Appendix A).

In more detail, for the two chosen applications of JHotDraw and SalesPoint, we specified the corresponding comebacks for all encountered application-breaking API refactorings. Moreover, to obtain executable applications we had to adapt discovered application-breaking API changes other than refactorings, and we could not manually insert the corresponding adaptation code directly into binary adapters. However, since after our application-based change discovery we knew exactly how the adaptation code for all such changes should look like, we used this knowledge to adapt changes beyond refactorings using ComeBack!. To begin with, we executed all comebacks and obtained the adapter fact base. Then, we manually augmented it with

[4] Since we are consistent with the terminology of Roberts [Rob99], most refactorings in Table 5.1 do not exactly correspond to the refactorings described by Dig and Johnson [DJ05], who used a different terminology for describing application-breaking API refactorings. However, while supporting refactorings of Table 5.1, in effect we also can compensate for most refactorings reported by Dig and Johnson [DJ05].

additional facts describing how to compensate for remaining changes. For example, the *FigureEnumerator* of JHotDraw 6.0 was wrapped into an implementation of the standard *Enumerator* used in JHotDraw 5.2. Thereafter, we generated executable binary adapters.

We conducted the performance testing under the Win Vista 32 bit, using the CPU Athlon Turion TL60 (2 core with 2 GHz each, 64 bit) and 2 GB RAM. For the adapters generated in the JHotDraw case study, the overhead reached 6.5%, mostly because of collection types commonly present in API method signatures. At the same time, the SalesPoint adapters implied less than 4.5% of overhead. Finally, the performance penalties implied by the adapters of RefactoringLAN were only 1%, because no collection types and only a few user-defined types are present in the signatures of its API methods.

Since we use the very same code generation logic and implementation to create adapters statically and dynamically, creating an adapter statically or dynamically takes almost identical time.[5] As a consequence, assuming the ratio of all dynamically created adapters to the statically created adapters as $X\%$, and, furthermore, assuming the same average type size of adapters created dynamically and statically (which impacts the time of code generation), creating adapters dynamically will take $X\%$ of the time required to generate the static adaptation layer. For the both applications adapted in our case study as well as for the refactored types of RefactoringLAN the generation of their static adaptation layers took less than 2 seconds (for RefactoringLAN, less than 1 second). Therefore, we considered the overhead implied by generating adapters dynamically negligible.

In general, it is impossible to precisely anticipate the performance overhead for a particular adapted framework-based application. Even if one would attempt to anticipate, using static code analysis, the dependency between the ratio of user-defined and collection types in the framework API and the performance overhead implied, the actual run-time overhead may vary considerably. Since usually only a subset of API types is reused in plugins, while the latter are usually not available for analysis, one cannot anticipate exactly which API types are reused and how. With regard to the collection types, their actual number of elements at run time is usually unknown and cannot be anticipated statically. In addition, considering other implementation platforms (e.g., .NET) one could implement certain adaptation actions more efficiently. For example, in .NET there is no need to use reflection for loading classes from separate name spaces, as it is required in Java. Moreover, it is impossible to anticipate neither the number of adapter types to be created dynamically nor their size, the latter having a direct impact on the speed of dynamic adapter creation. All in all, the performance overhead implied by adapters should be tested for each particular adaptation scenario, while the developers should decide, whether the estimated performance penalties are acceptable for the particular application in question.

## 5.2 Combining ComeBack! with Protocol Adaptation

Although in general the refactoring-based adaptation is capable of adapting more than 80% of application-breaking API changes, its restriction to pure structural changes also entails its limitations. API changes other than refactorings may require either manual or specification-based adaptation. For example, by introducing and removing API methods, or changing the order of calling certain API methods, framework developers may change the way the framework and its plugins should interact via message exchange (i.e., a sequence of method calls). If existing applications are not aware of such changes, the latter may lead to *protocol mismatches* characterized by improper message exchange among interacting components of an application [BBG+04], in this case between classes of the upgraded framework and old plugins. In component adaptation, to bridge protocol mismatches developers have to specify *adaptation specifications*—formal descriptions of messages that components may send and receive, as well as the valid message exchange. Basing on these adaptation specifications, *protocol adapters*, which support proper intercomponent communication, can be generated semi-automatically (e.g., [YS97, SR02a, CPS06, IT03, BCP06]).

[5] The only difference is the time required to collect metadata about the adapter to be created dynamically, as we discussed for Listing 3.4 on page 81. Since collecting this information uses the cached data in the adapter cache, it is highly efficient and imposes no noticeable performance overhead at run time.

However, once such specifications are provided, component evolution unavoidably demands their *maintenance*, because evolutionary API changes may alter component parts on which existing specifications rely, rendering the latter invalid. For instance, if a component type referred to in a specification is renamed by a component change at some later point in time, the specification is no longer valid, unless it is updated as well to reflect the renaming. In such cases, specifications must be updated along with the changes of the involved components, which raises the complexity and costs of adaptation.

Extending our refactoring-based approach, we want to maximally ease the task of adapting remaining application-breaking API changes not compensated for by refactoring-based adaptation. In particular, for protocol changes the corresponding adaptation specifications must be undemanding to write and should not require maintenance throughout subsequent framework API evolution via refactorings. We show how ComeBack! can be empowered to help framework developers writing adaptation specifications that are:

- **in-time**: specified at the actual time of framework change. It is inherently easier to specify the adaptation of small incremental API changes upon their application than to write large specifications involving complex implementation dependencies as one big step before upgrading the framework.
- **durable**: valid at the time of executing the specifications regardless of any API refactoring applied after the specifications were written. This eliminates the need to maintain adaptation specifications.

Given adaptation specifications, we use the information about subsequent API refactorings to perform our refactoring-based adaptation, which, besides compensating for application-breaking API refactorings, also shields the adaptation specifications from those refactorings. In this way, we adapt existing adaptation specifications in the sense that we preserve their validity in the context of API evolution via refactorings.

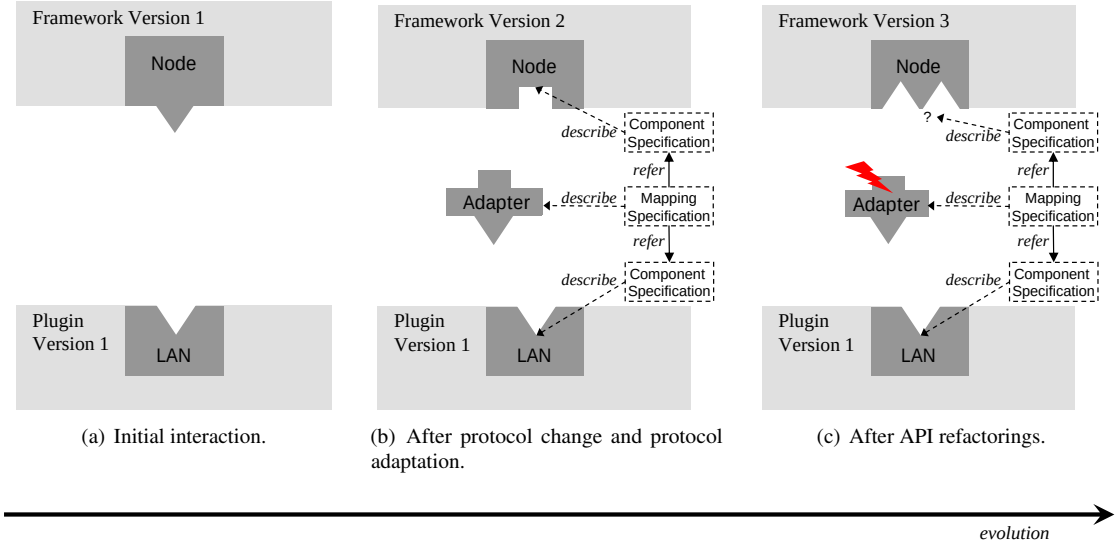
### 5.2.1 Introduction to Protocol Adaptation

Yellin and Strom call two components *functionally compatible*, when "at a high enough level of abstraction, the service provided by one component is equivalent to the service required by the other" [YS97]. Although functionality compatible, such components may fail to interact with each other via message exchange, if they expect from each other a different message order, or the occurrence of different messages during interaction—a case of protocol mismatch [BBG+04]. The problem stems from the fact that component expectations about the proper message exchange are usually hard-coded in method implementations, or are described informally in textual documentation, and may match badly, when combining independently developed components [YS97].

Discussing component adaptation, Becker et al. argue that the following protocol mismatches can be adapted [BBG+04][6]:

- **Non-matching message ordering.** Although components exchange the same kind of messages, their sequences are permuted.
- **Surplus of messages.** A component sends a message that is neither expected by the connected component nor necessary to fulfill the purpose of the interaction.
- **Absence of messages.** A component requires additional messages to fulfill the purpose of the interaction. The message content can be determined from outside (e.g., from the execution environment).

[6] Strictly speaking, protocol mismatches form a subset of *assertion mismatches*, because protocols can be specified by assertions of preconditions and postconditions, as shown by Meyer [Mey00, pp. 981–982]. However, when discussing the type checking performed at compile time, Meyer argues, that in general to identify mismatching assertions "would require a sophisticated theorem prover which, if at all feasible, is still far too difficult (in spite of decades of research in artificial intelligence)" [Mey00, p. 578]. In contrast to general assertion mismatches, Becker et al. [BBG+04] define a protocol mismatch as a special assertion mismatch that can be identified statically by comparing protocol specifications, stated in a formalism of a limited power (e.g., final state machines).




(a) Initial interaction.

(b) After protocol change and protocol adaptation.

(c) After API refactorings.

Figure 5.4: API evolution via protocol and structural changes. Initially, the framework and plugin classes cooperate as intended. To accommodate for protocol changes in *Node*, the second release of the framework requires additional interface and mapping specifications. The latter are then invalidated by subsequent API refactorings in the third framework release.

To reason about components' functional compatibility (e.g., deadlock-free component interaction), several authors (e.g., [YS97, SR02a, CPS06, IT03, BCP06] propose to augment component APIs with formal *interface specifications* of message sequencing constraints. If such interface specifications are in addition related by formal *mapping specifications* that describe the proper message exchange between components, both kinds of specifications make up adaptation specifications, which can be used to semi-automatically derive protocol adapters for the components in question [YS97, SR02a, CPS06, IT03, BCP06].

Similar to incompatibilities of independently developed components, protocol mismatches may occur in the context of framework API evolution. Namely, protocol mismatches between an upgraded framework and existing plugins may be caused by evolutionary framework transformations that change the order or occurrence of method calls between the framework's API classes and application-specific classes defined in plugins.

As an example, let us consider two cooperating classes of Figure 5.4(a): the *Node* class of the framework and the *LAN* class of the plugin (their API methods are omitted in the figure).[7]

- The *Node* class abstracts a separate network node and provides, among other methods, a method *broadcast()* to broadcast the given string message in the network.
- The *LAN* class models a local area network that can accept and remove nodes via the methods *enter()* and *exit()*, each accepting as their single parameter an instance of *Node*. In addition, *LAN* has a method *broadcastAck()*, the invocation of which indicates a successful message sending within the network.
- In the first framework version, *LAN* and *Node* collaborate as follows: First, a node may enter the network by calling *LAN*'s *enter()* method and passing a reference to itself as the actual method argument. Thereafter, *LAN* may ask the node to broadcast a message by calling the node's *broadcast()*

[7] Although the following example overlaps with RefactoringLAN, we did not include all methods used in the example into RefactoringLAN, because the example uses three consecutive framework versions and would unnecessarily complicate RefactoringLAN. The example will be used solely in this section, and is not referred to in other sections of the thesis.

with a string message to be sent. The node creates a packet out of the given message, forwards it to certain network nodes, and acknowledges back to the network by calling *LAN*'s *broadcastAck()* method. Finally, the node wishing to leave the network issues an exit request (*LAN*'s *exit()*) and is removed from the network by *LAN*.

It is important that, to properly implement the aforementioned class collaboration, application developers have to follow certain programming guidelines when implementing the *LAN* class in the plugin. For instance, they could implement an API interface (say, *ILAN*) to provide *LAN*'s method signatures as expected by *Node*, and they could check a developer guide to properly implement *LAN*'s *enter()* and *exit()* methods.

In the second framework release, framework developers generalize *Node* to increase its reusability for various types of networks. In particular, the developers introduce a two-phase protocol for entering the network: the node first asks for a permission to enter the network calling the method *enterReq()*, passing a reference to itself as the method argument, and only if allowed actually enters the network. The same changes are applied to the protocol for leaving the network. In addition, there is no acknowledgment of a successful broadcasting sent back from *Node* to *LAN* anymore. Instead, a node issues a *ceased()* message returning an instance of *Packet*, which is removed from the network by the node.

These API changes lead to a protocol mismatch: although the classes possess the functionality required for an interaction, they cannot properly interact and require adaptation. Since framework APIs are usually limited to the syntactical representation of the API types and methods, the adaptation of protocol mismatches requires additional formal adaptation specifications. In our adaptation, such specifications can be provided by framework developers. This is possible, because framework developers know exactly (1) the API classes being changed (e.g., *Node*) to provide their interface specifications, (2) the method signatures and message exchange in application classes involved in the interaction (e.g., *LAN*) to provide their interface specifications, and (3) the impact of framework changes on class interaction to provide mapping specifications. Using interface and mapping specifications, adapters can be derived to take over the proper message exchange between the upgraded framework and existing plugins (Figure 5.4(b)).

For our running example, the two top listings of Figure 5.5 show the behavior interfaces of *LAN* (left part) and *Node* (right part) specified using the simplified formalism of Yellin and Strom [YS97]. The interfaces (called, according to the original terminology [YS97], *collaborations* in the listings) are described as a set of sent and received messages augmented by a finite state machine specification. The latter defines the legal sequences of messages that can be exchanged between an instance of the class and an instance of the class' collaborator (*init* stands for the initial state, $-$ for sending, and $+$ for receiving messages). For example, in *Node*'s interface specification, after sending the enter request (line 18), an instance of *Node* may only accept either a *maynotEnter* message (line 19) or a *mayEnter* message (line 20). In the latter case, the instance can enter the network (line 21) and broadcast packets (line 22). For clarity, the valid states of *Node* are marked by numbers (1–6) and those of *LAN* by letters (A, B).

For these two interfaces, the bottom listing of Figure 5.5 provides an interface mapping *NodeLAN* relating the classes' states and messages, and effectively describing the corresponding protocol adapter for the two classes. For each permitted combination (enclosed in angle brackets $<>$) of the collaborators' states (denoted by numbers for *Node* and letters for *LAN*), the mapping specification defines the allowed transitions of both collaborators and the valid memory state *M* of messages being exchanged. For instance, after receiving an enter request from a node (line 2), the adapter allows by default the node to enter the network (line 4), but does not insert the node into the network yet. Instead, it saves the node in the adapter's internal memory (line 3). When the adapter receives the node's *enter* message (line 5), it obtains the saved node from the internal memory, cleans the memory, and injects the node into the network (lines 6–8). To keep track of the states of *Node* and *LAN*, the adapter updates its internal state information correspondingly (in between the brackets $<>$).

As mentioned, to properly exchange messages, certain method parameters need to be saved by the adapter internally, by a call *write* to the implicitly generalized virtual memory (as *thisNode* in line 3). When the saved parameters are not needed anymore, they are deleted from the virtual memory by the adapter, as denoted by *invalidate* (e.g., in line 8). Besides message reordering, by storing and retrieving method parameters the adapter can handle also message absence (e.g., *broadcastAck* to *LAN*, *mayEnter* and *mayExit* to *Node*) and message surplus (*ceased* from *Node*).

```
Collaboration Node {
  Receive Messages{
    mayEnter();
    maynotEnter(String why);
    mayExit();
    maynotExit(String why);
    broadcast(String message);
  };
  Send Messages{
    enterReq(Node thisNode);
    enter();
    exitReq(Node thisNode);
    exit();
    ceased(Packet packet);
  };
  Protocol{
    States{1(init),2,3,4,5,6,7};
    1 −enterReq ⟶ 2;
    2 +maynotEnter ⟶ 1;
    2 +mayEnter ⟶ 3;
    3 −enter ⟶ 4;
    4 +broadcast ⟶ 4;
    4 −ceased ⟶ 4;
    4 −exitReq ⟶ 5
    5 +mayExit ⟶ 7;
    6 +maynotExit ⟶ 4;
    7 −exit ⟶ 1;
  }
}
```

Interface specification of *Node*

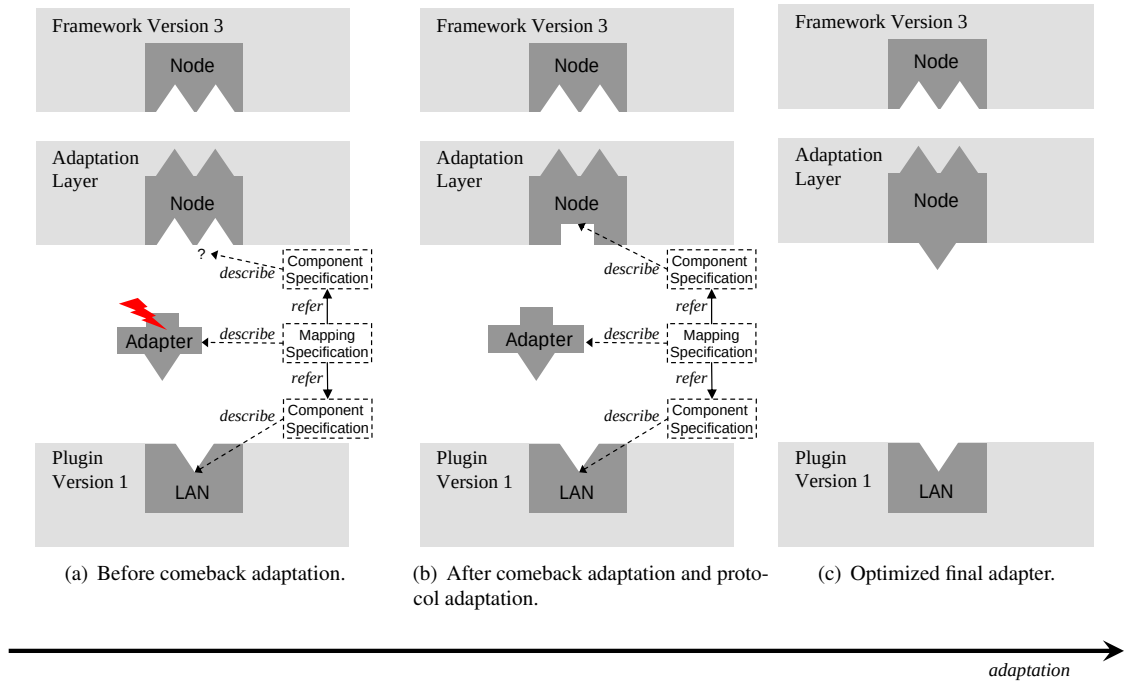
```
Collaboration LAN {
  Receive Messages{
    enter(Node node);
    exit(Node node);
    broadcastAck();
  };
  Send Messages{
    broadcast(String message);
  };
  Protocol{
    States{A(init),B};
    A +enter ⟶ A;
    A +exit ⟶ A;
    A −broadcast ⟶ B;
    B +broadcastAck ⟶ A;
  }
}
```

Interface specification of *LAN*

```
/*LEGEND M0={} M1={thisNode} M2={message}*/
<1,A,M0>: +enterReq from Node ⟶ <2,A,M1>,
    write(thisNode);
<2,A,M1>: −mayEnter to Node ⟶ <3,A,M1>;
<3,A,M1>: +enter from Node ⟶ <4,A,M1>;
<4,A,M1>: −enter to LAN ⟶ <4,A,M0>,
    node = read(thisNode),
    invalidate(thisNode);
<4,A,M0>: +broadcast from LAN ⟶ <4,A,M2>,
    write(message);
<4,A,M2>: −broadcast to Node ⟶ <4,B,M0>,
    message = read(message),
    invalidate(message);
<4,B,M0>: +broadcastAck to LAN ⟶ <4,A,M0>;
<4,A,M0>: −ceased from Node ⟶ <4,A,M0>;
<4,A,M0>: −exitReq from Node ⟶ <5,A,M1>,
    write(thisNode);
<5,A,M1>: +mayExit to Node ⟶ <5,A,M1>;
<5,A,M1>: −exit from Node ⟶ <6,A,M1>;
<6,A,M1>: +exit to LAN ⟶ <6,A,M0>,
    node = read(thisNode),
    invalidate(thisNode);
```

Mapping specification *NodeLAN*

Figure 5.5: Examples of adaptation specifications. An interface specification (each of the two top listing) describes class behavior as a set of messages the class can receive and send, together with a finite state machine that defines the set of legal class' states and transitions. By referring to interface specifications of collaborating classes, a mapping specification (the bottom listing) relates the classes' states and transitions that should occur during message exchange.




(a) Before comeback adaptation.

(b) After comeback adaptation and protocol adaptation.

(c) Optimized final adapter.

Figure 5.6: Adaptation of protocol and structural API changes. The comeback execution reestablishes the context of formal adaptation specifications. The latter can then be executed directly on the adaptation layer to insert the protocol adaptation code into previously created adapters.

### 5.2.2 Refactorings Invalidate Adaptation Specifications

Let us assume the adaptation specifications of Figure 5.5 are valid, the corresponding adapter bridging the protocol mismatch is derived, and the second framework release (including the changed *Node* class) is deployed successfully. However, in the third framework release, to vary on communication strategies besides the default broadcasting, framework developers rename the method *broadcast()* to *manycast()* (the `RenameMethod` refactoring) and, in addition, introduce the integer number of a communication strategy as the second argument of the renamed method (the `AddParameter` refactoring). The refactoring will clearly invalidate the interface specification of *Node* (the top left listing of Figure 5.5), which uses the old method name (lines 7 and 22) and the old list of method arguments (line 7). Moreover, the refactoring will also invalidate the *NodeLAN* mapping specification, which, in addition to using the obsolete method name, does not handle properly the newly introduced method parameter (lines 11–13). As a consequence, the API refactoring will render the adaptation specifications invalid (Figure 5.4(c)).

The situation aggravates with the growing number of releases and chaining of interdependent modifications. For instance, the developers of *Node* could later add a parameter to *maynotEnter* (the `AddParameter` refactoring) specifying the reason of rejecting the entrance of a node into the network. In general, any syntactic change affecting classes referred to by adaptation specifications will invalidate the latter. The reason is that by changing the syntactical representation, the refactorings modify the initial context, on which the specifications depend. Either the specifications have to be updated to match the new context or the context itself has to be recovered. We perform the latter using comebacks.

### 5.2.3 Recovering Adaptation Specifications by Comebacks

Besides automatically bridging the signature mismatches caused by refactorings, the execution of comebacks leads to another important result—it reestablishes the context for previously written adaptation specifications.

The intuition is that by effectively inverting refactorings on adapters, comebacks reconstruct the "right" syntactic names used in the adaptation specifications.

Figure 5.6(a) shows the state of adaptation before executing the comebacks for the refactorings `AddParameter` and `RenameMethod` applied to the framework API in the third framework release. In between the *LAN* and *Node* classes there is an adapter class (also called *Node*), possibly created in a step-wise manner by previously executed comebacks. At this stage, the *NodeLAN* specification is not (yet) valid. Now the comebacks of `AddParameter` and then of `RenameMethod` transform the adapter, leading to the situation shown in Figure 5.6(b). Because the comebacks inverted the original API refactorings on the adapter, the latter has exactly the same syntactic form as the original API class *Node* before the refactorings in Figure 5.4(b). At this stage of adaptation, the interface specification for *Node* and the mapping specification for *NodeLAN* become valid again and can be used (together with the interface specification for *LAN*) to derive the corresponding protocol adapter. That is, the comebacks reconstruct the syntactic parts of API types, to which the adaptation specification relates; the only difference is that now these types are adapters. As a consequence, the developer can provide adaptation specifications at the time of the actual protocol change, and the specifications will not be invalidated by subsequent API refactorings.

There is an important difference between the two middle subfigures of Figures 5.4 and 5.6: while in Figure 5.4(b) the protocol adapter is generated between the two initial classes, in Figure 5.6(b) it is generated between the LAN class and the previously derived adapter leading to stacking of adapters. Although not affecting the actual functionality, adapter stacking may considerably decrease the performance due to the additional layer of redirection. Even more important, generating protocol adapters separately would mean stopping the execution of comebacks, executing (using another tool) adaptation specifications for the protocol change in the API change history, combining in some way the obtained protocol adapter with the static adaptation layer, and then continuing with the comeback execution.

Fortunately, there is no need for separately generating protocol adapters in ComeBack!. Instead, given interface and mapping specifications, the validity of which is proved at the specification level (e.g., using a formal verifier of final state machines for the formalism used by Yellin and Strom [YS97]), we can translate the specifications into a corresponding set of Prolog facts and rules. The execution of these rules at the time of protocol adaptation updates the adapter fact base created by previously executed comebacks (and, possibly, previously created protocol adapters). Because the description of the protocol adapter is thus effectively embedded into the fact base, no separate adapter is required (Figure 5.6(c)).

Technically, assuming adaptation specifications are written in the formalism of Yellin and Strom [YS97] and are proved to be correct (e.g., by a verifier), we do not have to implement the state machines in our adapters. By construction, the application code behaves according to the state automaton before the upgrade and valid adaptation specifications do not change that behavior. In fact, the only states the generated code for protocol adaptation needs to maintain at run time is the pointer to the adapter's adaptee object and a storage for temporarily saved messages to implement the *write* and *read* operations of the mapping specification (e.g., lines 3 and 7 of the bottom listing in Figure 5.5). As a consequence, the task of combining protocol adaptation and refactoring-based adaptation is reduced to generating custom adapter methods that (1) read and write message contents to a local backing-store, such as a hashtable (to save and retrieve data exchanged), (2) perform actual message forwarding, (3) send default messages (in case of message absence), or (4) do nothing (in case of message surplus).

## 5.3 Summary

The binary adaptation performed by our tool ComeBack! is efficient and unobtrusive for existing plugins: they need neither manual adaptation nor recompilation. At run time, to transparently connect old plugins and the upgraded framework via adapter components, while avoiding accidental type name collisions, we separate the name spaces of the refactored framework and old plugins. In addition, for the applications consisting of plugins of different versions, we support the side-by-side plugin execution with the upgraded framework. The current functionality of the tool allows us to compensate for a number of common application-breaking API refactorings reported in the literature [DJ05] and found in our empirical investigations. According to

our performance tests, introducing adapters decreased the performance of the adapted systems by 6.5% in the worst case and by 1% in the best case. We believe that such performance degradation is acceptable for most framework-based applications. Moreover, since adapter components are regenerated for each new framework release and operate directly with the latest framework version, we avoid performance degradation that could be caused by stacking adapters.

Refactoring-based adaptation as implemented in ComeBack! is also undemanding by alleviating the process of writing and maintaining adaptation specifications. We show how protocol adaptation can be combined with ComeBack!: given a valid protocol adaptation specification, it is converted into corresponding facts and rules inside ComeBack! that generates the corresponding adaptation code. Although such combination is indeed possible, its implementation is out of the scope of the thesis. Integration of formal adaptation specifications, backed up with the verification of those specifications, remains a future work in the implementation of ComeBack!.

Finally, although in our discussions we focused on application-breaking API changes, not all API changes are harmful to existing applications. In fact, some changes in framework functionality should propagate to existing applications. For example, existing applications should benefit from the framework's performance optimizations (e.g., achieved by refactoring sequential code into parallel code to run efficiently on multicores [DME09]). Similarly, changes of certain default business values encapsulated in the framework (e.g., default tax values in tax management applications) should propagate to existing applications from the upgraded framework. While for the application-breaking API changes framework developers should provide compensating adapters, for changes that should propagate to existing applications framework developers should *test* change visibility in the corresponding adapter components. Integrating such testing in our adaptation technology is a future work.

# 6 Related Work

In this chapter we overview existing approaches relevant for realizing adapters (Section 6.1), and for preserving backward compatibility of modified software components and their dependent applications (Section 6.2).

## 6.1 Approaches to Realize Adapters

There are several approaches that could be employed to realize adapters in our context of adapting evolved APIs of object-oriented frameworks. For instance, one could implement adapters using aspect-orientation technologies [KHH+01], such as *composition filters* [BA01]. By intercepting method calls between the framework and plugins, such adapters would, for example, convert mismatching types, insert missing method arguments, properly redirect old calls to modified API methods, or provide the implementation of missing API methods. However, besides depending on additional tools (to define, compile, and load adaptation aspects), such adaptation technology would also require accessing plugin sources or binaries, unavailable in our adaptation context.[1]

Warth et al. [WSM06] introduce *expander*—a language construct that allows to unintrusively update existing classes with new methods, fields, and implemented interfaces. Application developers can customize the behavior of an API class by importing a set of expanders, the methods of which can also override the class' methods. Among possible uses of expanders, Warth et al. show their employment in black-box component reuse to adapt mismatches of method names as provided by an API class and expected by application code [WSM06]. Besides adaptation in black-box framework reuse, by implementing proper method dispatch inside expander methods, we could in principle support adaptation in the presence of white-box framework reuse. In addition, although expanders introduce object schizophrenia [WSM06], we could address this problem similarly to our technology (e.g., by caching expanders). However, using expanders requires a Java language extension (eJava), currently implemented only for a limited subset of Java [WSM06]. More importantly, expanders become available only when explicitly imported by application code, which is against one of the main assumptions of our adaptation technology, that of plugins available for neither analysis nor update.

By arguing for a proxy as a general concept—"an object mimicking another object"—Eugster observes its reincarnations in such design patterns as Proxy, Decorator, and Adapter [GHJV95], as well as in *remote invocations*, so-called *future objects* in asynchronous invocations, and *behavioral reflection* [Eug06]. In principle, the power of the Proxy design pattern [GHJV95, p. 207] would suffice to implement adapters required in our adaptation context: although currently Java supports *dynamic proxies* only for interfaces, one could employ a non-standard extension of the JVM [Eug06] or a code generation library (i.e., CGLIB [CGL]) to provide proxies also for classes. By considering the information about API refactorings we would reconstruct the old API as expected in the plugin, while for each reconstructed API type and refactored

[1] At the time of writing, no standard support for aspect-orientation is included in the JVM specification by Sun. Therefore, adaptation solutions at the level of JVM would require its non-standard extensions.

API type we would statically generate a proxy class equipped with the corresponding adaptation logic.[2] However, technically it would require modifying framework and plugins in order to update their calling sites to use introduced proxies. Moreover, there is an important conceptual difference between the key abilities of the Adapter and the Proxy design patterns: "An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject [... or] its interface may be effectively a subset of the subject's" [GHJV95, p. 215]. In our adaptation context, we primarily need the former ability, and not the latter one.

## 6.2 Change-Aware Backward Compatibility Preservation

Griswold and Notkin [GN93] discuss how to automatically propagate changes in the software libraries to existing applications. Their approach requires the full availability of application source code at the time of library change. The same requirement was assumed in other earlier works on evolutionary transformations of component-based applications (e.g., [JO93, Ber91, LHX94]). The approaches overcoming this often unacceptable requirement can be classified into three main groups: *prescription*, *prevention*, and *facilitation*. For preserving backward compatibility of modified components and existing applications, such approaches consider the properties and information about component changes (to be applied or already applied); therefore, we call these approaches *change-aware*.

### 6.2.1 Prescription

Approaches of this group prescribe component and application developers, how to properly maintain and reuse components.

#### Maintenance prescription

Often the way developers should change a component is prescribed either implicitly (e.g., implied by the programming practice in general or by some shared knowledge in a certain developer community), or in the form of best practice tutorials and handbooks. For example, the release guideline of the Java-based framework Struts [Str] defines three types of component APIs (external, internal, and private) that are used to classify component changes, prescribe strategies of component evolution, and define policies of component upgrade minimizing backward incompatibility with other Struts components.[3]

The Java language specification [GJSB05] aims for binary compatibility by defining a set of changes that are allowed to be applied to existing code without affecting linking and execution of dependent programs. However, subsequent works show security loopholes in the definition and implementation of Java binary compatibility [Dea97], and even the lack of guarantee that existing code successfully links and executes with upgraded components [WDE98, DWE98]. Addressing the latter issue, Eisenbach et al. [ESS02] suggest to ensure correct upgrade in an interactive deployment environment that uses a dedicated linking policy and a set of rules to inform developers, whether the linked component version is compatible with the upgraded component. The component binaries themselves can be implemented in a way, which fosters their safe and efficient transformation [GLW95].

Basing on the Design-By-Contract paradigm [Mey92], des Rivières shows how changes of type contracts in Java APIs can potentially break existing applications [Riv01]. For different levels (package, class, interface, method, field) he enumerates, which API changes will not break applications, and under which conditions. Where applicable, for some application-breaking API changes he suggests appropriate solutions—workarounds [Riv01].

[2] In Java, such a proxy would implement all adaptation logic in a single method. In the method, a further check would be required to find the method implementation that should be invoked to serve a particular framework or plugin caller. This check would add yet another level of method redirection, comparing to the message forwarding as implemented in our adapters.

[3] `http://commons.apache.org/releases/versioning.html`

Mikhajlov and Sekerinski [MS98] investigate the fragile base class problem, in which (often, seemingly safe) changes of API classes reused by inheritance may break dependent applications. Based on their findings, the authors formally define conditions and prescribe a set of requirements to avoid the fragile base class problem when independently evolving components [MS98]. In practice, however, some of these requirements (e.g., never start or stop calling a new API method from an existing base API class) are usually too restrictive for adequately evolving component APIs.

In general, approaches of maintenance prescription are common in practice, and are useful. However, alone they are often highly restrictive in the way they guarantee evolving an API without danger for existing applications. In cases they warn about potentially dangerous changes, maintenance-prescriptive approaches cannot be used in a tool to automatically protect applications from such changes. Still, approaches of this group should be considered complimentary to other approaches, fostering together more appropriate API evolution.

#### Reuse prescription

To make documentation on component reuse more precise, a number of researchers suggest to explicitly capture the collaboration (the constraints and interactions) between components and their dependent application code, prescribing thus proper component reuse. One approach is to describe component interactions and constraints in plain language [KL92] or using design patterns [Joh92]. Another approach is to define them formally by describing the semantics of reuse via type constraints, possibly augmented with protocol specifications of the method exchange permitted among components [Hau93, HHG90, KKS96, Lam93, SG95, SLMD96, Mez97].

At the very least, such descriptions can serve as a structured documentation for systematic software reuse. In case of component evolution, they may also reflect the evolutionary component changes to be propagated to dependent applications. Reuse prescription facilitates such propagation via specifying (with a certain degree of formality) the applied component changes, and the work required to update and to test previously built applications. Formal descriptions can partially automate checking conformance of existing applications to the reuse specifications of upgraded components [Hau93, Lam93, SG95, SLMD96, Mez97]. Moreover, some formal approaches (e.g., [SLMD96, Mez97]) may guide developers in addressing detected inconsistencies manually or semi-automatically.

Except for the infamous design patterns [GHJV95], to the best of our knowledge, other approaches of this group are not common in practice. One reason is indicated by Minsky, who argues that the manual implementations of conformance rules are laborious, unreliable, and difficult to verify, and are also hard to maintain, because they can be invalidated by a change anywhere in the system [Min96]. In practice, developers are reluctant to provide and maintain, besides component implementations, formal specifications of component reuse. Instead of formally specifying component reuse, in some cases, citing Johnson, "it might be better to improve object-oriented languages so that they can express patterns of collaboration more clearly" [Joh97].

### 6.2.2 Prevention

A group of approaches relies on the use of a legacy middleware [COMa] or a specific communication protocol [FCDR95, MC02] to interconnect components and, in case of component change, to prevent dependent applications from observing component changes.

In Microsoft's Component Object Model (COM) [COMa], unique identifiers are used to mark the interfaces with version labels. If an interface is extended, the environment silently generates and registers a new interface with a new identifier, while also keeping the old one. A client call is then dispatched, considering the client's version, to the corresponding interface version. However, COM supports solely interfaces: no inheritance between libraries and applications is possible. This serious limitation even makes Forman et al. consider programming in COM as procedural, and not object-oriented [FCDR95]. Moreover, this approach

implies a middleware-dependent application development; that is, developers must use an interface definition language and data types of the middleware, and obey its communication protocols.

McGurren and Conroy [MC02] suggest to use reflection in combination with Java dynamic proxies to shield applications from the substitution of interface implementors in a software library. Thereby, they support transparent instance replacement adaptation, when instances of different classes implementing an interface can be exchanged transparently for existing applications. Similarly to COM, their work is limited to interfaces and requires in addition instantiating objects via factory methods. Moreover, since the interface expected by application code is fixed (only its implementing classes can be exchanged), changes can be performed only on the level of method implementations.

IBM's System Object Model (SOM) [FCDR95, FD99] is a library packaging technology that enables applications to share class libraries regardless of the language they were written in. Among its features, SOM pioneered the concept of release-to-release binary compatibility [FCDR95]. The core of the technology is an object model that defines classes as special objects and metaclass communications as object protocols with constraints. By considering the information about library changes and predefined constraints on the library's class hierarchy (expressed at the level of metaclasses), the object model automatically and transparently adapts existing applications to changes in the core system library. Most changes (in fact, refactorings) supported in SOM are typical for C libraries (e.g., reordering members, retracting private members, adding private instance variables) and do not pose problems in languages with dynamic linking, such as Java. Although SOM does support adapting a few other interesting refactorings (namely, inserting new classes in the class hierarchy, pushing up methods and pulling down classes along the class hierarchy), most of common application-breaking refactorings (e.g., refactorings affecting method signatures, moving methods) are not supported.

### 6.2.3 Facilitation

This group consists of approaches to distribute the change information (in the form of *adaptation specifications*) and facilitate the remote component update. The update is performed either intrusively (by directly modifying dependent source or binary code) or unintrusively (by creating adapters that translate between existing code and the upgraded component).

#### Intrusive facilitation

In the approach of Chow and Notkin [CN96], maintainers of a software library manually specify the changes in the library's API interfaces and, in addition, rules describing how to upgrade applications. Out of change and rule specifications, a migration tool can be generated, which rewrites applications to accommodate them for library changes. The authors discuss the adaptation of changed API interfaces only (no discussion of adapting changed API classes).

Keller and Hölzle [KH98] specify component change information programmatically in *delta files*, which are then compiled to a binary format. A delta file is consulted at application load time by a dedicated class loader that rewrites application binaries accommodating them for component changes. Besides specifying delta files, developers have to use an extended Java class loader that implements the required binary rewriting. For legal and security issues, developers often do not trust solutions that manipulate directly binary code, and are reluctant to use such solutions (as reported by Dig [Dig07, p. 131]).

Since its version 1.5, Java includes a type system that supports parametric polymorphism as *generic classes*. In case an existing software component is updated to use generic classes, its dependent applications should be updated correspondingly. Using pointer analysis, Donovan et al. [DKTE04] examine the updated component and its dependent applications, and automatically update applications to instantiate generic types of the

component's API. Their approach does not require manual specifications, but is focused on a particular change—upgrading applications to use generic classes.[4]

To promote existing Java applications using new, instead of obsolete, language features (e.g., *HashMap* instead of *Hashtable*, *Iterator* instead of *Enumeration*), Balaban et al. [BTF05] manually provide a mapping between legacy classes and their replacements. Considering the mapping, an analysis based on type constraints determines and updates the declarations and allocation sites in the legacy code to use the preferred language types.

Roock and Havenstein [RH02] use *refactoring tags*, which are based on Java annotations, to specify modifications applied to a framework directly in the framework's source code. Annotation-aware migration tools interpret the refactoring tags and guide application developers in migrating applications to use the changed framework API.

To accommodate a large base of Linux drivers for API changes in driver support libraries of the kernel, Padioleau et al. [PLHM08] propose a scripting language, in which rewriting patterns (so-called *semantic patches*) can be specified. A dedicated transformation engine uses rewriting pattern specifications to modify sources of device drivers making them correspond to the updated driver support libraries.

Several aforementioned intrusive approaches [CN96, KH98, RH02, PLHM08] are, in fact, not limited to particular changes, for which they can compensate. The main advantage of these approaches, also comparing to our approach, is their power: indeed, if developers manage to provide specifications required for adaptation, potentially any kind of change can be adapted. However, all aforementioned intrusive approaches, except for the approach of Donovan et al. [DKTE04], require adaptation specifications (e.g., code mappings, delta files, rewriting rules, or tags), which, at least in the original descriptions of the approaches, have to be provided manually. In case of large and complex components, the cumbersome task of writing adaptation specifications is expensive and error-prone. Moreover, except for the approach of Keller and Hölzle [KH98], intrusive approaches require recompilation of applications being updated. Most important, intrusive adaptation may be impossible at all, in case applications are unavailable for analysis and update, or software licenses forbid their changes. In addition, developers often refrain from using a solution that directly manipulates their code.

#### Unintrusive facilitation

Allen and Garlan [AG97] introduced the *architectural connector* as an explicit semantic entity that relates components in a component-based system. As argued by Yellin and Strom [YS97], an adapter can be seen as an architectural connector that bridges incompatibility between cooperating components. Under such assumptions, the adaptation is driven by the idea of (1) formally capturing important syntactic and behavioral component properties (to reason about signature and protocol compatibility, deadlock freedom, synchronization of distributed components, etc.), as well as (2) formally relating components in terms of the defined properties. The former results in component interface specifications and the latter in interface mapping specifications. Using these formal adaptation specifications, adapters can be (semi-)automatically derived and checked for correctness with regard to the defined properties.[5]

Inspired by the seminal paper of Yellin and Strom [YS97], a number of approaches use, for example, finite state machines [SR02a], label transition systems [CPS06], message system charts augmented with temporal logic specifications [IT03], or process algebra [BCP06] to provide interface and mapping specifications. All these approaches require that component developers write formal adaptation specifications. As a side remark, to the best of our knowledge, no research exists on specifying formal adaptation specifications in component-based systems, in which API classes are reused by inheritance.

[4] Strictly speaking, even if not updated to use generic classes, Java applications would recompile and run with the updated component. Therefore, whereas making old Java applications use generic types of the updated component improves the code quality, it is not required for backward compatibility preservation. By contrast, in .NET such application upgrade is required, because old applications would not otherwise recompile with the updated component that uses generic types.

[5] Although initially all such approaches were developed to support component *integration*, they could be similarly applied to support component *evolution*.

### 6.2.4 Approaches for Refactoring-Based Adaptation: CatchUp! and ReBA

In the line with Dig and Johnson [DJ05], we envisage a number of tools that will use refactoring information to help developers properly upgrade component-based applications. However, at the time of writing, to the best of our knowledge, there are only two refactoring-based approaches besides our technology: CatchUp! and ReBA.

#### Record-and-Replay in CatchUp!

As a future work, in his PhD thesis Roberts suggests that "the framework vendor could release a set of refactorings [applied to the framework] that clients could apply to their applications that would migrate their code to the new version of the framework" [Rob99, p. 99]. This idea of recording the refactorings applied to a software component for their later execution on dependent application code was explored by Henkel and Diwan [HD05]. Their tool CatchUp! [HD05], implemented as a plugin for the Eclipse IDE [Ecl], is able to listen for, capture, and record refactorings applied, while developers evolve a component in Eclipse. The log file containing recorded refactorings is then delivered to application developers, who "replay" refactorings on the application code to synchronize it with the refactored component. In case developers are interested only in certain recorded refactorings, an upgrade wizard allows to interactively select those log's refactorings that should be executed on the application.

In its implementation, using the descriptions of refactorings recorded in the log, CatchUp! recreates the set of refactoring objects that changed the original component. The recreated refactoring objects can then be executed on the application, exactly as any other (ordinary) refactoring objects. In particular, their execution is preceded by Eclipse code analysis, which collects information about the declarations and allocation sites to be modified by the refactoring.

For a realistic upgrade of a component-based application, CatchUp! has several important limitations. To begin with, since recreated refactoring operators execute on source code only, reflective method calls in existing applications are not updated and become invalid. Furthermore, if changing the signature of a component method introduces a signature conflict with some existing application-specific method (i.e., the signatures of both methods are the same), the code analysis performed for the corresponding recreated refactoring object will discover a name conflict. As a consequence, the re-execution of the recreated refactoring object will fail. Moreover, the authors do not consider the impact of API type and method addition on existing applications. Since such changes are not compensated for in CatchUp!, they may lead to the problems associated with accidental method overriding and type loading, due to collisions of type and method names defined in the refactored component and existing applications.[6] Furthermore, this intrusive way of updating requires available application sources and their recompilation, and implies a new application version for each component upgrade. Finally, similarly to other intrusive approaches, CatchUp! cannot be used in cases, when applications are unavailable for analysis and upgrade, or software licenses forbid their changes.

#### Binary Compatibility in ReBA

Similar to our work, in their adaptation tool ReBA Dig et al. [DNMJ08] aim for preserving binary backward compatibility of refactored software components (libraries and frameworks) with existing applications. To make an application that used an old component API execute with the upgraded component, they recover the part of the old API refactored in the latest API and *inline* the recovered API part into the upgraded component. Effectively, instead of putting a wrapper around the component, ReBA gives it two APIs (the old and the new).

To recover from an old component the part of its API refactored in the new component release, for several application-breaking API refactorings Dig et al. define compensating refactorings that preserve the old

[6] For software frameworks, we discussed in detail such problems, manifesting as Unintended Functionality and Type Capture, in Section 2.2.3.

component API and, whenever required, express its functionality in terms of the latest API [DNMJ08]. For example, for an original refactoring `RenameMethod`(*oldMethod*, *newMethod*) compensates a combination of the original refactoring renaming the method, and the refactoring `AddMethod`(*oldMethod*) that inserts *oldMethod* forwarding to *newMethod*. As another example, for `DeleteMethod`(*method*) compensates an empty transformation (refactoring), the execution of which preserves *method* in the component API.

As an input, ReBA takes an old component, the latest component, and the trace of API refactorings that occurred between the two components in the temporal order of their application to the component API. As a first adaptation step, ReBA executes the compensating refactorings on the old component in the same order as the original refactorings. Once all compensating refactorings are executed, the old component API contains recovered types and members that would have otherwise been refactored by the original API refactorings. Thereafter, ReBA copies all recovered program elements into a replica of the latest component. Finally, ReBA rolls back on the obtained component all original renaming API refactorings that are not compensated in the first adaptation step to preserve name dependencies among refactorings of the refactoring trace.

The power of ReBA results from its inlining nature. Since no additional classes are created, there is no object schizophrenia intrinsic to adapters. In particular, the original object identity is not affected by adaptation. As the method forwarding is limited to at most one additional redirection per method call (from old API methods to API methods with refactored signatures and locations), the performance penalties are reported to be less than 1%. Moreover, having access to the old component implementation, ReBA recovers and inlines deleted methods automatically, without requiring developers to specify method implementations.

As a technical limitation, ReBA is Eclipse-centric: its compensating transformations are ad-hoc rewrites of syntactic descriptors that express refactoring information in Eclipse. Moreover, the authors do not discuss support for refactorings of interface types. In addition, it is not clear, how API changes other than refactorings should be compensated, when using ReBA.

Paradoxically, also the main limitation of ReBA stems from its inlining nature. Because the old API and the new API are represented by the very same component, it is impossible to separate the two APIs in case of their conflict. Such conflict may occur at the level of a single API type or the component API as a whole. An example of a conflict at the level of an API type, discussed by Dig et al. [DNMJ08], is ReBA's inability to compensate for deleting an API method, say $M$, and then renaming another API method to $M$. Another, more significant example, is accidentally overriding a newly introduced API method by an existing application method (as we showed in Figure 2.10 on page 33 for Unintended Plugin Functionality: Method Capture). In fact, in this case the adapted API type suffers exactly from the problem, which made us decide for delegation (instead of inheritance) in our white-box class adapters:[7] namely, the inability of the type to selectively call defined and redefined methods, depending on the calling site. Finally, an example of a conflict at the level of the component API as a whole is accidentally loading a newly introduced component type, when an existing application type is expected (as we showed in Figure 2.11 on page 34 for Type Capture). In general, all such name conflicts between the old and the new APIs could be solved, if the APIs were exposed by separate (i.e., adapter and upgraded) components, but are not solvable in the presence of ReBA's inlining.

[7] We discussed in detail the advantages of using delegation instead of inheritance, when discussing the design rationale of white-box class adapters in Section 3.1.5 on page 60.

# 7 Conclusions and Future Work

The main goal of this thesis is helping framework developers to appropriately evolve framework APIs and to safely upgrade a framework to new, improved versions, while shielding existing applications from application-breaking API changes. We present an adaptation technology that supports refactoring-based binary backward-compatible framework upgrade, and is both rigorous and practical. It is rigorous in the sense that for application-breaking API refactorings we define precisely the structure of compensating adapters and the automatic derivation of these adapters based on the semantics of application-breaking API refactorings. The adaptation is also practical, because it addresses most application-breaking API changes—API refactorings—automatically, and because it requires neither manual adaptation nor recompilation of existing plugins. It is also simple, because it obviates the process of writing and maintaining adaptation specifications for application-breaking API changes not compensated for automatically. We believe, that our technology not only helps reducing the costs and improving the quality of framework upgrade, but also relaxes the constraints on the permitted API changes allowing for more appropriate framework evolution.

## 7.1 Summary of Contributions

In Chapter 1 we claimed a set of thesis contributions. In the following, we recapitulate these contributions, and show how and where they were accounted for in the thesis:

**C1: Empirical estimation of the general feasibility of refactoring-based adaptation.** In Section 2.1 we defined and conducted a case study investigating the evolution of framework APIs evolved without backward compatibility preservation with existing applications. Although such evolution could arguably lead to less refactorings applied comparing to a more conventional framework evolution [DJ05], we showed that even in such circumstances API refactorings account for more than 85% of all application-breaking changes applied during framework maintenance. In contrast to framework maintenance, in case of framework replacement the ratio of application-breaking API refactorings was insignificant. In general, our results allow for great optimism that refactoring-based adaptation should be useful in a large number of evolving software components (including software frameworks).

**C2: Insights into the mechanics of application-breaking API refactorings.** Considering Java frameworks, in Section 2.2 we showed by examples why certain API refactorings, although being behavior-preserving by definition, may break existing applications. We identified four main groups of problems possibly caused by such refactorings: Non-localizable Functionality, Missing Functionality, Unintended Functionality, and Type Capture. Even if we could recompile plugins after framework refactorings, the recompilation would not solve most problems of the four groups. Moreover, plugin recompilation would not even detect problems of Unintended Functionality, which are especially likely in the presence of white-box reuse and inversion of control typical for software frameworks [JF88]. Our improved understanding of the mechanics of application-breaking API refactorings allowed for a precise discussion on what an adaptation technology should provide to compensate for such refactorings. Finally, because we assumed unavailability of plugins for analysis and update, we argued for adapters as generally applicable adaptation means.

**C3: Adaptation design compensating for a refactored API type.** In Section 3.1 we introduced the Black-Box Class and Interface Adapter patterns, and the White-Box Class and Interface Adapter patterns as variants of the Adapter design pattern. The four patterns compensate for application-breaking API refactorings in the presence of black-box and white-box framework reuse, and inversion of control. Consistently with the original description of the Adapter pattern, we presented our patterns in a number of pattern sections, including motivating examples, and structure and collaboration descriptions. In our examples, we also showed how problems introduced by application-breaking API refactorings can be systematically solved by the patterns. For adaptation of Java frameworks, we explained thoroughly why exactly the proposed design of adapters was elaborated, and how black-box and white-box adapters support binary backward compatibility of a refactored API type and the corresponding old plugin types.

**C4: Adaptation design compensating for a refactored framework API as a whole.** While the Black-Box and White-Box Adapter patterns consider adapting a particular refactored API type, we also considered various aspects of adapting the refactored API as a whole. In Section 3.2 we elaborated on exhaustive API adaptation as a set of static and run-time decisions for supporting binary backward compatibility of a refactored framework and its existing plugins. For a plugin developed with an older framework version, at adapter generation time we create the corresponding static adaptation layer that reconstructs the framework API of that old version and connects types of the latest API and the old plugin. The adapters are deployed together with the upgraded framework and are placed between the framework and the plugin. At application execution time, the instantiated objects of the adaptation layer (i.e., wrappers) make up the run-time adaptation layer, through which the plugin and the framework communicate. Via a set of examples supported by algorithms used for adapter caching, for wrapping and unwrapping of framework and plugin objects, and for proper adapter instantiation we showed how to preserve type safeness in the presence of adapters.

**C5: Description of challenges, adopted solutions, and limitations in using adapters for refactored framework APIs.** Using adapters as adaptation means to compensate for API refactorings implied a set of challenges, some of which we could overcome, sometimes under certain additional restrictions, while others we had to accept as technology limitations. In Section 3.3 we showed by several examples how we solved object schizophrenia, and supported reflective calls and Java object serialization in the presence of adapters. We also realized or (in some cases) suggested as a future work several optimizations to decrease the adapters' performance overhead. Considering a set of common refactorings, we discussed why we could or could not compensate for them using the Adapter design pattern and our pattern variants thereof.

**C6: Rigorous refactoring-based adapter derivation.** In Chapter 4 we formalized the concept of a weak refactoring inverse as a program transformation inverting syntactic changes introduced by refactorings. Based on this definition, we formalized a comeback as a special kind of weak refactoring inverse that can be executed on adapters. The comeback definition enabled a precise description of a systematic adapter construction based on the history of application-breaking API refactorings. As a negative result, we showed the lack of an automated proof of chaining weak refactoring inverses for a given chain of refactorings, and discussed the origins of this limitation in the current state of the art of refactoring formalization. For several common refactorings, we defined and proved corresponding weak refactoring inverses and comebacks in Chapter 4 and Appendix D.

**C7: Practical tool for refactoring-based API adaptation.** In Chapter 5 we presented the prototypical implementation of our adaptation concepts—the generative logic-based tool ComeBack!. The tool supports transparent framework upgrade by connecting the upgraded framework and existing plugins via adapters, while separating the name spaces of the framework and plugins to avoid accidental collisions of their type names. In addition, ComeBack! supports side-by-side execution of plugins of different versions and the upgraded framework within the same application. Tool evaluation showed its ability to automatically compensate for most application-breaking API refactorings encountered in the study of Dig and Johnson [DJ05] and all such refactorings detected in our case study. The performance estimation of the generated adapters showed a run-time overhead we considered acceptable for a large number of adaptation scenarios. Although currently the changes beyond refactorings have to

be compensated manually, as an important contribution we presented an example of how protocol adaptation can be straightforwardly embedded into ComeBack!, supporting semi-automatic adaptation of changes other than refactorings.

## 7.2 Future Work

We envisage extensions of the work of this thesis in theoretical and practical directions.

### 7.2.1 Support for Other Languages and Language Features

An obvious direction of extending our adaptation technology is to apply it to other languages, such as .NET. Since every programming language has its specific set of features, refactorings are also language-specific. Supporting a new language will require defining a set of refactorings for its features. In turn, the way how refactorings are defined for a particular language feature will impact the way compensating adaptation transformation are defined. However, in case the metamodel of the language to be supported has considerable similarities with the metamodel of the language features currently supported in our adaptation formalization, our metamodel can be used as a core to be extended by features specific to the given target language. Moreover, since we are not bound to a particular refactoring engine and syntactic language particularities in our operational comeback definitions (as implemented in ComeBack!), we should be able to reuse comebacks for other languages, whenever comeback semantics are appropriate. In particular, in our follow-up work we will investigate, how comeback definitions initially specified for Java can be reused for adapting .NET frameworks.

In the adaptation context of Java frameworks, we formally defined comebacks for several common refactorings applied to a subset of Java language features. Although we do support other refactorings in our adaptation tool by implementing corresponding comebacks, their comeback definitions are not yet formally proved (e.g., in first-order predicate logic). This limitation is inherent in the state of the art of refactoring formalization, in which refactorings are defined only for a subset of features common for most object-oriented languages. To define the adaptation of refactorings applied to other Java language features (e.g., interfaces, enumerations, or annotations) would require formalizing the refactorings of those features first. In particular, a future work should investigate the opportunity of combining our approach with adaptation technologies supporting Java generic types [KETF07]. Moreover, a future work should investigate how to combine the adaptation of refactored user-defined API types, as supported in our approach, and the adaptation of evolved language library types [DKTE04, BTF05, KETF07], which we currently do not support.

### 7.2.2 Evaluation of the Technology Impact on Maintenance Costs and Activities

Since we do not place additional strain on framework and application developers to compensate for most application-breaking API changes (i.e., API refactorings), our technology saves costs of maintaining framework-based applications. The costs are saved at both sites, those of framework developers and application developers, because for compensated application-breaking API refactorings (1) framework developers do not have to write adaptation specifications or update patches to upgrade applications, and (2) application developers do not need to adapt applications manually. To estimate cost savings, we plan to compare the time and effort spent by a group of developers, who do not use our tool for framework upgrade, with a group of developers equipped with ComeBack!. We intend also to observe the developers' learning curve in using ComeBack!.

As another important empirical study, we plan to investigate the impact of the availability of our adaptation technology on the number and kind of evolutionary API changes. We expect that, not being afraid of breaking existing applications, framework developers will restructure APIs eagerly to maximize framework reusability.

### 7.2.3 Refactorings and Adaptation in Feature-Oriented Programming

Feature-oriented programming (FOP) is concerned with designing and implementing *features*—refinements in the product functionality [Pre97]. By modeling the *problem space* of a domain, a *feature model* defines all legal feature configurations. A particular configuration chosen by the user is used to generate a final product out of feature modules that make up the *solution space* [CE00].

Similar to refactoring, FOP can be seen as an architectural metaprogramming [Bat07]: feature composition modifies (by addition and extension) base programs using features. Several recent publications (e.g., [Bat07, BS07, KAT+08, TBD06, ŞH08]) point out various contexts where refactorings and FOP overlap. In particular, we are currently exploring the issues on refactoring software artifacts in FOP [ŞH08].

In FOP, refactoring means restructuring software artifacts of the solution space, the problem space, or both. The key issue in the refactoring of the two spaces is that their changes should not be considered in isolation; otherwise, seemingly safe changes may lead to wrongly composed final products. For example, consider refactoring the solution space that makes one feature module dependent on another module. If applied only to the solution space, this valid (with regard to the module implementation) refactoring is not reflected in the feature model as an additional constraint between the features involved. As a consequence, the model will permit a configuration that includes the depending feature without the feature it depends on. As another example, if a feature in the feature model is made optional, whereas other features depend on its functionality in their implementation, the feature may be missing in a configuration leading to an invalid final product implementation.

In general, many changes made to one space should propagate to the other space. More precisely, in case refactorings change constraints on the valid combination of features or feature modules, constraints of another space must be updated (adapted) correspondingly. Current approaches for automatically detecting inconsistencies of space constraints [CP06, TBKC07] are using SAT solvers, and require manually solving the detected inconsistencies. These approaches, however, are not appropriate in the context of agile refactoring of problem and solution spaces. Drawing analogies with conventional code refactoring, it would mean changing the program, recompiling it to detect possible problems and then solving the latter manually. Instead, when applying small incremental changes to features or their implementations, it would be preferable to immediately know, whether changes are safe and do not lead to space inconsistencies. Moreover, similar to a refactoring engine updating calls to a renamed method or prompting for a default value of a new parameter, some space inconsistencies could be fixed (semi-)automatically. We consider as a future work defining and realizing an interactive refactoring environment to foster safe refactoring of the problem and solution spaces, and semi-automatic space synchronization.

### 7.2.4 Gentle Framework Outsourcing

Although reusing a proprietary framework helps a company quickly develop quality applications, maintaining such a framework is time-consuming and resource-intensive. Facing the appearance of new commercial and open-source frameworks with richer functionality, the company must constantly extend and improve the functionality of its proprietary framework. Therefore, the company must invest considerably in software maintenance to keep its own framework competitive. The costs required may quickly get too high, especially for small-size and medium-size companies with limited resources. As an alternative, such a company may consider *software outsourcing* [Ulr02, p. 55]: replacing the whole or, at least, a part of the current functionality of their proprietary component by commercial or open-source components.

However, outsourcing is a white-box modernization—a resource-intensive process with high failure risk [SPL03, p. 9]. The risk is especially high, when the outsourcing is done in one big step, as a *big-bang software replacement* [RH06, p. 237]. In such cases, to keep existing applications running developers need to maintain the old version of the framework for the whole time of performing outsourcing. Besides additional costs of maintaining an old framework version, the task of migrating applications to use the outsourced framework is complex and error-prone, especially in case the implementation of the outsourced framework considerably deviates from the framework implementation before outsourcing. Moreover, the

users of the applications have to wait until the whole outsourcing is performed, and cannot benefit from improvements achieved at intermediate stages of outsourcing.

Instead, during framework outsourcing we want to keep applications running with the framework being outsourced. For that, we can provide adapters that shield the applications from the changes in the framework's API. Since this activity is an example of the so-called *gentle software migration* [RH06, p. 237], we call it *gentle framework outsourcing* [ŞA08].

As opposed to the big-bang replacement, in which refactorings are rare (as we discussed in Section 2.1.3 for the replacement of JHotDraw), in gentle framework outsourcing refactorings play a key role. When analyzing the framework and preparing it for outsourcing, refactorings help restructuring the framework's internals to decrease code coupling, increase code cohesion, and make valuable business decisions otherwise hidden in the legacy code easily comprehensible in the framework implementation. The main activity at this stage is refactoring to pattern [Ker04]—restructuring the implementation to follow general and domain-specific design patterns.

Refactoring is also the main activity when actually replacing a framework implementation with external components chosen for outsourcing. For example, replacing a method call to an equivalent "outsourced" method, refactoring a method's algorithm to use functionality of an external component, or replacing old types with more powerful types of an external framework can be modeled as refactorings. Using the history of API refactorings, ComeBack! can generate adapters that protect existing applications during outsourcing. Moreover, in outsourcing there is inevitably a need for internal adapters to wire together old and outsourced framework parts. These adapters can also be generated based on refactoring information, similarly to adapters that preserve applications. In case more sophisticated adapters are required (e.g., protocol adapters), either specific adaptation approaches can complement the refactoring-based adaptation, or the required functionality can be implemented in adapters by hand.

We consider applying gentle framework outsourcing in several academic case studies and, together with our industrial partners, in an industrial project. This will bring new input as well as developers for ComeBack!.

### 7.2.5 Progress in Formalizing and Implementing Refactorings

Since the introduction of the first tool for automated refactoring—the Refactoring Browser [Refa]—a number of refactoring tools were developed and successfully applied in practice. Using a refactoring tool enables the programmer to rapidly apply design changes to existing code without having to worry about accidentally introducing code errors. When applying the refactoring operation requested by the programmer, the refactoring tool checks that the program behavior is preserved according to the semantic rules of the object language. In Java, support for automated refactorings is provided by stand-alone tools and as a functionality of such popular Integrated Development Environments as Eclipse [Ecl], JetBrains' IntelliJ IDEA [Int], Borland's JBuilder [JBu] or Sun's NetBeans [Net].

Although indeed successful in practice, both open-source and commercial refactoring tools have been shown to contain numerous, potentially harmful bugs [SEM08, SVEM09, ST09, EESV].[1] In most cases, these bugs are due to certain language semantics overlooked by the developers while implementing the program analysis used in refactoring tools. As a consequence of such bugs in refactoring tools, a program can be incorrectly transformed even if applied refactorings are seemingly simple and safe, such as renaming a program entity [SEM08] or changing its accessibility [ST09].

Such bugs find their way into refactoring tools, including heavily tested commercial ones, not because of the negligence of the tools' programmers, but due to the complexity of the object language (e.g., Java) and the lack of its formal semantics. The main reason is indicated by Schäfer et al.: "Deriving the correct [refactoring] preconditions relies on a global understanding of the object language in which programs to be refactored are written, and has to account for all corner cases that might possibly lead to an incorrect refactoring. In a complex modern language like Java this is an arduous task even for very simple refactorings" [SVEM09]. Since Java is only weakly formalized (in a textual language specification), when

[1] For an impressive survey of such bugs identified in major Java refactoring tools we refer to Ekman et al. [EESV].

implementing a Java refactoring it is possible to misinterpret or simply overlook relevant language properties. Moreover, the lack of rigour in Java formalization prevents automatically checking the correctness of the implementation of refactorings, while manually written programs testing for such correctness are incomplete, for the same reason. As a consequence, many refactorings are specified incompletely, and most (if not all) refactoring tools are flawed [SEM08, SVEM09, ST09, EESV].

Since our refactoring-based adaptation technology is focused on compensating for application-breaking refactorings, its application depends directly on the quality of the original refactorings. Even if a comeback is proved to be correct with regard to the corresponding application-breaking refactoring, if the latter is flawed, the former will not fix the flaw. That is, if an original refactoring incorrectly transforms the framework due to a bug in the refactoring's specification or implementation, the corresponding comeback will not compensate for the damage caused by the bug. Furthermore, comebacks—the main transformations compensating for application-breaking API refactorings—are themselves refactorings; hence, their specifications may be incomplete and their implementations may be flawed as for any other refactorings.

All in all, the application in practice of our (and, probably any) refactoring-based approach will depend on the progress in formalizing refactorings and developing refactoring tools for conventional object-oriented programming languages, such as Java [Tip07, KETF07, DDGM07, SEM08, KS08, SVEM09, ST09, DME09, WST09].

# A

# RefactoringLAN: Two Framework API Versions and Plugin Types

In this appendix, we summarize RefactoringLAN—a set of examples of framework reuse and API refactorings we use throughout the thesis. It is inspired by the LAN simulation lab used at the University of Antwerp for teaching framework technologies [LAN, DRG+05]. The core of RefactoringLAN are types defined for modelling a computer network, with such associated concepts as the network itself, its nodes, packets, and maintenance utilities. While designing RefactoringLAN, in the trade-off between making it realistic and simple we preferred the simplicity. Introducing new types and members for each example would unnecessarily complicate the presentation; therefore, we defined as few API types and members as possible to be used in all thesis examples.[1] Each particular example should be considered as an excerpt from RefactoringLAN, with unnecessary details omitted.

RefactoringLAN consists of two parts: the framework API of the first version reused in a plugin, and the refactored framework API of the second version. For each of the two parts, in the following two sections we present a figure and a description summarizing involved types and members. The visual notation used in the figures is similar to UML class diagrams [Obj05]. However, since we are not concerned about depicting public fields, the field parts of the class boxes are omitted. Moreover, since we operate only with API methods, visibility modifiers are omitted. Arrows between class boxes point from the caller to the formal type of the callee as expected by the caller. For each object serving as a message receiver, its formal type (as expected by the caller) and actual type (i.e., the object's class) are specified in source comments.

As a final remark, because the framework's *Node* in both versions of the framework API specifies an abstract method, strictly speaking, the class is abstract and cannot be directly instantiated. However, in certain thesis examples, in which *Node*'s abstract method is not involved, considering the class concrete (and not abstract) simplifies the presentation. For convenience, in all excerpts from RefactoringLAN, in which *Node* does not contain an abstract method, the *Node* class should be considered concrete; otherwise, it should be considered abstract.

## A.1 RefactoringLAN: First Framework API Reused in Plugin

Figure A.1 shows the framework API of the first version reused in the plugin of the same version. API types are reused in the plugin both by object composition and by inheritance. Besides ordinary calls from the plugin to the framework, the framework also calls back methods implemented in the plugin. In the following, we list API types and plugin types, followed by a short description and (if applies) a list of supported API methods.

[1] RefactoringLAN should be considered as a collection of examples developed solely for demonstration purposes; all its examples are implemented in our ComeBack! tool and can be obtained from the tool's homepage at `http://comeback.sourceforge.net`.

### A.1.1 Framework API Types

- *IPacket*: an interface modelling network packets.
  - *whoCreated()* returns the node that created the packet.
- *Packet*: (implements *IPacket*) a class modelling network packets.
  - *Packet()* (constructor) accepts the node creating the packet.
  - *whoCreated()* returns the node that created the packet.
  - *setMessage()* sets the given string as the packet message.
  - *setTimestamp()* uses the internal clock to set the timestamp of the message.
- *EncryptedPacket*: (subclasses *Packet*) a class modelling encrypted network packets.
  - *setMessage()* overrides the inherited method to encrypt the message being saved.
- *PacketFactory*: a class for creating network packets.
  - *createPacket()* given a node and its identifier, returns a new packet. Internally, invokes the constructor of *Packet* and logs the identifier argument for tracing packet creators.
  - *packetSeenInNode()* returns a report about the occurrence of the given packet in the given node. Internally, calls the method *whoCreated()* specified in *IPacket*.
- *Report*: a class that encapsulates trace information relating network packets and nodes (since irrelevant for the presentation, its members are omitted).
- *INode*: an interface modelling network nodes.
- *Node*: (implements *INode*) a class modelling network nodes.
  - *getName()* returns the name of the node.
  - *getIdentifier()* returns the unique node identifier.
  - *getInfo()* returns the node description.
  - *getNodeLocation()* (abstract) returns the location of the node in the network.
  - *broadcast()* sends the given message using broadcasting communication strategy.
  - *print()* outputs the given message to the console.
  - *log()* writes a specific node content to a database.
- *Server*: (subclasses *Node*) a class modeling servers as specific network nodes.
- *Utils*: a class offering a set of utility methods (since irrelevant for the presentation, its API methods are omitted). The class acts as a caller invoking methods on framework and plugin instances. In all its method calls, the formal type of the callee as expected by the caller is an API type defined in the first framework version.

### A.1.2 Plugin Types

- *PluginEncryptedPacket*: (subclasses *EncryptedPacket*) a class for modelling encrypted network packets, the encryption of which is defined by the plugin (since irrelevant for the presentation, the class members are omitted).
- *SecurePacket*: (implements *IPacket*) a class modelling secure network packets.
  - *whoCreated()* returns the node that created the packet.

  - *securityInfo()* returns the description of the security policy as defined and used in the plugin packet.

- *SecureNode*: (subclasses *Node*) a class modelling secure network nodes.
  - *getNodeLocation()* implements the inherited abstract method to return the node location in the network.
  - *broadcast()* overrides the inherited method to encrypt the message before broadcasting.
  - *getDescription()* returns the description of the encryption algorithm as used in the plugin. Internally, makes a supercall to the *getIdentifier()* method defined in *Node*.

- *LAN*: a class modelling a local area network (since irrelevant for the presentation, its methods are omitted). The class acts as a caller invoking methods on framework and plugin instances. In all its method calls, the formal type of the callee as expected by the caller is an API type (in black-box framework reuse) or a subtype of an API type (in white-box framework reuse) defined in the first framework version.

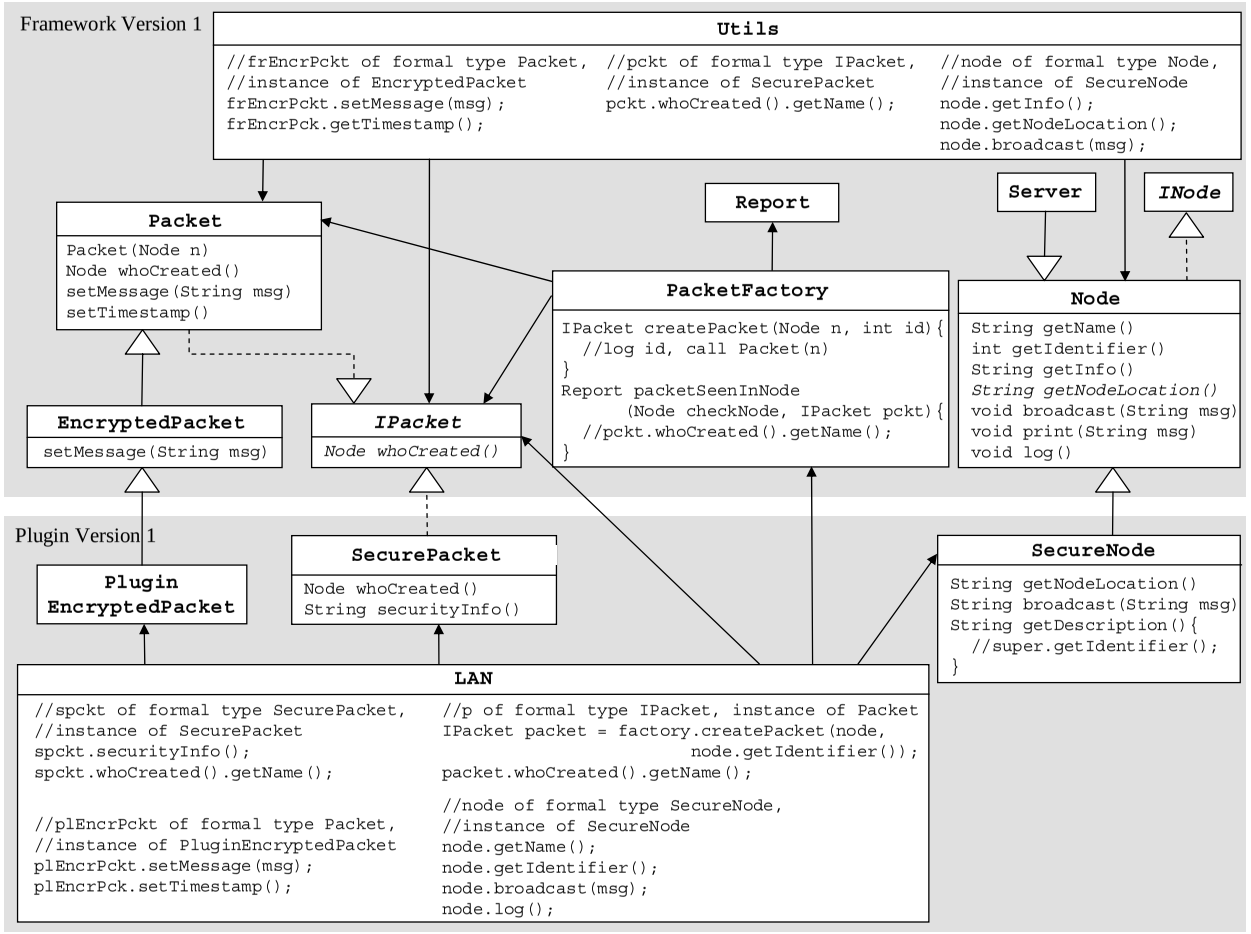



Figure A.1: RefactoringLAN (first framework version). The framework API of the first version is reused in the plugin of the same version by object composition and inheritance. The arrows show the relations between the callers and the callees.

## A.2 RefactoringLAN: Second, Refactored Framework API

Figure A.2 shows the second framework API resulting from the application of a number of refactorings to API types of the first framework version. In the following, we list the API refactorings applied between the first and the second framework versions, and denote exactly the affected API types and methods.

- `AddClass`: adds a new API class *SecurePacket* with two API methods *securityInfo()* and *getSecurity-Level()*. The former method returns the description of the security policy as defined and used in the framework packet. The latter method returns an integer indicating the current level of security used in the framework packet.
- `RenameMethod`: renames five API methods.
  - in *IPacket* and *Packet*: *whoCreated()* to *getCreator()*.
  - in *Node*: *getIdentifier()* to *getID()*, *getNodeLocation()* to *getLocation()*, and *broadcast()* to *manycast()*.
  - in *PacketFactory*: *packetSeenInNode()* to *tracePacket()*.
- `AddMethod`: adds a new API method *getDescription()* to *Node*. The method is semantically equivalent to *Node*'s *getInfo()* returning the node description.
- `RemoveParameter`: removes the second parameter (of formal type *int*) from the signature of the method *createPacket()* defined in *PacketFactory*. The parameter is redundant, since the node identifier can be obtained from the node, passed as the first method argument.
- `AddParameter`: adds the second parameter (of type *int*) to the renamed method *manycast()* in *Node*. The parameter allows to specify other communication strategies besides broadcasting (e.g., multicasting).
- `PushDownMethod`: moves the method *log()* from *Node* to its existing subclass *Server*. The refactoring addresses a specific framework requirement: only servers may log their content to a database.
- `ExtractSubclass`: subclasses *Node* with a new class *Workstation* and pushes down the method *print()*. The refactoring addresses a specific framework requirement: only workstations may print a message to the console.
- `AddMethodCall`: adds calls to the methods of the newly introduced class *SecurePacket*. Can be seen as a refactoring from the point of view of existing applications, in case they are not exposed to the effects of the introduced method calls.
- `ReplaceMethodCall`: update existing calls to invoke refactored methods. This is not a separate refactoring, but rather a transformation associated with other refactorings. In Figure A.2 we show it as a separate transformation, to ease tracing the updated framework call sites.

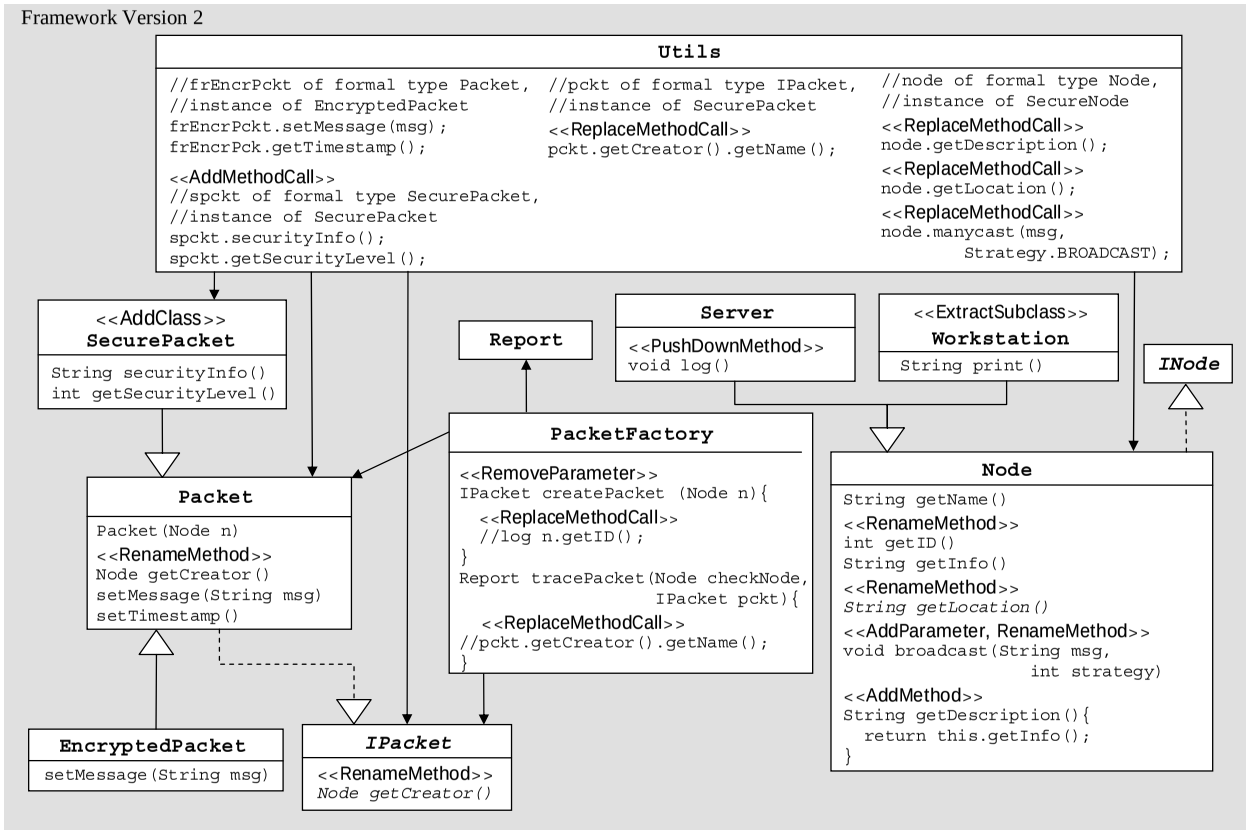



Figure A.2: RefactoringLAN (second framework version). To denote the applied API refactorings, all refactored types, members and method calls are annotated with the corresponding stereotypes.

# B

# Adaptation-Aware Exception Chaining and Object Serialization

## B.1 Handling Exception Chaining in Adapter Methods

When discussing the insights of an adapter method (Listing 3.3, page 77) we mentioned that library collections must be treated specially, when wrapping and unwrapping method arguments and return values in adapter methods. Otherwise, a *leakage* of objects of one domain (i.e., the framework or plugin) into the other domain may occur (as discussed furthermore in Section 3.3.2, page 93). However, library collections are not the only possible source of object leakage. Another language feature being dangerous in this regard is exception chaining, where an exception object can point to another exception object indicating the original exception cause. Basically, this is just another name for packing exceptions inside other exceptions. If a packed exception is of a user-defined API type, the refactoring of the latter may be as dangerous to existing callers as the refactoring of any other user-defined API type. Therefore, the packed exception object needs to be obtained from the enclosing exception object (similarly to obtaining an object from a library collection), and wrapped or unwrapped (similarly to wrapping and unwrapping of ordinary objects).

Listing B.1 shows the code snippet of an adapter method coping with exception chaining as implemented in Java. For simplicity, this code was omitted from Listing 3.3 on page 77 by assuming only at most one exception to be possibly thrown by a forwarding call to the wrapper's adaptee object. The helper functions **isWrapper** and **isUserDefined** were explained for Listing 3.3.

Listing B.1: Handling exception chaining in an adapter method

```
...
//forward to adaptee via reflection (details omitted)
Object value;
try {
  value = adaptee.methodName(arguments);
} catch(Throwable t) {    //unroll chain of exceptions into a list
  List<Throwable> chain = new ArrayList<Throwable>(3);
  Throwable temp = t;
  //starting with the first exception, go along the exception chain
  while (temp != null) {
    chain.add(temp);
    temp = temp.getCause();
  }
  //check each chained exception for need of wrapping or unwrapping
  boolean changed = false;
  for (int i = chain.size()-1; i >= 0; i--) {
    temp = chain.get(i);
    if (isWrapper(temp)) {
      unwrap(temp);
      changed = true;
    } else if (isUserDefined(temp)) {
      wrap(temp);
```

```
      changed = true;
    } else if (changed) {
      Throwable newTemp = temp.getClass().newInstance();
      newTemp.setStackTrace(temp.getStackTrace());
      newTemp.initCause(chain.get(i + 1));
      temp = newTemp;
    }
    chain.set(i, temp);
  }
  throw chain.get(0);
}

  if (value != null) {
...
```

## B.2 Custom Serialization in Adapters

The methods *writeReplace()* and *readResolve()* in Java allow to replace the object being serialized or deserialized with another object. For example, if a serialized database object is to be deserialized, while the content of the database changed, it is usually desirable to use the updated database information during deserialization. In this case, by implementing *readResolve()* it is possible to read the updated database information and use it for instantiating the object instead of the obsolete information previously serialized. The *writeReplace()* method works similarly, but for custom serialization. If such methods are present in the original API types, adapters must locate and invoke them to properly realize the serialization mechanism. Listing B.2 complements Listing 3.6 from page 92 with the code for object replacement supported in the custom adapter serialization.

Listing B.2: Adapter serialization in Java (complete algorithm)

```
/* adaptee is the adapter's field containing an instance of (framework or plugin)
    adaptee */
private void readObject(ObjectInputStream in) throws IOException {
    Class<?> clazz = ... //the adaptee type,
                         //filled in at adapter generation time
    Constructor<?> constructor;
    try {
        constructor = clazz.getConstructor();
    } catch (NoSuchMethodException nsme) {
        try { //no public constructor available, seek protected
            constructor = clazz.getDeclaredConstructor();
        } catch (Exception ex) {
            //neither public nor protected default constructor found
            //custom serialization mechanism flawed, critical failure
        }
    }
    Object toBeRead;
    try {
        toBeRead = constructor.newInstance();
    } catch (Exception ex) {
        //custom serialization mechanism flawed, critical failure
    }
    try {
        Method method = clazz.getDeclaredMethod("readObject",
                ObjectInputStream.class);
        method.setAccessible(true); //circumvent accessibility
        method.invoke(toBeRead, in);
        method.setAccessible(false); //can only be false
    } catch (NoSuchMethodException nsme) {
        //custom serialization not found, critical failure
    } catch (Exception ex) {
        //custom serialization mechanism flawed, critical failure
    }
```

```
    //find and call custom deserialization of readResolve
    try {
        Method method = clazz.getDeclaredMethod("readResolve");
        //since readResolve is found, obtain the replacement object
        boolean accessible = method.isAccessible();
        method.setAccessible(true); //circumvent accessibility
        toBeRead = method.invoke(toBeRead);
        method.setAccessible(accessible); //restore state
        if (toBeRead == null) {
            //custom serialization mechanism flawed, critical failure
        }
    } catch (NoSuchMethodException nsme) {} //no replacement object=>just continue
    //instantiate the adaptee field
    adaptee = toBeRead;
}

private void writeObject(ObjectOutputStream out) throws IOException {
    //start with the custom serialization of writeReplace
    Class<?> clazz = adaptee.getClass();
    Object toBeWritten = adaptee;
    try {
        Method method = clazz.getDeclaredMethod("writeReplace");
        //since writeReplace was found, obtain the replacement object
        boolean accessible = method.isAccessible();
        method.setAccessible(true); //circumvent accessibility
        toBeWritten = method.invoke(toBeWritten);
        method.setAccessible(accessible); //restore state
        if (toBeWritten == null) {
            //custom serialization mechanism flawed, critical failure
        }
        //set the type of the object to be serialized to the type of the replacement
            object
        clazz = toBeWritten.getClass();
    } catch (NoSuchMethodException nsme) {} //no replacement object=>just continue
    try {
        Method method = clazz.getDeclaredMethod("writeObject",
                ObjectOutputStream.class);
        method.setAccessible(true); //circumvent accessibility
        method.invoke(toBeWritten, out);
        method.setAccessible(false); //can only be false
    } catch (NoSuchMethodException nsme) {
        //custom serialization not found, critical failure
    } catch (Exception ex) {
        //custom serialization mechanism flawed, critical failure
    }
}
```

# C

# Introducing Adaptation Layer is Behavior-Preserving

In this appendix we formally prove that the adapter methods forwarding to the actual program methods (in our context, by program we mean the framework being adapted) do not change the semantics of the latter. As a consequence, the addition of an adaptation layer is behavior-preserving with regard to the original semantics of the program. To achieve such formalization, in the following the *denotational semantics* [Sto81] of a typical adapter method will be investigated. We start with a short overview of the denotational semantics as defined for Java in Section C.1 and then present the formal proof in Section C.2.

## C.1 Background on Java Denotational Semantics

The denotational semantics of a programming language combines the syntactic description of that language with a formalism of its semantics [Sto81]. The language syntax is specified with the help of a generating *grammar*, which is a set of rules of the form $V \rightarrow W$ (or $V ::= W$), called *productions*, that specify how words of the language can be derived. A production consists of *terminal* and *nonterminal* symbols, where a nonterminal can be thought of as a placeholder for a production and a terminal is a part of a word of the language. For our proof, it shall suffice to restrict grammars to be *context-free*, meaning that the left-hand side of all productions only consist of single nonterminal symbols ($V$) and the right-hand side ($W$) is a (possibly empty) string of terminals and nonterminals.

As our comeback technology has been developed for Java, for our proof we will use the dynamic denotational semantics for Java introduced by Alves-Foss and Lam [AFL99], which base on the Java Language Specification (JLS) [GJSB05] and the Java Virtual Machine Specification [LY99].

### C.1.1 Continuations and Semantic Functions

The denotational semantics for Java programs is defined in terms of an environment $\gamma \in \Gamma$, which represents the static and dynamic aspects of the execution engine together with related helper functions, and a store $\sigma \in \Sigma$ for representing the memory during program execution.

When evaluating a syntactic construct, **continuations** are functions pointing to the rest of the code that should be evaluated afterwards. This abstraction is used for the normal control flow as well as for exceptions.

- A command continuation $\theta : \Gamma \times \Sigma$ takes an environment $\gamma \in \Gamma$ and a store $\sigma \in \Sigma$ and returns an answer representing a modification.
- An expression continuation $\kappa : \mathcal{V} \times \mathcal{T} \times \Sigma \rightarrow \Sigma$ takes a value $v \in \mathcal{V}$, the type of the value $t \in \mathcal{T}$, and a store $\sigma \in \Sigma$ and returns a store based on the denotation of the evaluated program part.

**Operational semantic functions** define the relationship between a language construct and the execution time behavior:

- Command functions $\mathcal{C}[\![]\!] : \Gamma \times \theta \times \Sigma$ take the current environment $\gamma \in \Gamma$, a command continuation $\theta$, and the current store $\sigma \in \Sigma$ and return a continuation function. They are used for evaluating commands.
- Expression functions $\mathcal{E}[\![]\!] : \Gamma \times \kappa \times \Sigma$ take the current environment $\gamma \in \Gamma$, an expression continuation $\kappa$, and the current store $\sigma \in \Sigma$ and return a continuation function. They are used for evaluating expressions resulting in a value.

**Definitional semantic functions** are auxiliary to the actual execution in that they define the execution context:

- Value functions $\mathcal{V}[\![]\!] : \Gamma \to \mathcal{V} \times \mathcal{T}$ take the current environment $\gamma \in \Gamma$ and return a pair $(v,t)$ consisting of a value (a basic type or a string) $v \in \mathcal{V}$ and its type $t \in \mathcal{T}$. The **fst** and **snd** helper functions project onto the first and second element of the pair, respectively.

### C.1.2 Denotational Semantics Productions

This section contains those corrected and abridged production rules as initially defined by Alves-Foss and Lam [AFL99] that are used in the proof of the following Section C.2. A question mark above right a production rule name indicates optionality, that is, the input can match the rule but does not need to.

A return statement first causes the expression to be evaluated and the result to be stored in the environment. Then the command continuation function recorded in the environment when invoking the method that should be returned from is retrieved using the *getComCont* function and evaluated. The $\textbf{promote}(\tau_1,(r,\tau))$ function performs numeric promotion of the given value $r$ of the given type $\tau$ to the desired type $\tau_1$ according to the rules of the JLS [GJSB05]. *&return*, *&returnVal*, and *&returnType* are auxiliary variables used to record nesting, scoping, and control flow information in the environment.

$\mathcal{C}[\![$<RetStmt>$]\!]\gamma\theta\sigma ::=$
$\quad \mathcal{C}[\![\textbf{return}$ <Expr>;$]\!]\gamma\theta\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
$\quad\quad \forall r,\tau,\sigma.\kappa(r,\tau,\sigma) = \theta_1(\gamma_1,\sigma)$ where
$\quad\quad\quad \theta_1 = \gamma.getComCont(\&return)$ and
$\quad\quad\quad \tau_1 = \gamma[\&returnType]$ and
$\quad\quad\quad r_1 = \textbf{promote}(\tau_1,(r,\tau))$ and
$\quad\quad\quad \gamma_1 = \gamma[\&returnVal \leftarrow r_1]$

A method invocation expression first causes the method arguments to be evaluated. Alves-Foss and Lam do not define the **getSigs** function but it seems safe to assume that it returns a representation of the method signature. The value $r$ holds the list of arguments as prepared by the <ArgList> production rule. That signature and the method name are used to look up the command continuation function for evaluating the statements in the method body with the help of the *getMethod* function of the environment.

$\mathcal{E}[\![$<MethodInv>$]\!]\gamma\kappa\sigma ::=$
$\quad \mathcal{E}[\![$<Name>(<ArgList>$^?$)$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<ArgList>$]\!]\gamma\kappa_1\sigma$ where
$\quad\quad \forall r,\tau,\sigma_1.\kappa_1(r,\tau,\sigma_1) = m(\gamma\theta\sigma_1)$ where
$\quad\quad\quad sig = \textbf{getSigs}(r)$ and
$\quad\quad\quad m = \gamma.getMethod(\textbf{fst}\mathcal{V}[\![$<Name>$]\!]\gamma, sig)$ and
$\quad\quad\quad \forall\gamma_2,\sigma_2.\theta(\gamma_2,\sigma_2) = \kappa(\gamma_2[\&returnVal],\gamma_2[\&returnType],\sigma_2)$

An argument list contains arguments that are evaluated from left to right. The **append** function appends the given lists.

$\mathcal{E}[\![$<ArgList>$]\!]\gamma\kappa\sigma ::=$
  [ $\mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ ]
| $\mathcal{E}[\![$<ArgList$_1$>,<Expr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<ArgList$_1$>$]\!]\gamma\kappa_1\sigma$ where
  $\forall r_1,\tau_1,\sigma_1.\kappa_1(r_1,\tau_1,\sigma_1) = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa_2\sigma_1$ where
  $\forall r_2,\tau_2,\sigma_2.\kappa_2(r_2,\tau_2,\sigma_2) = \kappa(q,\tau,\sigma_2)$ where
  $q = \mathbf{append}(r_1,r_2)$ and
  $\tau = \tau_1 + \tau_2$

For our purpose, it is sufficient to look at simple names and ignore qualified names.

$\mathcal{V}[\![$<Name>$]\!]\gamma ::=$
  $\mathcal{V}[\![$<SimpleName>$]\!]\gamma$

A simple name is made up of an identifier, which is typically represented by a special terminal symbol (here, typeset in italics). The reason for using a single placeholder terminal symbol for all identifiers is that a context-free grammar cannot verify the validity of the identifier's occurrence. For example, checking whether an identifier has been declared before it is used cannot be done by the parser that enforces the grammar but has to be done by the compiler traversing the abstract syntax tree.

The evaluation of a simple name will produce a pair containing the value of the name and the type "name". Unfortunately, Alves-Foss and Lam do not provide a definition for the *ValueOf* function.

$\mathcal{V}[\![$<SimpleName>$]\!]\gamma ::=$
  $\mathcal{V}[\![Id]\!]\gamma = (\text{ValueOf}(Id), \text{“}name''\text{)}$

A method body is a single block of statements.

$\mathcal{C}[\![$<MethodBody>$]\!]\gamma\theta\sigma ::=$
  $\mathcal{C}[\![$<Block>$]\!]\gamma\theta\sigma$

A block contains a list of statements in curly braces represented by a block statement list.

$\mathcal{C}[\![$<Block>$]\!]\gamma\theta\sigma ::=$
  $[\![\{$<BlockStmtList>$^?\}]\!]\gamma\theta\sigma = \mathcal{C}[\![$<BlockStmtList>$]\!]\gamma\theta\sigma$

A block statement list evaluates the contained block statements from left to right.

$\mathcal{C}[\![$<BlockStmtList>$]\!]\gamma\theta\sigma ::=$
  $\mathcal{C}[\![$<BlockStmt>$]\!]\gamma\theta\sigma$
| $\mathcal{C}[\![$<BlockStmtList$_1$><BlockStmt>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<BlockStmtList$_1$>$]\!]\gamma\theta_1\sigma$ where
  $\forall\gamma_1,\sigma_1.\theta_1(\gamma_1,\sigma_1) = \mathcal{C}[\![$<BlockStmt>$]\!]\gamma_1\theta\sigma_1$

$\mathcal{C}[\![$<BlockStmt>$]\!]\gamma\theta\sigma ::=$
  $\mathcal{C}[\![$<Stmt>$]\!]\gamma\theta\sigma$

The following production rule has been shortened to only show the evaluation of expression statements.

$\mathcal{C}[\![\text{<Stmt>}]\!]\gamma\theta\sigma ::=$
$\quad \mathcal{C}[\![\text{<ExprStmt>}]\!]\gamma\theta\sigma$

Expression statements are expressions that can also act as statements, e.g., method invocations.

$\mathcal{C}[\![\text{<ExprStmt>}]\!]\gamma\theta\sigma ::=$
$\quad \mathcal{C}[\![\text{<Expr>}]\!]\gamma\theta\sigma = \mathcal{E}[\![\text{<Expr>}]\!]\gamma\kappa\sigma$ where
$\quad\quad \forall r,\tau,\sigma_1.\kappa(r,\tau,\sigma_1) = \theta(\gamma,\sigma_1)$

## C.2 Proof of an Adapter Method Not Affecting API Semantics

To show that the addition of the adapters constituting the adaptation layer does not change the semantics of the program (i.e., the framework, in our case), we formulate and prove a supporting theorem. The informal description of the proof follows its formal description, while the production rules used were introduced in the previous Section C.1.2.

**Theorem 5** (Denotational Semantics of Adapters)
The adaptation layer does not change the denotational semantics of the framework API.

*Proof.* Let $\texttt{methodAPI}(\tau_1,\ldots,\tau_n):\tau_r$ be the original API method called with $\tau_1,\ldots,\tau_n \in \mathcal{T}$ being the parameter types, $\texttt{a}_1,\ldots,\texttt{a}_n$ the corresponding arguments, and $\tau_r \in \mathcal{T}$ being the return type; `methodAD` the corresponding forwarding adapter method with the same parameters and return type; the environment $\gamma \in \Gamma$ and the store $\sigma \in \Sigma$ the state of the execution environment; and $\kappa : \mathcal{V} \times \mathcal{T} \times \Sigma \to \Sigma$ an expression continuation; then $\mathcal{E}[\![\texttt{methodAD}(\texttt{a}_1,\ldots,\texttt{a}_n);]\!]\gamma\kappa\sigma = \mathcal{E}[\![\texttt{methodAPI}(\texttt{a}_1,\ldots,\texttt{a}_n);]\!]\gamma\kappa\sigma$ (the right-hand side lists the production rules applied):

$\quad \mathcal{E}[\![\texttt{methodAD}(\texttt{a}_1,\ldots,\texttt{a}_n);]\!]\gamma\kappa\sigma$
$= \mathcal{E}[\![\texttt{a}_1,\ldots,\texttt{a}_n]\!]\gamma\kappa_1\sigma$ where $\forall r,\tau,\sigma_1.\kappa_1(r,\tau,\sigma_1) = m(\gamma\theta\sigma_1)$ where <MethodInv>
$\quad\quad m = \gamma.\mathit{getMethod}(\texttt{methodAD}, \textbf{getSigs}(r))$ and
$\quad\quad \forall\gamma_2,\sigma_2.\theta(\gamma_2,\sigma_2) = \kappa(\gamma_2[\&\mathit{returnVal}],\tau_r,\sigma_2)$
$= \mathcal{E}[\![\texttt{a}_1,\ldots,\texttt{a}_{n-1}]\!]\gamma\kappa_2\sigma$ where $\forall r_1,\tau_1,\sigma_1.\kappa_2(r_1,\tau_1,\sigma_1) = \mathcal{E}[\![\texttt{a}_n]\!]\gamma\kappa_3\sigma_1$ where <ArgList>
$\quad\quad \forall r_2,\tau_2,\sigma_2.\kappa_3(r_2,\tau_2,\sigma_2) = \kappa_1(q,\tau,\sigma_2)$ where
$\quad\quad q = \textbf{append}(r_1,r_2)$ and $\tau = \tau_1 + \tau_2$
$= \ldots$ <ArgList>
$= \mathcal{E}[\![\texttt{a}_1]\!]\gamma\kappa_n\sigma$
$= m(\gamma\theta\sigma)$ <MethodBody>
$= \mathcal{C}[\![\{\texttt{return methodAPI}(\texttt{a}_1,\ldots,\texttt{a}_n);\}]\!]\gamma\theta\sigma$ <Block>
$= \mathcal{C}[\![\texttt{return methodAPI}(\texttt{a}_1,\ldots,\texttt{a}_n);]\!]\gamma\theta\sigma$ <BlockStmtList>
$= \mathcal{E}[\![\texttt{methodAPI}(\texttt{a}_1,\ldots,\texttt{a}_n);]\!]\gamma\kappa\sigma$ where $\forall r,\tau,\sigma.\kappa(r,\tau,\sigma) = \theta_1(\gamma_1,\sigma)$ where <RetStmt>
$\quad\quad \theta_1 = \gamma.\mathit{getComCont}(\&\mathit{return})$ and $r_1 = \textbf{promote}(\tau_r,(r,\tau))$ and
$\quad\quad \gamma_1 = \gamma[\&\mathit{returnVal} \leftarrow r_1]$

The first derivation step uses the production rule <MethodInv> for method invocation expressions. The new expression continuation $\kappa_1$ continues the derivation with the method body, once all method arguments have been evaluated using the <ArgList> production. The latter is applied recursively until all argument

expressions have been evaluated and incorporated into the store $\sigma_1$. After applying the <MethodBody> statement production rule the <Block> and the <BlockStmtList> statement productions are used. The derivation stops after the application of the <RetStmt> production, which results in a method invocation expression of a method of the new API. □

Since the proof applies to an adapter as an abstract specification (as discussed in Section 4.2 on page 111), in effect it implies that the semantics of the methods of the black-box and white-box executable adapters generated from the adapter specification do not change the framework's semantics.

# D

# Proofs of Weak Refactoring Inverses and Comebacks For Several Common Refactorings

In the following, we give definitions and proofs of weak refactoring inverses and comebacks for several common refactorings, which, when applied to a component API, may break existing applications dependent on that component. As a result, these transformations enable the creation of an adaptation layer representing the previous API version (before being affected by the refactorings) and so pave the way for refactoring-based adaptation. For completeness, we also repeat here the definitions of refactorings, weak refactoring inverses, and comebacks used as running examples in Chapter 4. The notations used were introduced in Section 4.3.5.

## D.1 `AddEmptyClass` and `RemoveEmptyClass`

The `AddEmptyClass` refactoring adds an empty, unreferenced class to the program (adopted from Roberts [Rob99, p. 103], where it is called `AddClass`).

`AddEmptyClass`(*class*, *super*, *sub*)

$$\begin{aligned}
&\textit{pre} : \text{IsClass}(\textit{super}) \wedge && (1.1)\\
&\quad \neg\text{IsClass}(\textit{class}) \wedge && (1.2)\\
&\quad \forall c \in \textit{sub}.(\text{IsClass}(c) \wedge \text{Superclass}(c) = \textit{super}) && (1.3)
\end{aligned}$$

---

$$\begin{aligned}
&\textit{post} : \text{IsClass}' = \text{IsClass}[\textit{class}/\textit{true}] && (1.4)\\
&\quad \text{ClassReferences}' = \text{ClassReferences}[\textit{class}/\emptyset] && (1.5)\\
&\quad \text{Superclass}' = \forall c \in \textit{sub}.\text{Superclass}[\textit{class}/\textit{super}][c/\textit{class}] && (1.6)\\
&\quad \text{InstanceVariablesDefinedBy}' = \text{InstanceVariablesDefinedBy}[\textit{class}/\emptyset] && (1.7)
\end{aligned}$$

The `RemoveEmptyClass` refactoring removes an empty, unreferenced class from the program (adopted from Roberts [Rob99, p. 104], where it is called `RemoveClass`).

`RemoveEmptyClass`(*class*)

$$\begin{aligned}
&\textit{pre} : \text{IsClass}(\textit{class}) \wedge && (1.8)\\
&\quad \text{ClassReferences}(\textit{class}) = \emptyset \wedge && (1.9)\\
&\quad \text{IsEmptyClass}(\textit{class}) && (1.10)
\end{aligned}$$

---

$$post: \text{IsClass}' = \text{IsClass}[class/false] \tag{1.11}$$
$$\text{ClassReferences}' = \forall c \neq class.\forall s.\text{ClassReferences}[class/\uparrow] [c/\text{ClassReferences}(c) \setminus \{(class,s)\}] \tag{1.12}$$
$$\text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[class/\uparrow] [c/\text{Superclass}(class)] \tag{1.13}$$
$$\text{ReferencesToInstanceVariable}' = \forall v.\forall c.\forall s. \text{ReferencesToInstanceVariable}[(class,v)/\uparrow] [(c,v)/\text{ReferencesToInstanceVariable}(c,v) \setminus \{(class,s)\}] \tag{1.14}$$
$$\tag{1.15}$$

Composing the `RemoveInstanceVariable`, `RemoveMethod`, and `RemoveEmptyClass` refactorings yields the `RemoveClass` refactoring for removing non-empty, unreferenced classes:

$$\begin{aligned}&\texttt{RemoveClass}(class) := \langle \\ &\quad \forall v \in \text{InstanceVariablesDefinedBy}(class).\texttt{RemoveInstanceVariable}(class,v), \\ &\quad \forall m \in \{m|\text{Method}(class,m)\downarrow\}.\texttt{RemoveMethod}(class,m), \\ &\quad \texttt{RemoveEmptyClass}(class)\rangle.\end{aligned}$$

**Theorem 6** (`InvAddEmptyClass` I)
The weak refactoring inverse for the `AddEmptyClass` refactoring, `InvAddEmptyClass`, is the `RemoveEmptyClass` refactoring:

$$\texttt{InvAddEmptyClass}(class, superclass, subclasses) := \texttt{RemoveEmptyClass}(class).$$

*Proof.*

1. `InvAddEmptyClass` is behavior-preserving by definition, because it consists of the single refactoring `RemoveEmptyClass`.

2. $post_{\texttt{AddEmptyClass}}(P) \models pre_{\texttt{InvAddEmptyClass}} \Leftrightarrow$
   $post_{\texttt{AddEmptyClass}}(P) \models pre_{\texttt{RemoveEmptyClass}}$:
   Let $I'$ be the interpretation gained by applying the `AddEmptyClass` refactoring to the source code $S$ of the program $P$ reflected by $P' = post_{\texttt{AddEmptyClass}}(P)$:

   $$I' = \left(trans_{\texttt{AddEmptyClass}}(S), \cdot^I\right).$$

   Then $pre_{\texttt{RemoveEmptyClass}}$ evaluates to true under $I'$:

   $$\begin{aligned}&[\text{IsClass}(class)]^{I'} \wedge^* [\text{ClassReferences}(class)]^{I'} = \emptyset \wedge^* [\text{IsEmptyClass}(class)]^{I'} \\ &= [\text{IsClass}(class)]^{I\{1.4:\text{IsClass}[class/true]\}} \wedge^* \\ &\quad [\text{ClassReferences}(class)]^{I\{1.5:\text{ClassReferences}[class/\emptyset]\}} = \emptyset \wedge^* \\ &\quad \nexists s.[\text{Method}(class,s)]^{I'}\downarrow \wedge^* \\ &\quad \nexists v.v \in [\text{InstanceVariablesDefinedBy}(class)]^{I\{1.7:\text{InstanceVariablesDefinedBy}[class/\emptyset]\}} \\ &= \top \wedge^* \top \wedge^* \top \wedge^* \top = \top\end{aligned}$$

   The second-to-last truth value can be derived from Invariant 2 (page 122) and the fact that the `AddEmptyClass` refactoring does not add any methods to the new class. More formally, it can be shown that the formula $\forall s.\text{Method}(class,s)\uparrow$ is a refactoring invariant for the `AddEmptyClass` refactoring.

3. $post_{\texttt{InvAddEmptyClass}}(post_{\texttt{AddEmptyClass}}(P)) \models pre_{\texttt{AddEmptyClass}} \Leftrightarrow$
$post_{\texttt{RemoveEmptyClass}}(post_{\texttt{AddEmptyClass}}(P)) \models pre_{\texttt{AddEmptyClass}}$:
Let $I''$ be the interpretation gained by applying the `InvAddEmptyClass` inverse refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = post_{\texttt{InvAddEmptyClass}}(P')$:

$$I'' = \left(trans_{\texttt{RemoveEmptyClass}}(S'), \cdot^{I}\right).$$

The refactoring invariant of the `AddEmptyClass` refactoring is

$$inv(\texttt{AddEmptyClass}) = \{\text{IsClass}(super), \forall c \in sub.\text{IsClass}(c)\},$$

and the invariant of the refactoring chain $\langle \texttt{AddEmptyClass}, \texttt{RemoveEmptyClass} \rangle$ is

$$inv(\langle \texttt{AddEmptyClass}, \texttt{RemoveEmptyClass} \rangle) = \{\text{IsClass}(super), \forall c \in sub.\text{IsClass}(c)\}.$$

Then $pre_{\texttt{AddEmptyClass}}$ evaluates to true under $I''$ and the invariant:

$$\begin{aligned}
&[\text{IsClass}(super)]^{I''} \wedge^* \neg^* [\text{IsClass}(class)]^{I''} \wedge^* \\
&\forall c \in sub. \left( [\text{IsClass}(c)]^{I''} \wedge^* [\text{Superclass}(c)]^{I''} = superclass \right) \\
&= [\text{IsClass}(super)]^{I'\{\text{Inv}\}} \wedge^* \neg^* [\text{IsClass}(class)]^{I'\{1.11:\text{IsClass}[class/false]\}} \wedge^* \\
&\forall c \in sub. \left( [\text{IsClass}(c)]^{I'\{\text{Inv}\}} \wedge^* \right. \\
&\left. [\text{Superclass}(c)]^{I'\{1.13:\text{Superclass}[c/\text{Superclass}(class)]\}} = superclass \right) \\
&= \top \wedge^* \neg^* \bot \wedge^* (\top \wedge^* \top) = \top.
\end{aligned}$$

□

Although the `RemoveEmptyClass` refactoring is a weak inverse for the `AddEmptyClass` refactoring, it is not a comeback, because it cannot always be executed on adapters: although the framework class is empty, the generated adapter will have an adaptee field if it is at the root of an adapter hierarchy, violating thus the precondition of the `RemoveEmptyClass` refactoring. The adaptee field—if defined in the adapter and not inherited by it—cannot be simply removed, it must be pushed down to all subclasses of the base adapter, if there are any. The `RemoveEmptyAdapter` refactoring addresses this issue:

$$\begin{aligned}
&\texttt{RemoveEmptyAdapter}(class) := \langle \\
&\quad \forall v \in \{v | v \in \text{InstanceVariablesDefinedBy}(class) \wedge v = \text{“adaptee”}\}. \\
&\quad\quad \texttt{PushDownInstanceVariable}(\text{Subclasses}(class), v), \\
&\quad \texttt{RemoveEmptyClass}(class) \rangle.
\end{aligned}$$

The adaptee field will always be removed from the adapter before the `RemoveEmptyClass` refactoring is executed, even if there are no subclasses into which the field can be pushed down. This is due to the `PushDownInstanceVariable` refactoring (formally specified in Appendix E.3) using universal quantification internally, so that it can also be applied to a class without subclasses.

The `RemoveEmptyAdapter` refactoring also qualifies as a weak inverse for the `AddEmptyClass` refactoring, but additionally it can be executed on adapters. As a weak refactoring inverse, it will conceptually always be executed in a context created by a previous execution of the `AddEmptyClass` refactoring. Although its precondition is theoretically stronger than that of the `RemoveEmptyClass` refactoring, it is not in practice, because the removed classes will always be empty. Note that the universal quantification works similar to an `if-then` construct: it is successfully executed if the quantified set is empty. As a comeback, when executed on adapters, the `RemoveEmptyAdapter` refactoring will take care of pushing down the adaptee field to all subclasses.

**Corollary 4** (`RemoveEmptyAdapter` I)
The `RemoveEmptyAdapter` refactoring is a comeback of the `AddEmptyClass` refactoring.

**Theorem 7** (`InvRemoveEmptyClass` I)
The weak refactoring inverse for the `RemoveEmptyClass` refactoring, `InvRemoveEmptyClass`, is the `AddEmptyClass` refactoring:

$$\texttt{InvRemoveEmptyClass}(\mathit{class}) := \texttt{AddEmptyClass}(\mathit{class}, \text{Superclass}(\mathit{class}), \text{Subclasses}(\mathit{class})).$$

*Proof.* This proof is very similar to the one of Theorem 6.

1. `InvRemoveEmptyClass` is behavior-preserving by definition, because it consists of the single refactoring `AddEmptyClass`.

2. $\mathit{post}_{\texttt{RemoveEmptyClass}}(P) \models \mathit{pre}_{\texttt{InvRemoveEmptyClass}} \Leftrightarrow$
   $\mathit{post}_{\texttt{RemoveEmptyClass}}(P) \models \mathit{pre}_{\texttt{AddEmptyClass}}$:
   Let $I'$ be the interpretation gained by applying the `RemoveEmptyClass` refactoring to the source code $S$ of the program $P$ reflected by $P' = \mathit{post}_{\texttt{RemoveEmptyClass}}(P)$

   $$I' = \left(\mathit{trans}_{\texttt{RemoveEmptyClass}}(S), \cdot^I\right),$$

   and let $Z$ be the variable assignment induced by theorem 7

   $$Z = \{\mathit{class} \mapsto \mathit{class}, \mathit{super} \mapsto \text{Superclass}(\mathit{class}), \mathit{sub} \mapsto \text{Subclasses}(\mathit{class})\}.$$

   Then $\mathit{pre}_{\texttt{AddEmptyClass}}$ evaluates to true under $I'$ and $Z$:

   $$\begin{aligned}
   &[\text{IsClass}(\mathit{super})]^{I',Z} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I',Z} \wedge^* \\
   &\forall c \in \mathit{sub}^Z.([\text{IsClass}(c)]^{I',Z} \wedge^* [\text{Superclass}(c)]^{I',Z} = \mathit{super}) \\
   =\ &[\text{IsClass}(\text{Superclass}(\mathit{class}))]^{I'} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I'} \wedge^* \\
   &\forall c \in \text{Subclasses}(\mathit{class}).([\text{IsClass}(c)]^{I'} \wedge^* [\text{Superclass}(c)]^{I'} = \text{Superclass}(\mathit{class})) \\
   =\ &\top \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I\{1.11:\text{IsClass}[\mathit{class}/\mathit{false}]\}} \wedge^* \\
   &\forall c \in \text{Subclasses}(\mathit{class}). \Big([\text{IsClass}(c)]^{I\{\text{Inv 5}\}} \wedge^* \\
   &[\text{Superclass}(c)]^{I\{1.13:\text{Superclass}[c/\text{Superclass}(\mathit{class})]\}} = \text{Superclass}(\mathit{class})\Big) \\
   =\ &\top \wedge^* \neg^* \bot \wedge^* (\top \wedge^* \top) = \top
   \end{aligned}$$

   The first and the last truth value can be derived from Invariant 5 (page 122) and the fact that the `RemoveEmptyClass` refactoring does not remove the superclass of the removed class.

3. $\mathit{post}_{\texttt{InvRemoveEmptyClass}}(\mathit{post}_{\texttt{RemoveEmptyClass}}(P)) \models \mathit{pre}_{\texttt{RemoveEmptyClass}} \Leftrightarrow$
   $\mathit{post}_{\texttt{AddEmptyClass}}(\mathit{post}_{\texttt{RemoveEmptyClass}}(P)) \models \mathit{pre}_{\texttt{RemoveEmptyClass}}$:
   Let $I''$ be the interpretation gained by applying the `InvRemoveEmptyClass` inverse refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = \mathit{post}_{\texttt{InvRemoveEmptyClass}}(P')$:

   $$I'' = \left(\mathit{trans}_{\texttt{AddEmptyClass}}(S'), \cdot^I\right).$$

   Then $\mathit{pre}_{\texttt{RemoveEmptyClass}}$ evaluates to true under $I''$:

   $$\begin{aligned}
   &[\text{IsClass}(\mathit{class})]^{I''} \wedge^* [\text{ClassReferences}(\mathit{class})]^{I''} = \emptyset \wedge^* [\text{IsEmptyClass}(\mathit{class})]^{I''} \\
   =\ &[\text{IsClass}(\mathit{class})]^{I'\{1.4:\text{IsClass}[\mathit{class}/\mathit{true}]\}} \wedge^* \\
   &[\text{ClassReferences}(\mathit{class})]^{I'\{1.5:\text{ClassReferences}[\mathit{class}/\emptyset]\}} = \emptyset \wedge^* \\
   &\nexists s.[\text{Method}(\mathit{class}, s)]^{I''} \downarrow \wedge^* \\
   &\nexists v.v \in [\text{InstanceVariablesDefinedBy}(\mathit{class})]^{I'\{1.7:\text{InstanceVariablesDefinedBy}[\mathit{class}/\emptyset]\}} \\
   =\ &\top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top
   \end{aligned}$$

□

**Theorem 8** (`InvRemoveEmptyClass` II)
`InvRemoveEmptyClass` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `AddEmptyClass` refactoring precondition it holds that $I_P \models a \Longrightarrow I_A \models a$:

$$I_P \models \text{IsClass}(super) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{IsClass}(super)$$

$$I_P \models \neg\text{IsClass}(class) \Leftrightarrow I_P \not\models \text{IsClass}(class) \overset{Corollary\ 2}{\Longrightarrow}$$
$$I_A \not\models \text{IsClass}(class) \Leftrightarrow I_A \models \neg\text{IsClass}(class)$$

$$\forall c \in sub.\left(I_P \models \text{IsClass}(c) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{IsClass}(c)\right)$$

$$\forall c \in sub.\left(I_P \models (\text{Superclass}(c) = super) \overset{Corollary\ 2}{\Longrightarrow} I_A \models (\text{Superclass}(c) = super)\right)$$

□

## D.2 `AddMethod` and `RemoveMethod`

The `AddMethod` refactoring adds a new, unreferenced method to a class. If a method with the same name already exists in a superclass, the method to be added must have the same semantics (adopted from Roberts [Rob99, p. 106]).

`AddMethod`(*class*, *selector*, *body*)

$$\begin{aligned}
&pre: \text{IsClass}(class) \wedge && (2.1)\\
&\quad \neg\text{DefinesSelector}(class, selector) \wedge && (2.2)\\
&\quad (\neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \vee && (2.3)\\
&\quad \text{LookedUpMethod}(\text{Superclass}(class), selector) \overset{\alpha}{\equiv} body\big) && (2.4)
\end{aligned}$$

---

$$\begin{aligned}
&post: \text{Method}' = \text{Method}[(class, selector)/body] && (2.5)\\
&\quad \text{Senders}' = \text{Senders}[(class, selector)/\emptyset] && (2.6)\\
&\quad \text{ReferencesToInstanceVariable}' = \forall c \in \text{Superclass}^*(class) \cup \{class\}\,.\\
&\qquad \forall v \in \{v|\text{DefinesInstanceVariable}(c, v)\} \cap \text{UnboundVariables}(body).\\
&\qquad \text{ReferencesToInstanceVariable}[(c, v)/\\
&\qquad \text{ReferencesToInstanceVariables}(c, v) \cup \{(class, selector)\}] && (2.7)\\
&\quad \text{ClassReferences}' = \forall c \in \{c|\text{IsClass}(c)\} \cap \text{UnboundVariables}(body).\\
&\qquad \text{ClassReferences}[c/\text{ClassReferences}(c) \cup \{(class, selector)\}] && (2.8)
\end{aligned}$$

The `RemoveMethod` refactoring removes a method from a class if it is either unreferenced or overrides an inherited method with the same semantics (adopted from Roberts [Rob99, p. 107]).

`RemoveMethod`(*class*, *selector*, *body*)

$$\begin{aligned}
&pre: \text{IsClass}(class) \wedge && (2.9)\\
&\quad \text{DefinesSelector}(class, selector) \wedge && (2.10)\\
&\quad (\text{Senders}(class, selector) = \emptyset \vee && (2.11)\\
&\quad \text{Method}(class, selector) \overset{\alpha}{\equiv} \text{LookedUpMethod}(\text{Superclass}(class), selector)\big) && (2.12)
\end{aligned}$$

$$post: \text{Method}' = \text{Method}[(class, selector)/\uparrow] \tag{2.13}$$

$$\text{Senders}' = \text{Senders}[(class, selector)/\uparrow] \tag{2.14}$$

$$\begin{aligned}&\text{ClassReferences}' = \forall c \in \{c|\text{IsClass}(c)\} \cap \text{UnboundVariables}(body).\\ &\quad \text{ClassReferences}[c/\text{ClassReferences}(c) \setminus \{(class, selector)\}]\end{aligned} \tag{2.15}$$

$$\begin{aligned}&\text{ReferencesToInstanceVariable}' = \forall c \in \text{Superclass}^*(class) \cup \{class\}.\\ &\quad \forall v \in \{v|\text{DefinesInstanceVariable}(c, v)\} \cap \text{UnboundVariables}(body).\\ &\quad \text{ReferencesToInstanceVariable}[(c, v)/\\ &\quad \text{ReferencesToInstanceVariables}(c, v) \setminus \{(class, selector)\}]\end{aligned} \tag{2.16}$$

**Theorem 9** (`InvAddMethod` I)
The weak refactoring inverse for the `AddMethod` refactoring, `InvAddMethod`, is the `RemoveMethod` refactoring:

$$\texttt{InvAddMethod}(class, selector, body) := \texttt{RemoveMethod}(class, selector, body).$$

*Proof.*

1. `InvAddMethod` is behavior-preserving by definition, because it consists of the single refactoring `RemoveMethod`.

2. $post_{\texttt{AddMethod}}(P) \models pre_{\texttt{InvAddMethod}} \Leftrightarrow post_{\texttt{AddMethod}}(P) \models pre_{\texttt{RemoveMethod}}$:
   Let $I'$ be the interpretation gained by applying the `AddMethod` refactoring to the source code $S$ of the program $P$ reflected by $P' = post_{\texttt{AddMethod}}(P)$:

   $$I' = \left(trans_{\texttt{AddMethod}}(S), \cdot^{I}\right).$$

   The refactoring invariant of the `AddMethod` refactoring is

   $$\begin{aligned}inv(\texttt{AddMethod}) \quad = \quad &\{\text{IsClass}(class) \wedge\\ &(\neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \vee\\ &\text{LookedUpMethod}(\text{Superclass}(class), selector) \overset{\alpha}{\equiv} body)\Big)\Big\}.\end{aligned}$$

   Then $pre_{\texttt{RemoveMethod}}$ evaluates to true under $I'$ and the invariant:

   $$\begin{aligned}&[\text{IsClass}(class)]^{I'} \wedge^* [\text{DefinesSelector}(class, selector)]^{I'} \wedge^*\\ &\Big([\text{Senders}(class, selector)]^{I'} = \emptyset \vee^*\\ &[\text{Method}(class, s)]^{I'} \overset{\alpha}{\equiv} [\text{LookedUpMethod}(\text{Superclass}(class), s)]^{I'}\Big)\\ =\ &[\text{IsClass}(class)]^{I\{\text{Inv}\}} \wedge^* [\text{Method}(class, selector)]^{I\{2.5:\text{Method}[(class,selector)/body]\}\downarrow} \wedge^*\\ &\Big([\text{Senders}(class, selector)]^{I\{2.6:\text{Senders}[(class,selector)/\emptyset]\}} = \emptyset \vee^* *\Big)\\ &\top \wedge^* \top \wedge^* (\top \vee^* *) = \top\end{aligned}$$

3. $post_{\texttt{InvAddMethod}}(post_{\texttt{AddMethod}}(P)) \models pre_{\texttt{AddMethod}} \Leftrightarrow$
   $post_{\texttt{RemoveMethod}}(post_{\texttt{AddMethod}}(P)) \models pre_{\texttt{AddMethod}}$:
   Let $I''$ be the interpretation gained by applying the `InvAddMethod` inverse refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = post_{\texttt{InvAddMethod}}(P')$:

   $$I'' = \left(trans_{\texttt{RemoveMethod}}(S'), \cdot^{I}\right).$$

The invariant of the refactoring chain $\langle \texttt{AddMethod}, \texttt{RemoveMethod} \rangle$ is

$$\begin{aligned} inv(\langle \texttt{AddMethod}, \texttt{RemoveMethod} \rangle) = \{ & \text{IsClass}(class) \wedge \\ & (\neg \text{UnderstandsSelector}(\text{Superclass}(class), selector) \vee \\ & \text{LookedUpMethod}(\text{Superclass}(class), selector) \overset{\alpha}{\equiv} body) \Big) \Big\}. \end{aligned}$$

Then $pre_{\texttt{AddMethod}}$ evaluates to true under $I''$ and the invariant:

$$\begin{aligned} & [\text{IsClass}(class)]^{I''} \wedge^* \neg^* [\text{DefinesSelector}(class, selector)]^{I''} \wedge^* \\ & \Big( \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I''} \vee^* \\ & [\text{LookedUpMethod}(\text{Superclass}(class), selector)]^{I''} \overset{\alpha}{\equiv} body \Big) \\ = & [\text{IsClass}(class)]^{I'\{\text{Inv}\}} \wedge^* [\text{Method}(class, selector)]^{I'\{\text{2.13:Method}[(class,selector)/\uparrow]\}} \uparrow \wedge^* \\ & \Big( \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'\{\text{Inv}\}} \vee^* \\ & [\text{LookedUpMethod}(\text{Superclass}(class), selector)]^{I'\{\text{Inv}\}} \overset{\alpha}{\equiv} body \Big) \\ = & \top \wedge^* \top \wedge^* \top = \top \end{aligned}$$

□

**Theorem 10** (`InvAddMethod` II)
`InvAddMethod` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `RemoveMethod` refactoring precondition it holds that $I_P \models a \Longrightarrow I_A \models a$:

$$I_P \models \text{IsClass}(class) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{IsClass}(class)$$

$$I_P \models \text{DefinesSelector}(class, selector) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{DefinesSelector}(class, selector)$$

$$I_P \models \text{Senders}(class, selector) = \emptyset \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{Senders}(class, selector) = \emptyset$$

$$I_P \models \text{Method}(class, s) \overset{\alpha}{\equiv} \text{LookedUpMethod}(\text{Superclass}(class), s) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{Method}(class, s) \overset{\alpha}{\equiv} \text{LookedUpMethod}(\text{Superclass}(class), s)$$

□

Unless the refactoring engine performing the `RemoveMethod` refactoring provides a means for storing the code of the deleted method, there cannot be any refactoring inverse for it. Although state-of-the-technology systems, such as Eclipse [Ecl], NetBeans [Net], and JBuilder [JBu], only store the history of executed refactorings and their parameters, in the following we show that the `AddMethod` refactoring can serve as a weak inverse for the `RemoveMethod` refactoring. Thus, existing refactoring engines would need to be extended (or framework developers need to manually intervene), so that the following proof can be put into practice.

**Theorem 11** (`InvRemoveMethod` I)
The weak refactoring inverse for the `RemoveMethod` refactoring, `InvRemoveMethod`, is the `AddMethod` refactoring:

$$\texttt{InvRemoveMethod}(class, selector, body) := \texttt{AddMethod}(class, selector, body).$$

*Proof.*

1. `InvRemoveMethod` is behavior-preserving by definition, because it consists of the single refactoring `AddMethod`.

2. $post_{\texttt{RemoveMethod}}(P) \models pre_{\texttt{InvRemoveMethod}} \Leftrightarrow post_{\texttt{RemoveMethod}}(P) \models pre_{\texttt{AddMethod}}$:
Let $I'$ be the interpretation gained by applying the `RemoveMethod` refactoring to the source code $S$ of the program $P$ reflected by $P' = post_{\texttt{RemoveMethod}}(P)$:

$$I' = \left(trans_{\texttt{RemoveMethod}}(S), \cdot^{I}\right).$$

The refactoring invariant of the `RemoveMethod` refactoring is

$$inv(\texttt{RemoveMethod}) = \{\text{IsClass}(class)\}.$$

Then $pre_{\texttt{AddMethod}}$ evaluates to true under $I'$ and the invariant:

$$\begin{aligned}
&[\text{IsClass}(class)]^{I'} \wedge^* \neg^* [\text{DefinesSelector}(class, selector)]^{I'} \wedge^* \\
&\Big(\neg^* [\text{UnderstandsSelector}(class, selector)]^{I'} \vee^* \\
&[\text{LookedUpMethod}(class, selector)]^{I'} \overset{\alpha}{\equiv} body\Big) \\
&= [\text{IsClass}(class)]^{I\{\text{Inv}\}} \wedge^* [\text{Method}(class, selector)]^{I\{2.13:\text{Method}[(class,selector)/\uparrow]\}} \uparrow \wedge^* \\
&= \top \wedge^* \top \wedge^* \top = \top
\end{aligned}$$

The last truth value can be derived from the fact that the removed method must have been added before using the `AddMethod` refactoring and therefore fulfilled that part of the precondition.

3. $post_{\texttt{InvRemoveMethod}}(post_{\texttt{RemoveMethod}}(P)) \models pre_{\texttt{RemoveMethod}} \Leftrightarrow$
$post_{\texttt{AddMethod}}(post_{\texttt{RemoveMethod}}(P)) \models pre_{\texttt{RemoveMethod}}$:
Let $I''$ be the interpretation gained by applying the `InvRemoveMethod` inverse refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = post_{\texttt{InvRemoveMethod}}(P')$:

$$I'' = \left(trans_{\texttt{AddMethod}}(S'), \cdot^{I}\right).$$

The invariant of the refactoring chain $\langle \texttt{RemoveMethod}, \texttt{AddMethod} \rangle$ is

$$inv(\langle \texttt{RemoveMethod}, \texttt{AddMethod} \rangle) = \{\text{IsClass}(class)\}.$$

Then $pre_{\texttt{RemoveMethod}}$ evaluates to true under $I''$ and the invariant:

$$\begin{aligned}
&[\text{IsClass}(class)]^{I''} \wedge^* [\text{DefinesSelector}(class, selector)]^{I''} \wedge^* \\
&\Big([\text{Senders}(class, selector)]^{I''} = \emptyset \vee^* \\
&[\text{Method}(class, s)]^{I''} \overset{\alpha}{\equiv} [\text{LookedUpMethod}(\text{Superclass}(class), s)]^{I''}\Big) \\
&= [\text{IsClass}(class)]^{I'\{\text{Inv}\}} \wedge^* [\text{Method}(class, selector)]^{I'\{2.5:\text{Method}[(class,selector)/body]\}} \downarrow \wedge^* \\
&\Big([\text{Senders}(class, selector)]^{I'\{2.6:\text{Senders}[(class,selector)/\emptyset]\}} = \emptyset \vee^* *\Big) \\
&\top \wedge^* \top \wedge^* (\top \vee^* *) = \top
\end{aligned}$$

□

**Theorem 12** (`InvRemoveMethod` II)
`InvRemoveMethod` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `AddMethod` refactoring precondition it holds that $I_P \models a \Longrightarrow I_A \models a$:

$$I_P \models \text{IsClass}(class) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{IsClass}(class)$$

$$I_P \models \neg\text{DefinesSelector}(class, selector) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \neg\text{DefinesSelector}(class, selector)$$

$$I_P \models \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \Leftrightarrow$$
$$I_P \not\models \text{UnderstandsSelector}(\text{Superclass}(class), selector)$$
$$\overset{Corollary\ 2}{\Longrightarrow} I_A \not\models \text{UnderstandsSelector}(\text{Superclass}(class), selector) \Leftrightarrow$$
$$I_A \models \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector)$$

$$I_P \models \text{LookedUpMethod}(\text{Superclass}(class), selector) \overset{\alpha}{\equiv} body \overset{Corollary\ 2}{\Longrightarrow}$$
$$I_A \models \text{LookedUpMethod}(\text{Superclass}(class), selector) \overset{\alpha}{\equiv} body$$

□

## D.3 `RenameClass`

The `RenameClass` refactoring renames an existing class in the program and updates all references. It updates the references in the source code with the help of the *RenameReferences* analysis function, the inner workings of which are not important here. The following definition is adopted from Roberts [Rob99, p. 105].

`RenameClass`(*class*, *newClass*)

$$pre : \text{IsClass}(class) \wedge \tag{3.1}$$
$$\neg\text{IsClass}(newClass) \tag{3.2}$$

---

$$post : \text{IsClass}' = \text{IsClass}[class/false][newClass/true] \tag{3.3}$$
$$\text{ClassReferences}' = \forall c \neq class. \forall s \in \{s | (class, s) \in \text{ClassReferences}(c)\}.$$
$$\text{ClassReferences}[class/\uparrow][newClass/\text{ClassReferences}(class)]$$
$$[c/\text{ClassReferences}(c) \setminus \{(class, s)\} \cup \{(newClass, s)\}] \tag{3.4}$$
$$\text{ReferencesToInstanceVariable}' = \forall (c, v, s) \in$$
$$\{(c, v, s) | (class, s) \in \text{ReferencesToInstanceVariable}(c, v) \wedge c \neq class\}.$$
$$\forall lv. \forall lr \in \{lr | (class, lr) \in \text{ReferencesToInstanceVariable}(class, lv)\}.$$
$$\text{ReferencesToInstanceVariable}[(class, lv)/\uparrow][(newClass, lv)/$$
$$\text{ReferencesToInstanceVariable}(class, lv) \setminus \{(class, lr)\} \cup \{(newClass, lr)\}]$$
$$[(c, v)/\text{ReferencesToInstanceVariable}(c, v) \setminus \{(class, s)\} \cup \{(newClass, s)\}] \tag{3.5}$$
$$\text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[class/\uparrow]$$
$$[newClass/\text{Superclass}(class)][c/newClass] \tag{3.6}$$
$$\text{Method}' = \forall c. \forall s. \text{Method}[(newClass, s)/\text{Method}(class, s)][(class, s)/\uparrow]$$
$$[(c, s)/\text{RenameReferences}(\text{Method}(c, s), class, newClass)] \tag{3.7}$$
$$\text{Senders}' = \forall s. \text{Senders}[(newClass, s)/\text{Senders}(class, s)][(class, s)/\uparrow] \tag{3.8}$$
$$\text{InstanceVariablesDefinedBy}' = \text{InstanceVariablesDefinedBy}[newClass/$$
$$\text{InstanceVariablesDefinedBy}(class)][class/\uparrow] \tag{3.9}$$

**Theorem 13** (`InvRenameClass` I)
The weak refactoring inverse for the `RenameClass` refactoring, `InvRenameClass`, is the `RenameClass` refactoring:

$$\texttt{InvRenameClass}(\mathit{class}, \mathit{newClass}) := \texttt{RenameClass}(\mathit{newClass}, \mathit{class}).$$

*Proof.*

1. `InvRenameClass` is behavior-preserving by definition, because it consists of the single refactoring `RenameClass`.

2. $\mathit{post}_{\texttt{RenameClass}}(P) \models \mathit{pre}_{\texttt{InvRenameClass}} \Leftrightarrow \mathit{post}_{\texttt{RenameClass}}(P) \models \mathit{pre}_{\texttt{RenameClass}}$:
   Let $I'$ be the interpretation gained by applying the `RenameClass` refactoring to the source code $S$ of the program $P$ reflected by $P' = \mathit{post}_{\texttt{RenameClass}}(P)$

   $$I' = \left(\mathit{trans}_{\texttt{RenameClass}}(S), \cdot^{I}\right),$$

   and let $Z$ be the variable assignment induced by theorem 13

   $$Z = \{\mathit{class} \mapsto \mathit{newClass}, \mathit{newClass} \mapsto \mathit{class}\}.$$

   Then $\mathit{pre}_{\texttt{RenameClass}}$ evaluates to true under $I'$ and $Z$:

   $$\begin{aligned}
   &[\text{IsClass}(\mathit{class})]^{I',Z} \wedge^* \neg^* [\text{IsClass}(\mathit{newClass})]^{I',Z} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I'} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I'} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I\{3.3:\text{IsClass}[\mathit{newClass}/\mathit{true}]\}} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I\{3.3:\text{IsClass}[\mathit{class}/\mathit{false}]\}} \\
   &= \top \wedge^* \neg^* \bot = \top
   \end{aligned}$$

3. $\mathit{post}_{\texttt{InvRenameClass}}(\mathit{post}_{\texttt{RenameClass}}(P)) \models \mathit{pre}_{\texttt{RenameClass}}$:
   Let $I''$ be the interpretation gained by applying the `RenameClass` refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = \mathit{post}_{\texttt{RenameClass}}(P')$

   $$I'' = \left(\mathit{trans}_{\texttt{RenameClass}}(S'), \cdot^{I}\right),$$

   and let $Z'$ be the variable assignment induced by theorem 13

   $$Z' = \{\mathit{class} \mapsto \mathit{newClass}, \mathit{newClass} \mapsto \mathit{class}\}.$$

   Then $\mathit{pre}_{\texttt{RenameClass}}$ evaluates to true under $I''$ and $Z'$:

   $$\begin{aligned}
   &[\text{IsClass}(\mathit{class})]^{I'',Z'} \wedge^* \neg^* [\text{IsClass}(\mathit{newClass})]^{I'',Z'} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I''} \wedge^* \neg^* [\text{IsClass}(\mathit{class})]^{I''} \\
   &= [\text{IsClass}(\mathit{newClass})]^{I'\{3.3:\text{IsClass}[\mathit{newClass}/\mathit{true}]\}} \wedge^* \\
   &\quad \neg^* [\text{IsClass}(\mathit{class})]^{I'\{3.3:\text{IsClass}[\mathit{class}/\mathit{false}]\}} \\
   &= \top \wedge^* \neg^* \bot = \top
   \end{aligned}$$

□

**Theorem 14** (`InvRenameClass` II)
`InvRenameClass` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `AddEmptyClass` refactoring precondition it holds that $I_P \models a \Longrightarrow I_A \models a$:

$$I_P \models \text{IsClass}(\mathit{class}) \overset{\mathit{Corollary}\ 2}{\Longrightarrow} I_A \models \text{IsClass}(\mathit{class})$$

$$I_P \models \neg\text{IsClass}(\mathit{newClass}) \Leftrightarrow I_P \not\models \text{IsClass}(\mathit{newClass}) \overset{\mathit{Corollary}\ 2}{\Longrightarrow} I_A \not\models \text{IsClass}(\mathit{newClass}) \Leftrightarrow I_A \models \neg\text{IsClass}(\mathit{newClass})$$

□

## D.4 `RenameMethod`

The `RenameMethod` refactoring renames an existing method in a set of classes and updates all references (adopted from Roberts [Rob99, p. 108]).

`RenameMethod`(*classes*, *selector*, *newSelector*)

$$\begin{aligned}
&\mathit{pre} : \forall c \in \mathit{classes}.\text{IsClass}(c) \wedge && (4.1)\\
&\quad \text{DefinesSelector}(c, \mathit{selector}) \wedge && (4.2)\\
&\quad (\forall d \notin \mathit{classes}.\text{DefinesSelector}(d, \mathit{selector}) \Rightarrow \\
&\qquad (\text{Senders}(d, \mathit{selector}) \cap \text{Senders}(c, \mathit{selector})) = \emptyset) && (4.3)
\end{aligned}$$

---

$$\begin{aligned}
&\mathit{post} : \text{Method}' = \forall d.\forall s.\forall c \in \mathit{classes}.\text{Method}[(c, \mathit{newSelector})/\text{Method}(c, \mathit{selector})] \\
&\qquad [(c, \mathit{selector})/\uparrow][(d,s)/\text{RenameReferences}(\text{Method}(d,s), \mathit{selector}, \mathit{newSelector})] && (4.4)\\
&\quad \text{Senders}' = \forall c \in \mathit{classes}.\text{Senders}[(c, \mathit{newSelector})/\text{Senders}(c, \mathit{selector})] \\
&\qquad [(c, \mathit{selector})/\uparrow] && (4.5)\\
&\quad \text{ReferencesToInstanceVariable}' = \forall c \in \mathit{classes}.\forall (d,v) \in \\
&\qquad \{(d,v) | (c, \mathit{selector}) \in \text{ReferencesToInstanceVariable}(d,v)\}. \\
&\qquad \text{ReferencesToInstanceVariable}[(d,v)/\text{ReferencesToInstanceVariable}(d,v) \setminus \\
&\qquad \{(c, \mathit{selector})\} \cup \{(c, \mathit{newSelector})\} && (4.6)\\
&\quad \text{ClassReferences}' = \forall c \in \{c | \text{IsClass}(c)\}. \\
&\qquad \forall \mathit{class} \in \{\mathit{class} | \mathit{class} \in \mathit{classes} \wedge (\mathit{class}, \mathit{selector}) \in \text{ClassReferences}(c)\}. \\
&\qquad \text{ClassReferences}[c/\text{ClassReferences}(c) \setminus \\
&\qquad \{(\mathit{class}, \mathit{selector})\} \cup \{(\mathit{class}, \mathit{newSelector})\}] && (4.7)
\end{aligned}$$

**Theorem 15** (`InvRenameMethod` I)
The weak refactoring inverse for the `RenameMethod` refactoring, `InvRenameMethod`, is the `RenameMethod` refactoring:

$$\begin{aligned}
&\texttt{InvRenameMethod}(\mathit{classes}, \mathit{selector}, \mathit{newSelector}) := \\
&\qquad \texttt{RenameMethod}(\mathit{classes}, \mathit{newSelector}, \mathit{selector}).
\end{aligned}$$

*Proof.*

1. `InvRenameMethod` is behavior-preserving by definition, because it consists of the single refactoring `RenameMethod`.

2. $\mathit{post}_{\texttt{RenameMethod}}(P) \models \mathit{pre}_{\texttt{InvRenameMethod}} \Leftrightarrow \mathit{post}_{\texttt{RenameMethod}}(P) \models \mathit{pre}_{\texttt{RenameMethod}}$:
   Let $I'$ be the interpretation gained by applying the `RenameMethod` refactoring to the source code $S$ of the program $P$ reflected by $P' = \mathit{post}_{\texttt{RenameMethod}}(P)$

   $$I' = \left(\mathit{trans}_{\texttt{RenameMethod}}(S), \cdot^I\right),$$

   and let $Z$ be the variable assignment induced by theorem 15

   $$Z = \{\mathit{classes} \mapsto \mathit{classes}, \mathit{selector} \mapsto \mathit{newSelector}, \mathit{newSelector} \mapsto \mathit{selector}\}.$$

   The refactoring invariant of the `RenameMethod` refactoring is

   $$\begin{aligned}
   \mathit{inv}(\texttt{RenameMethod}) \quad = \quad & \{\forall c \in \mathit{classes}.\text{IsClass}(c) \wedge \\
   & (\forall d \notin \mathit{classes}.\text{DefinesSelector}(d, \mathit{selector}) \Rightarrow \\
   & \quad (\text{Senders}(d, \mathit{selector}) \cap \text{Senders}(c, \mathit{selector})) = \emptyset)\}.
   \end{aligned}$$

Then $pre_{\texttt{RenameMethod}}$ evaluates to true under $I', Z$, and the invariant:

$$
\begin{aligned}
&\forall c \in classes^{Z}.[\text{IsClass}(c)]^{I',Z} \wedge^{*} [\text{DefinesSelector}(c, selector)]^{I',Z} \wedge^{*} \\
&\Big(\forall d \notin classes^{Z}.[\text{DefinesSelector}(d, selector)]^{I',Z} \Rightarrow^{*} \\
&\Big([\text{Senders}(d, selector)]^{I',Z} \cap [\text{Senders}(c, selector)]^{I',Z}\Big) = \emptyset\Big) \\
= \forall c \in classes.&[\text{IsClass}(c)]^{I'} \wedge^{*} [\text{DefinesSelector}(c, newSelector)]^{I'} \wedge^{*} \\
&\Big(\forall d \notin classes.[\text{DefinesSelector}(d, newSelector)]^{I'} \Rightarrow^{*} \\
&\Big([\text{Senders}(d, newSelector)]^{I'} \cap [\text{Senders}(c, newSelector)]^{I'}\Big) = \emptyset\Big) \\
= \forall c \in classes.&[\text{IsClass}(c)]^{I\{\text{Inv}\}} \wedge^{*} \\
&[\text{Method}(c, newSelector)]^{I\{4.4:\text{Method}[(c,newSelector)/\text{Method}(c,selector)]\}\downarrow} \wedge^{*} \\
&\Big(\forall d \notin classes.[\text{DefinesSelector}(d, newSelector)]^{I\{\text{Inv}\}} \Rightarrow^{*} \\
&\Big([\text{Senders}(d, newSelector)]^{I\{\text{Inv}\}} \cap [\text{Senders}(c, newSelector)]^{I\{\text{Inv}\}}\Big) = \emptyset\Big) \\
= \top&
\end{aligned}
$$

3. $post_{\texttt{InvRenameMethod}}(post_{\texttt{RenameMethod}}(P)) \models pre_{\texttt{RenameMethod}}$:
Let $I''$ be the interpretation gained by applying the `RenameMethod` refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = post_{\texttt{RenameMethod}}(P')$

$$I'' = \left(trans_{\texttt{RenameMethod}}(S'), \cdot^{I}\right),$$

and let $Z'$ be the variable assignment induced by theorem 15

$$Z' = \{classes \mapsto classes, selector \mapsto newSelector, newSelector \mapsto selector\}.$$

Then $pre_{\texttt{RenameMethod}}$ evaluates to true under $I'', Z'$, and the invariant:

$$
\begin{aligned}
&\forall c \in classes^{Z'}.[\text{IsClass}(c)]^{I'',Z'} \wedge^{*} [\text{DefinesSelector}(c, selector)]^{I'',Z'} \wedge^{*} \\
&\Big(\forall d \notin classes^{Z'}.[\text{DefinesSelector}(d, selector)]^{I'',Z'} \Rightarrow^{*} \\
&\Big([\text{Senders}(d, selector)]^{I'',Z'} \cap [\text{Senders}(c, selector)]^{I'',Z'}\Big) = \emptyset\Big) \\
= \forall c \in classes.&[\text{IsClass}(c)]^{I''} \wedge^{*} [\text{DefinesSelector}(c, newSelector)]^{I''} \wedge^{*} \\
&\Big(\forall d \notin classes.[\text{DefinesSelector}(d, newSelector)]^{I''} \Rightarrow^{*} \\
&\Big([\text{Senders}(d, newSelector)]^{I''} \cap [\text{Senders}(c, newSelector)]^{I''}\Big) = \emptyset\Big) \\
= \forall c \in classes.&[\text{IsClass}(c)]^{I'\{\text{Inv}\}} \wedge^{*} \\
&[\text{Method}(c, newSelector)]^{I'\{4.4:\text{Method}[(c,newSelector)/\text{Method}(c,selector)]\}\downarrow} \wedge^{*} \\
&\Big(\forall d \notin classes.[\text{DefinesSelector}(d, newSelector)]^{I'\{\text{Inv}\}} \Rightarrow^{*} \\
&\Big([\text{Senders}(d, newSelector)]^{I'\{\text{Inv}\}} \cap [\text{Senders}(c, newSelector)]^{I'\{\text{Inv}\}}\Big) = \emptyset\Big) \\
= \top&
\end{aligned}
$$

□

**Theorem 16** (`InvRenameMethod` II)
`InvRenameMethod` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `RenameMethod` refactoring precondition it holds that $I_P \models a \Longrightarrow I_A \models a$:

$$\forall c \in classes.\left(I_P \models \text{IsClass}(c) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{IsClass}(c)\right)$$

$$\forall c \in classes.\left(I_P \models \text{DefinesSelector}(c, selector) \overset{Corollary\ 2}{\Longrightarrow} I_A \models \text{DefinesSelector}(c, selector)\right)$$

$$\begin{aligned}&\forall c \in classes.(I_P \models (\forall d \notin classes.\text{DefinesSelector}(d, selector) \Rightarrow \\ &\qquad (\text{Senders}(d, selector) \cap \text{Senders}(c, selector)) = \emptyset) \\ &\overset{Corollary\ 2}{\Longrightarrow} I_A \models (\forall d \notin classes.\text{DefinesSelector}(d, selector) \Rightarrow \\ &\qquad (\text{Senders}(d, selector) \cap \text{Senders}(c, selector)) = \emptyset))\end{aligned}$$

□

## D.5 `MoveMethod`

The `MoveMethod` refactoring moves an existing method from one class to another and ajust the method to its new location. The following is our general definition of the refactoring, which is based on a more specialized definition as described by Roberts [Rob99, p. 109].

`MoveMethod`(*class*, *selector*, *newClass*)

$$\begin{aligned}&pre : \text{IsClass}(class) \wedge && (5.1)\\ &\quad \text{DefinesSelector}(class, selector) \wedge && (5.2)\\ &\quad \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \wedge && (5.3)\\ &\quad \text{IsClass}(newClass) \wedge && (5.4)\\ &\quad \neg\text{UnderstandsSelector}(newClass, selector) \wedge && (5.5)\\ &\quad \text{UnboundVariables}(class, selector) = \emptyset && (5.6)\end{aligned}$$

---

$$\begin{aligned}&post : \text{Method}' = \text{Method}[(newClass, selector)/\text{Method}(class, selector)] \\ &\qquad [(class, selector)/\uparrow] && (5.7)\\ &\quad \text{Senders}' = \text{Senders}[(newClass, selector)/\text{Senders}(class, selector)] \\ &\qquad [(class, selector)/\uparrow] && (5.8)\\ &\quad \text{ClassReferences}' = \forall c \in \{c | \text{IsClass}(c) \wedge (class, selector) \in \text{ClassReferences}(c)\}. \\ &\qquad \text{ClassReferences}[c/\text{ClassReferences}(c) \setminus \\ &\qquad \{(class, selector)\} \cup \{(newClass, selector)\}] && (5.9)\end{aligned}$$

**Theorem 17** (`InvMoveMethod` I)
The weak refactoring inverse for the `MoveMethod` refactoring, `InvMoveMethod`, is the `MoveMethod` refactoring:

$$\begin{aligned}&\texttt{InvMoveMethod}(class, selector, newClass) := \\ &\qquad \texttt{MoveMethod}(newClass, selector, class).\end{aligned}$$

*Proof.*

1. `InvMoveMethod` is behavior-preserving by definition, because it consists of the single refactoring `MoveMethod`.

2. $post_{\texttt{MoveMethod}}(P) \models pre_{\texttt{InvMoveMethod}} \Leftrightarrow post_{\texttt{MoveMethod}}(P) \models pre_{\texttt{MoveMethod}}$:
Let $I'$ be the interpretation gained by applying the `MoveMethod` refactoring to the source code $S$ of the program $P$ reflected by $P' = post_{\texttt{MoveMethod}}(P)$

$$I' = \left(trans_{\texttt{MoveMethod}}(S), \cdot^{I}\right),$$

and let $Z$ be the variable assignment induced by theorem 17

$$Z = \{class \mapsto newClass, selector \mapsto selector, newClass \mapsto class\}.$$

The refactoring invariant of the `MoveMethod` refactoring is

$$\begin{aligned} inv(\texttt{MoveMethod}) \quad = \quad & \{\text{IsClass}(class) \wedge \\ & \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \wedge \\ & \text{IsClass}(newClass)\}. \end{aligned}$$

Then $pre_{\texttt{MoveMethod}}$ evaluates to true under $I', Z$, and the invariant:

$$\begin{aligned} & [\text{IsClass}(class)]^{I',Z} \wedge^* [\text{DefinesSelector}(class, selector)]^{I',Z} \wedge^* \\ & \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I',Z} \wedge^* \\ & [\text{IsClass}(newClass)]^{I',Z} \wedge^* \neg^* [\text{UnderstandsSelector}(newClass, selector)]^{I',Z} \wedge^* \\ & [\text{UnboundVariables}(class, selector)]^{I',Z} = \emptyset \\ = \ & [\text{IsClass}(newClass)]^{I'} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I'} \wedge^* \\ & \neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I'} \wedge^* \\ & [\text{IsClass}(class)]^{I'} \wedge^* \neg^* [\text{UnderstandsSelector}(class, selector)]^{I'} \wedge^* \\ & [\text{UnboundVariables}(newClass, selector)]^{I'} = \emptyset \\ = \ & [\text{IsClass}(newClass)]^{I'} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I'} \wedge^* \\ & \neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I'} \wedge^* \\ & [\text{IsClass}(class)]^{I'} \wedge^* \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'} \wedge^* \\ & \neg^* [\text{DefinesSelector}(class, selector)]^{I'} \wedge^* [\text{UnboundVariables}(newClass, selector)]^{I'} = \emptyset \\ = \ & [\text{IsClass}(newClass)]^{I\{\text{Inv}\}} \wedge^* \\ & [\text{Method}(newClass, selector)]^{I\{5.7:\text{Method}[(newClass, selector)/\text{Method}(class, selector)]\}} \downarrow \wedge^* \\ & \top \wedge^* [\text{IsClass}(class)]^{I\{\text{Inv}\}} \wedge^* \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I\{\text{Inv}\}} \wedge^* \\ & [\text{Method}(class, selector)]^{I\{5.7:\text{Method}[(class, selector)/\uparrow]\}} \uparrow \wedge^* \top \\ = \ & \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top = \top \end{aligned}$$

The last truth value for the *UnboundVariables* function can be deduced from the fact that the abstract syntax tree of the moved method is not changed and so no references to non-local variables are introduced.

3. $post_{\texttt{InvMoveMethod}}(post_{\texttt{MoveMethod}}(P)) \models pre_{\texttt{MoveMethod}}$:
Let $I''$ be the interpretation gained by applying the `MoveMethod` refactoring to the source code $S'$ of the program $P'$ reflected by $P'' = post_{\texttt{MoveMethod}}(P')$

$$I'' = \left(trans_{\texttt{MoveMethod}}(S'), \cdot^{I}\right),$$

and let $Z'$ be the variable assignment induced by theorem 17

$$Z' = \{class \mapsto newClass, selector \mapsto selector, newClass \mapsto class\}.$$

Then $pre_{\texttt{MoveMethod}}$ evaluates to true under $I'', Z'$, and the invariant:

$$
\begin{aligned}
&[\text{IsClass}(class)]^{I'',Z'} \wedge^* [\text{DefinesSelector}(class, selector)]^{I'',Z'} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'',Z} \wedge^* \\
&[\text{IsClass}(newClass)]^{I'',Z'} \wedge^* \neg^* [\text{UnderstandsSelector}(newClass, selector)]^{I'',Z'} \wedge^* \\
&[\text{UnboundVariables}(class, selector)]^{I'',Z'} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I''} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I''} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I''} \wedge^* \\
&[\text{IsClass}(class)]^{I''} \wedge^* \neg^* [\text{UnderstandsSelector}(class, selector)]^{I''} \wedge^* \\
&[\text{UnboundVariables}(newClass, selector)]^{I''} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I''} \wedge^* [\text{DefinesSelector}(newClass, selector)]^{I''} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(newClass), selector)]^{I''} \wedge^* \\
&[\text{IsClass}(class)]^{I''} \wedge^* \neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I''} \wedge^* \\
&\neg^* [\text{DefinesSelector}(class, selector)]^{I''} \wedge^* [\text{UnboundVariables}(newClass, selector)]^{I''} = \emptyset \\
= &[\text{IsClass}(newClass)]^{I'\{\text{Inv}\}} \wedge^* \\
&[\text{Method}(newClass, selector)]^{I'\{5.7:\text{Method}[(newClass,selector)/\text{Method}(class,selector)]\}\downarrow} \wedge^* \\
&\top \wedge^* [\text{IsClass}(class)]^{I'\{\text{Inv}\}} \wedge^* \\
&\neg^* [\text{UnderstandsSelector}(\text{Superclass}(class), selector)]^{I'\{\text{Inv}\}} \wedge^* \\
&[\text{Method}(class, selector)]^{I'\{5.7:\text{Method}[(class,selector)/\uparrow]\}\uparrow} \wedge^* \top \\
= &\top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top \wedge^* \top = \top
\end{aligned}
$$

Again, the last truth value is implied by not changing the method's abstract syntax tree when moving it to the new class.

□

**Theorem 18** (`InvMoveMethod` II)
`InvMoveMethod` can be executed on adapters.

*Proof.* Let $I_P$ be the interpretation induced by the program $P$ and let $I_A$ be the interpretation representing the program after the execution of the `CreateAdapterLayer` refactoring, then for each assertion $a$ in the `MoveMethod` refactoring precondition it holds that $I_P \models a \implies I_A \models a$:

$$I_P \models \text{IsClass}(class) \overset{Corollary\ 2}{\implies} I_A \models \text{IsClass}(class)$$

$$I_P \models \text{DefinesSelector}(class, selector) \overset{Corollary\ 2}{\implies} I_A \models \text{DefinesSelector}(class, selector)$$

$$I_P \models \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector) \overset{Corollary\ 2}{\implies} I_A \models \neg\text{UnderstandsSelector}(\text{Superclass}(class), selector)$$

$$I_P \models \text{IsClass}(newClass) \overset{Corollary\ 2}{\implies} I_A \models \text{IsClass}(newClass)$$

$$I_P \models \neg\text{UnderstandsSelector}(newClass, selector) \overset{Corollary\ 2}{\implies} I_A \models \neg\text{UnderstandsSelector}(newClass, selector)$$

$$I_P \models \text{UnboundVariables}(class, selector) = \emptyset \overset{Corollary\ 2}{\implies} I_A \models \text{UnboundVariables}(class, selector) = \emptyset$$

Note that the last implication only holds true, if access to the adaptee field in adapters is encapsulated, i.e., performed via a method call. If it were accessed directly, the adaptee field would be in the set of unbound variables of the adapter method. □

## D.6 `ExtractMethod` and `InlineMethod`

The `ExtractMethod` refactoring [FBB+99, p. 89] extracts a part of a method into a new method. Applied to a framework method, the refactoring changes only the API and not the executed code, because the extracted method has to be called unconditionally from the original method. At the core of the transformation is the addition of a new method, which can be implemented using the `AddMethod` refactoring, and the modification of the abstract syntax tree of the existing method inserting a call to the new method. However, the language metamodel introduced in Section 4.1.2 on page 107 does not permit the specification of this abstract syntax tree transformation at the necessary level of detail (e.g., passing accessed local variables as arguments to the extracted method). For the sake of investigating the aptness of these refactorings for adaptation, we abstract from these intricacies and only look at the precondition of the `AddMethod` refactoring as presented in Section D.2.

The transformation inverting a method extraction is called `InlineMethod` [FBB+99, p. 95] and first replaces all call sites of the inlined method with the method body before the method is deleted. The latter part can be implemented using the `RemoveMethod` refactoring also presented in Section D.2. As a consequence of this reuse of basic refactorings, extracting and inlining methods can also be performed on adapters. The resulting semantics, however, are slightly different from that of real inverse transformations and shall be pointed out briefly.

For inverting a method extraction on adapters it suffices to execute the weak refactoring inverse for the addition of the extracted method. Since adapter methods contain forwarding calls to their adaptees, the application of the `RemoveMethod` refactoring will restore the state of the API before the extraction but the adapter method will still end up calling the extracted control flow in its latest form. However, it will not lead to any problems, because the execution of weak inverse transformations on adapters must reconstruct the old API represented by adapters (in this case, to avoid accidental method overriding) and not the control flow of the adaptees.

The inlining of a method can be inverted by simply re-adding the inlined method to the class it was removed from. The place(s) the method has been inlined into do not need to be changed back to call the re-added method, because they contain semantically equivalent code. Similarly to the inverse transformation for the `RemoveMethod` refactoring, it is necessary to provide the missing (in this case, inlined) code, so that it can be inserted again when inverting the inlining of an API method.

Finally, this discussion is not applicable to API methods that should serve as hook methods. In these cases, simply adding an inlined method or removing an extracted method may lead to fragile base class problems [MS98].

# E

# Refactorings in Adaptation Layer Construction

This appendix presents additional refactorings that do not require definitions and proofs of weak refactoring inverses and comebacks, because their effects are assumed to be invisible to old callers of the refactored API, in which no instance variable is declared public or protected. Therefore, under our assumptions the execution of these refactorings cannot break existing applications. However, because we use these refactorings during the construction of the adaptation layer (as described in Section 4.3.2), we show their formal definitions adapted to a statically-typed environment of Java.

## E.1 `AddInstanceVariable`

The `AddInstanceVariable` refactoring adds a new instance variable to an existing class (adopted from Roberts [Rob99, p. 110]).

`AddInstanceVariable`($class$, $varName$, $type$)

$pre$ : IsClass($class$)$\wedge$
IsClass($type$)$\wedge$
$\neg$HierarchyReferencesInstVariable($class$, $varName$)

---

$post$ : InstanceVariablesDefinedBy$'$ = InstanceVariablesDefinedBy[$class$/
InstanceVariablesDefinedBy($class$) $\cup$ {$varName$}]
ClassOf$'$ = ClassOf[($class$, $varName$)/$type$]
ReferencesToInstanceVariable$'$ = ReferencesToInstanceVariable[($class$, $varName$)/$\emptyset$]

## E.2 `PullUpInstanceVariable`

The `PullUpInstanceVariable` refactoring moves existing instance variables from a set of sibling classes to their superclass. In case the superclass does not already define an instance variable with the same name, the refactoring definition of Roberts [Rob99, p. 113]) will attempt to evaluate the *ReferencesToInstanceVariable* function for an undefined input. To prevent this, we introduce the following helper function to return an empty set in the case of undefined input and modify accordingly the original definition of the refactoring.

**EmptySetIfUndefined(*function, input*)** Wrapper returning an empty set if the given function is undefined for the given input.

$$\text{EmptySetIfUndefined}(function, input) := \begin{cases} \emptyset & \text{if } function(input)\uparrow \\ function(input) & \text{otherwise} \end{cases}$$

`PullUpInstanceVariable`($classes$, $varName$)

$pre$ : $\exists super.\forall c \in classes.\,(\text{IsClass}(c) \wedge$
$\quad \text{Superclass}(c) = super \wedge$
$\quad varName \in \text{InstanceVariablesDefinedBy}(c))$

---

$post$ : $\text{InstanceVariablesDefinedBy}' = \forall c \in classes.\text{InstanceVariablesDefinedBy}[c/$
$\quad \text{InstanceVariablesDefinedBy}(c) \setminus \{varName\}][\text{Superclass}(c)/$
$\quad \text{InstanceVariablesDefinedBy}(\text{Superclass}(c)) \cup \{varName\}]$
$\text{ClassOf}' = \forall c \in classes.\text{ClassOf}[(\text{Superclass}(c), varName)/\text{MostSpecificSuperclass}($
$\quad \text{ClassOf}(c, varName), \text{ClassOf}(\text{Superclass}(c), varName))][(c, varName)/\uparrow]$
$\text{ReferencesToInstanceVariable}' = \forall c \in classes.\text{ReferencesToInstanceVariable}$
$\quad [(\text{Superclass}(c), varName)/\text{ReferencesToInstanceVariable}(c, varName) \cup$
$\quad \text{EmptySetIfUndefined}(\text{ReferencesToInstanceVariable}, (\text{Superclass}(c), varName))]$
$\quad [(c, varName)/\uparrow]$

The actual source code transformation need to ensure that the variable pulled up is initialized correctly in the subclasses, so that its run-time type corresponds to the type of the original variable slot. Furthermore, all read accesses to the variable from within the subclasses need to be enhanced with casts, otherwise the source code might not compile anymore.

## E.3 `PushDownInstanceVariable`

The `PushDownInstanceVariable` refactoring moves an existing instance variable from a class to a set of subclasses (adopted from Roberts [Rob99, p. 114]).

`PushDownInstanceVariable`($classes$, $varName$)

$pre$ : $\exists super.\forall c \in classes.\,(\text{IsClass}(c) \wedge$
$\quad \text{Superclass}(c) = super \wedge$
$\quad varName \in \text{InstanceVariablesDefinedBy}(super))$

---

$post$ : $\text{InstanceVariablesDefinedBy}' = \forall c \in classes.\text{InstanceVariablesDefinedBy}[c/$
$\quad \text{InstanceVariablesDefinedBy}(c) \cup \{varName\}][\text{Superclass}(c)/$
$\quad \text{InstanceVariablesDefinedBy}(\text{Superclass}(c)) \setminus \{varName\}]$
$\text{ClassOf}' = \forall c \in classes.\text{ClassOf}[(c, varName)/\text{ClassOf}(\text{Superclass}(c), varName)]$
$\quad [(\text{Superclass}(c), varName)/\uparrow]$
$\text{ReferencesToInstanceVariable}' = \forall c \in classes.\text{ReferencesToInstanceVariable}$
$\quad [(c, varName)/\text{ReferencesToInstanceVariable}(\text{Superclass}(c), varName)]$
$\quad [(\text{Superclass}(c), varName)/\uparrow]$

# Bibliography


[AD04] Jonathan Aldrich and Kevin Donnelly. Selective open recursion: Modular reasoning about components and inheritance. In *SAVCBS '04: Proceedings of the FSE 2004 Workshop on Specification and Verification of Component-Based Systems*, November 2004.

[AFL99] Jim Alves-Foss and Fong Shing Lam. Dynamic denotational semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 201–240, London, UK, 1999. Springer-Verlag.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[Ang02] Dennis Angeline. Side-by-side execution of the .NET framework. `http://msdn2.microsoft.com/en-us/library/ms994410.aspx`, December 2002.

[ASM] ASM homepage. `http://asm.objectweb.org`.

[ASU86] Alfred Vaino Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, January 1986.

[BA01] Lodewijk Bergmans and Mehmet Akşit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, November 2004.

[Bat07] Don S. Batory. Program refactoring, program synthesis, and model-driven development. In *CC '07: Proceedings of the 16th International Conference on Compiler Construction*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2007.

[Bax92] Ira D. Baxter. Design maintenance systems. *Commununication of the ACM*, 35(4):73–89, 1992.

[BBG+04] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 193–215. Springer, 2004.

[BBP+79] Friedrich L. Bauer, Manfred Broy, Helmuth Partsch, Peter Pepper, and Hans Wössner. Systematics of transformation rules. In Friedrich L. Bauer and Manfred Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 1979.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[BCP06] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Component adaptation through flexible subservicing. *Science of Computer Programming*, 63(1):39–56, 2006.

[Ber91] Paul L. Bergstein. Object-preserving class transformations. In Andreas Paepcke, editor, *OOPSLA '91: Proceedings of the 6th Conference on Object-oriented Programming Systems, Languages and Applications*, pages 299–313, New York, NY, USA, November 1991. ACM. Printed as SIGPLAN Notices 26(11).

[BS07] Don S. Batory and Doug Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-07-66, University of Texas at Austin, Department of Computer Sciences, 2007.

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA '05: Proceedings of the 20th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

[CDN] Comarch CDN XL. `http://www.comarch.com/cdn/Products/CDN+XL`.

[CE00] Krzysztof Czarnecki and Ullrich Eisenecker. *Generative Programming Methods, Tools and Applications*. Addison-Wesley, Boston, MA, 2000.

[CGL] CGLIB homepage. `http://cglib.sourceforge.net`.

[CN96] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 4th International Conference on Software Maintenance*, pages 359–368, Washington, DC, USA, November 1996. IEEE Computer Society.

[COMa] Microsoft COM. `http://www.microsoft.com/Com/default.mspx`.

[Comb] Comarch homepage. `http://www.comarch.com`.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 211–220, New York, NY, USA, 2006. ACM.

[CPS06] Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing behavioural mismatch in software composition. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2006.

[DCED96] Pierre Deransart, Laurent Cervoni, and Abdel Ali Ed-Dbali. *Prolog: The Standard Reference Manual*. Springer, London, UK, 1996.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Computer Science*, pages 404–428. Springer, 2006.

[DDGM07] Brett Daniel, Danny Dig, Kelly Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *CCS '97: Proceedings of the 4th Conference on Computer and Communications Security*, pages 18–27, New York, NY, USA, 1997. ACM.

[Dig07] Danny Dig. *Automated Upgrading of Component-Based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, October 2007.

[Dij75] Edsger Wybe Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DJ05] Danny Dig and Ralph E. Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[DKTE04] Alan Donovan, Adam Kiežun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 15–34, New York, NY, USA, 2004. ACM.

[DME09] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 397–407, New York, NY, USA, 2009. ACM.

[DMJN07] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[DNMJ08] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 441–450, New York, NY, USA, 2008. ACM.

[DR08] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 481–490, New York, NY, USA, 2008. ACM.

[DRG+05] Serge Demeyer, Filip van Rysselberghe, Tudor Girba, Jacek Ratzinger, Radu Marinescu, Tom Mens, Bart Du Bois, Dirk Janssens, Stéphane Ducasse, Michele Lanza, Matthias Rieger, Harald Gall, and Mohammad El-Ramly. The LAN-simulation: A refactoring teaching example. In *IWPSE '05: Proceedings of the 8th International Workshop on Principles of Software Evolution*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[DTS99] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0—the FAMOOS information exchange model. Technical report, University of Berne, August 1999. `http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/famix20.pdf`.

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *OOPSLA '98: Proceedings of the 13th Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 341–361, New York, NY, USA, 1998. ACM Press.

[Ecl] Eclipse foundation. `http://www.eclipse.org`.

[EESV] Torbjörn Ekman, Ran Ettinger, Max Schäfer, and Mathieu Verbaere. Refactoring bugs. `http://progtools.comlab.ox.ac.uk/refactoring/bugreports`.

[Eet07] Niels van Eetvelde. *A graph transformation approach to refactoring*. PhD thesis, University of Antwerp, Universiteit Antwerpen, Belgium, April 2007.

[EJ04] Niels van Eetvelde and Dirk Janssens. Extending graph rewriting for refactoring. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT '04: Proceedings of the 2nd International Conference on Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 399–415, London, UK, 2004. Springer.

[ESJ02] Susan Eisenbach, Chris Sadler, and Vladimir Jurisic. Feeling the way through DLL hell. In *USE'02: Proceedings of the 1st Workshop on Unanticipated Software Evolution*, Lecture Notes in Computer Science, pages 404–428. Springer, June 2002.

[ESS02] Susan Eisenbach, Chris Sadler, and Shakil Shaikh. Evolution of distributed Java programs. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 51–66, London, UK, June 2002. Springer-Verlag.

[Eug06] Patrick Th. Eugster. Uniform proxies for Java. In Peri L. Tarr and William R. Cook, editors, *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 139–162, New York, NY, USA, 2006. ACM Press.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Donald Bradley Roberts. *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999.

[FCDR95] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95: Proceedings of the 10th Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.

[FD99] Ira R. Forman and Scott H. Danforth. *Putting metaclasses to work: a new dimension in object-oriented programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[FO95] Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley Professional, Reading, Massachusetts, 1995.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12:17–26, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Mattew Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading, Massachusetts, 1995.

[GJSB05] James Gosling, William Nelson Joy, Guy Lewis Steele, Jr, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, June 2005.

[GLW95] Susan L. Graham, Steven Lucco, and Robert Wahbe. Adaptable binary programs. In *TCON '95: Proceedings of the USENIX Technical Conference*, pages 26–30, Berkeley, CA, USA, 1995. USENIX Association.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.

[GS99] Thomas Genßler and Benedikt Schulz. Transforming inheritance into composition—a reengineering pattern. In *EuroPLoP '99: Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing*, 1999.

[Hau93] Franz J. Hauck. Inheritance modeled with explicit bindings: An approach to typed inheritance. In *OOPSLA '93: Proceedings of the 6th Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 231–239, October 1993. Printed as SIGPLAN Notices 28(10).

[HD05] Johannes Henkel and Amer 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.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *OOPSLA/ECOOP '90: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 169–180, New York, NY, USA, 1990. ACM Press.

[HJE06] Berthold Hoffmann, Dirk Janssens, and Niels van Eetvelde. Cloning and expanding graph transformation rules for refactoring. In *Electronic Notes in Theoretical Computer Science*, volume 152, pages 53–67, March 2006.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[II06] ISO (International Standard Organisation) and IEC (International Electrotechnical Commission). International Standard ISO/IEC 14764.:2006 (E) IEEE Std 14764-2006. In *ISO Catalogue of Published Standards*. ISO/IEC, 2006.

[Int] IntelliJ IDEA. `http://www.jetbrains.com`.

[IT03] Paola Inverardi and Massimo Tivoli. Software architecture for correct components assembly. In *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 92–121. Springer, 2003.

[JAC] JAC Project: Java Aspect Components. `http://jac.objectweb.org`.

[JBu] Borland JBuilder. `http://www.codegear.com/products/jbuilder`.

[JDT] Eclipse JDT: Java Development Tool. `http://www.eclipse.org/jdt`.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[JHo] JHotDraw homepage. `http://www.jhotdraw.org`.

[JO93] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *ISOTAS '93: Proceedings of the 1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Sciene*, pages 264–278, London, UK, November 1993. Springer.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In Andreas Paepcke, editor, *OOPSLA '92: Proceedings of the 7th Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 63–76, New York, NY, USA, 1992. ACM Press.

[Joh97] Ralph E. Johnson. Components, frameworks, patterns. In *SSR '97: Proceedings of the Symposium on Software Reusability*, pages 10–17, New York, NY, USA, May 1997. ACM Press.

[JZ91] Ralph E. Johnson and Jonathan M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(11):22–35, November 1991.

[KAT+08] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. Language-independent safe decomposition of legacy applications into features. Technical Report 02/2008, School of Computer Sience, University of Magdeburg, 2008.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[KETF07] Adam Kieżun, Michael D. Ernst, Frank Tip, and Robert Fuhrer. Refactoring for parameterizing Java classes. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 437–446, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[KH98] Ralph Keller and Urs Hölzle. Binary component adaptation. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.

[KKS96] Nils Klarlund, Jari Koistinen, and Michael I. Schwartzbach. Formal design constraints. In *OOPSLA '96: Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 370–383, New York, NY, USA, October 1996. ACM. Printed as SIPLAN Noticies 31(10).

[KL92] Gregor Kiczales and John Lamping. Issues in the design and specification of class libraries. In Andreas Paepcke, editor, *OOPSLA '92: Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 435–451, New York, NY, USA, 1992. ACM. Printed as SIGPLAN Noticies 27(10).

[Kni00] Günter Kniesel. *Dynamic object-based inheritance with subtyping*. PhD thesis, Universität Bonn, Institut für Informatik, Germany, 2000.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edition, December 2003.

[KS08] Hannes Kegel and Friedrich Steimann. Systematically refactoring inheritance to delegation in Java. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 431–440, New York, NY, USA, 2008. ACM.

[Lam93] John Lamping. Typing the specialization interface. In *OOPSLA '93: Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 201–214, New York, NY, USA, 1993. ACM. Printed as SIGPLAN Noticies 28(10).

[LAN] LAN-simulation lab session. `http://www.lore.ua.ac.be`.

[LB85] Meir M. Lehman and Laszlo A. Belady. *Program evolution: processes of software change*. APIC Studies in Data Processing. Academic Press Professional, San Diego, CA, USA, 1985.

[Leh96] Meir M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *Proceedings of the 5th European Workshop on Software Process Technology*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer-Verlag, October 1996.

[LHX94] Karl J. Lieberherr, Walter L. Hursch, and Cun Xiao. Object-extending class transformations. *Formal Aspects of Computing*, 6(4):391–416, 1994.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA '86: Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 21, pages 214–223, Portland, Oregon, USA, September 1986. ACM.

[Lis87] Barbara H. Liskov. Keynote address—data abstraction and hierarchy. In *OOPSLA '87 Addendum: Addendum to the Proceedings of the 2nd Conference on Object Oriented Programming Systems Languages and Applications*, pages 17–34, New York, NY, USA, October 1987. ACM. Printed as SIGPLAN Noticies 23(5).

[Log] Apache Log4j. `http://logging.apache.org/log4j`.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.

[MC02] Finbar McGurren and Damien Conroy. X-adapt: An architecture for dynamic systems. In *ECOOP'02 Workshop Reader: Report of the 7th Workshop on Component-Oriented Programming (WCOP '02)*, volume 2548 of *Lecture Notes in Computer Science*, pages 70–78. Springer Verlag, 2002.

[MEDJ05] Tom Mens, Niels van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[Mez97] Mira Mezini. Maintaining the consistency of class libraries during their evolution. In *OOPSLA '05: Proceedings of the 20th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–21, New York, NY, USA, 1997. ACM Press. Printed as SIGPLAN Notices 32(10).

[Min96] Naftaly H. Minsky. Law-governed regularities in object systems. Part 1: An abstract model. *TAPOS - Theory and Practice of Object Systems*, 2(4):283–301, 1996.

[MM83] James Martin and Carma L. McClure. *Software Maintenance: The Problem and Its Solutions*. Prentice Hall, 1983.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer, July 1998.

[Net] NetBeans. `http://www.netbeans.com`.

[NP90] John Teofil Nosek and Prashant C. Palvia. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.

[Obj05] Object Management Group. Unified Modeling Language: Infrastructure, version 2.0. `http://www.omg.org/cgi-bin/doc?formal/05-07-05`, July 2005.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1992.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[PLHM08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. *SIGOPS Operational System Review*, 42(4):247–260, 2008.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[PS83] Helmuth Partsch and Ralf Steinbrüggen. Program transformation systems. *ACM Computer Surveys*, 15(3):199–236, September 1983.

[Refa] Refactoring Browser. `http://www.refactory.com/RefactoringBrowser/index.html`.

[Refb] RefactoringCrawler homepage.
https://netfiles.uiuc.edu/dig/RefactoringCrawler.

[RH02] Stefan Roock and Andreas Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP '02: Proceedings of the International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pages 182–185, 2002.

[RH06] Ralf H. Reussner and Wilhelm Hasselbring. *Handbuch der Software-Architektur*. Dpunkt Verlag, 2006.

[Riv01] Jim des Rivières. Evolving Java-based APIs.
http://wiki.eclipse.org/Evolving_Java-based_APIs, 2001.

[RJ97] Donald Bradly Roberts and Ralph E. Johnson. Evolve frameworks into domain-specific languages. In Robert C. Martin, editor, *Pattern Languages of Program Design*. Addison-Wesley Professional, Reading, Massachusetts, 1997.

[Rob99] Donald Bradley Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1999.

[ŞA08] Ilie Şavga and Uwe Aßmann. Toward agile open-source framework outsourcing. In *MMSS '08: Model-Driven Modernization of Software Systems*, June 2008. Workshop co-located with the 4th European Conference on Model Driven Architecture.

[Sal] SalesPoint homepage. http://st.inf.tu-dresden.de/SalesPoint/v3.3.

[San88] Luis E. Sanchis. *Reflexive Structures: An Introduction to Computability Theory*. Springer, Heidelberg, Germany, October 1988.

[SEM08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In Gail E. Harris, editor, *OOPSLA '08: Proceedings of the 23rd Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 277–294. ACM Press, 2008.

[SG95] Raymie Stata and John Voegel Guttag. Modular reasoning in presence of subclassing. In *OOPSLA '95: Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 200–214. ACM, October 1995. Printed as SIGPLAN Noticies 30(10).

[ŞH08] Ilie Şavga and Florian Heidenreich. Refactoring in feature-oriented programming: Open issues. In *McGPLE '08: Proceedings of the 1st Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, number MIP-0802, pages 35–40. Department of Informatics and Mathematics, University of Passau, October 2008.

[SJM08] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 471–480, New York, NY, USA, 2008. ACM.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96: Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285, New York, NY, USA, October 1996. ACM. Printed as SIPLAN Noticies 31(10).

[Smu68] Raymond M. Smullyan. *First-Order Logic*. Dover Publications, New York, USA, 1968.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[SR02a] Heinz W. Schmidt and Ralf H. Reussner. Generating adapters for concurrent component protocol synchronisation. In Bart Jacobs and Arend Rensink, editors, *FMOODS: Formal Methods for Open Object-Based Distributed Systems*, volume 209 of *IFIP Conference Proceedings*, pages 213–229. Kluwer, 2002.

[SR02b] K. Chandra Sekharaiah and D. Janaki Ram. Object schizophrenia problem in object role system design. In *OOIS '02: Proceedings of the 8th International Conference on Object-Oriented Information Systems*, pages 494–506, London, UK, 2002. Springer.

[ŞR07a] Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 175–184, New York, NY, USA, 2007. ACM.

[ŞR07b] Ilie Şavga and Michael Rudolf. Refactoring-driven adaptation in evolving frameworks. In *WRT '07: Proceedings of the 1st Workshop on Refactorings Tools*, 2007. Printed as TU Berlin Technical Report ISSN 1436-9915.

[ŞR08] Ilie Şavga and Michael Rudolf. Refactoring-based adaptation of adaptation specifications. In *SERA '08: Best Papers Selection Proceedings of the 6th Conference on Software Engineering Research, Management and Applications*, volume 150 of *Studies in Computational Intelligence*, pages 189–203. Springer Verlag, 2008.

[ŞRG08] Ilie Şavga, Michael Rudolf, and Sebastian Götz. Comeback!: a refactoring-based tool for binary-compatible framework upgrade. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 941–942, New York, NY, USA, 2008. ACM.

[ŞRGA08] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[ŞRL08] Ilie Şavga, Michael Rudolf, and Jan Lehmann. Controlled adaptation-oriented evolution of object-oriented components. In C. Pahl, editor, *IASTED SE '08: Proceedings of the IASTED International Conference on Software Engineering*. Acta Press, 2008.

[ŞRS+07] Ilie Şavga, Michael Rudolf, Jacek Śliwerski, Jan Lehmann, and Harald Wendel. API changes—how far would you go? In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 329–330, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[ST09] Friedrich Steimann and Andreas Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In Sophia Drossopoulou, editor, *ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 2009.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA '87: Proceedings of the 12th Conference on Object-oriented Programming Systems, Languages and Applications*, pages 138–146, New York, NY, USA, December 1987. ACM. Printed as SIGPLAN Noticies 22(12).

[Sto81] Joseph Edward Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, September 1981.

[Str] Apache Struts. `http://struts.apache.org`.

[SVEM09] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In Sophia Drossopoulou, editor, *ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 369–393. Springer, 2009.

[Swa76] Burton E. Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Szy98] Clemens A. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, NY, 1998.

[TB01] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, January 2001.

[TBD06] Salvador Trujillo, Don S. Batory, and Oscar Díaz. Feature refactoring a multi-representation program into a product line. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 191–200, New York, NY, USA, 2006. ACM.

[TBKC07] Sahil Thaker, Don S. Batory, David Kitchin, and William Cook. Safe composition of product lines. In Charles Consel and Julia L. Lawall, editors, *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[Tip07] Frank Tip. Refactoring using type constraints. In Hanne Riis Nielson and Gilberto Filé, editors, *Software and Systems (SAS) Proceedings of the 14th International Symposium on Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2007.

[Ulr02] William M. Ulrich. *Legacy Systems: Transformation Strategies*. Prentice Hall PTR, 2002.

[Vli00] Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley, 2nd edition, 2000.

[WD06] Peter Weißgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[WDE98] David Wragg, Sophia Drossopoulou, and Susan Eisenbach. Java binary compatibility is almost correct. Technical Report 3/98, Imperial College, 1998.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *POPL '80: Proceedings of the 7th Symposium on Principles of Programming Languages*, pages 83–94, New York, NY, USA, 1980. ACM Press.

[Wer99] Michael Werner. *Facilitating Schema Evolution With Automatic Program Transformation*. PhD thesis, Northeastern University, July 1999.

[WSM06] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 37–56, New York, NY, USA, 2006. ACM Press.

[WST09] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In Hans van Vliet and Valérie Issarny, editors, *ESEC-FSE '09: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 173–182. ACM, 2009.

[Yin94] Robert K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, 1994.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM TOPLAS: ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.