Practical Refactoring-Based Framework Upgrade

Ilie Şavga Michael Rudolf Sebastian Götz Uwe Aßmann
Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany
{is13,s0600108,sebastian.goetz,ua1}@inf.tu-dresden.de

Abstract

Although the API of a software framework should stay stable, in practice it often changes during maintenance. When deploying a new framework version such changes may invalidate *plugins*—modules that used one of its previous versions. While manual plugin adaptation is expensive and error-prone, automatic adaptation demands cumbersome specifications, which the developers are reluctant to write and maintain. Basing on the history of structural framework changes (*refactorings*), in our previous work we formally defined how to automatically derive an adaptation layer that shields plugins from framework changes. In this paper we make our approach *practical*. Two case studies of unconstrained API evolution show that our approach scales in a large number of adaptation scenarios and comparing to other adaptation techniques. The evaluation of our logic-based tool ComeBack! demonstrates that it can adapt efficiently most of the problem-causing API refactorings.

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

General Terms Experimentation, Design, Languages

Keywords Frameworks, maintenance, adaptation, refactoring

1. Introduction

A software framework is a software component that embodies a skeleton solution for a family of related software products and is instantiated by modules containing custom code (*plugins*) [23]. Frameworks are software artifacts, which evolve considerably due to new or changed requirements, bug fixing, or quality improvement. If changes affect framework's Application Programming Interface (API), they may be *backward-incompatible* and invalidate existing plugins; 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 error-prone and expensive, the costs often becoming unacceptable in case of large and complex frameworks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM 978-1-60558-267-2/08/10. . . $5.00

To reduce the costs of component upgrade, a number of techniques propose to, at least partially, automate component adaptation [11, 13, 24, 30, 31, 41]. Most of them rely on and require additional adaptation specifications or annotations, which are used to intrusively update components' sources or binaries. In reality, developers are reluctant to write such specifications. Moreover, component evolution unavoidably demands to *maintain* these specifications, because evolutionary changes may alter component parts on which existing specifications rely, rendering the latter useless. In such cases, specifications must be updated correspondingly along with the change of the involved components, which raises the complexity and costs of component adaptation. Finally, intrusive upgrade of components may be impossible at all if their sources are unavailable or software licenses forbid their change.

Instead of directly upgrading plugins, we suggest to *protect* them from framework API changes by providing binary adapter layers that translate between the plugin and the latest framework version. The adapters [21] mimic the public types of the old framework version, while delegating to the types of the latest version. Our approach is based on the fact that, according to the case studies of Dig and Johnson [16], more than 80% of the application-breaking backward-incompatible changes in evolving frameworks are refactorings—behavior-preserving source transformations [29]. Common examples of refactorings include renaming classes and members to better reflect their meaning, moving members to decrease coupling and increase cohesion, adding new and removing unused members. Figure 1 shows the application of the `Extract-Subclass` refactoring to a framework class modeling network nodes.[1] If an existing plugin class *LAN* calling the method *broadcast* on an instance of *Node* is not available for update, it will fail to both recompile and link with the new *Node* version.

We treat refactorings as formal specifications of syntactic changes and use their trace to automatically derive adapters be-

[1] Our examples are inspired by the LAN simulation lab [7].

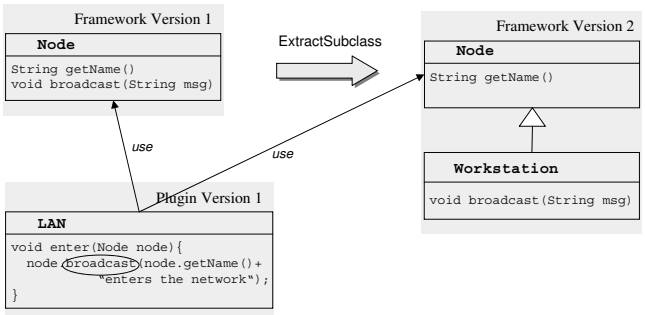

Figure 1. Broken plugin. The method *broadcast* is moved from *Node* to its new subclass *Workstation* and cannot be located from *LAN*.

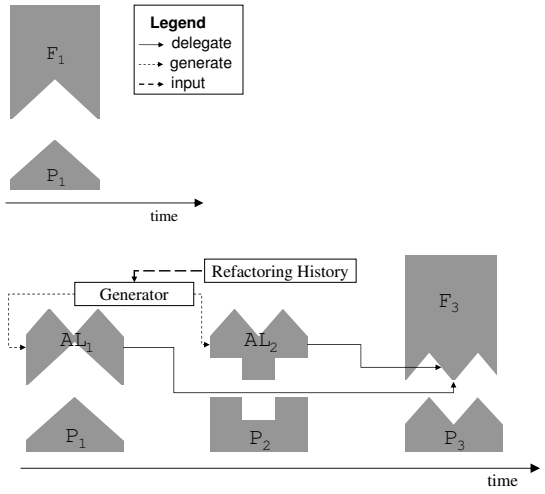

Figure 2. Refactoring-based plugin adaptation. In the upper part, the framework version F_1 is deployed to the user, who creates a plugin P_1. Later, two new framework versions are released, with F_3 as the latest one. While new plugins (P_3) are developed against the latest version, existing ones (P_1 and P_2) are preserved by creating adapter layers AL_1 and AL_2 (the lower part).

fore upgrading a framework (Fig. 2). The refactoring information required can be logged automatically (e.g., as Command objects in Eclipse [5]), or harvested semi-automatically [17, 39] and thus does not require additional specifications for most of the problem-causing changes [16] from developers. Our adaptation technology is not limited to consecutive framework versions: adapters can be generated for any of the previous framework versions. Moreover, the adapted plugins of different versions may co-exist and simultaneously use the latest framework version (*side-by-side execution* [10]), because each of them is provided with its version-specific adapter layer. In addition, adapters are optimized in that they are not stacked on top of each other; each adaptation layer delegates directly to the latest framework version reducing the performance penalties implied by delegation.

To ensure the soundness of our adaptation (no potentially breaking changes propagate to plugins), in our previous work [32] we proposed a *rigorous* plugin adaptation. We achieved soundness by formally defining the process of refactoring-based adapter derivation and proving that the resulting adapters do not change visible framework behavior, while compensating for API refactorings.

This paper discusses how we make our rigorous adaptation also *practical* by supporting refactoring-based binary-compatible upgrade of Java and .NET frameworks. Its main contributions are:

- **Re-exploration of plugin-breaking API changes**. To assess the feasibility and forsee the frontiers of refactoring-based adaptation we perform two case studies, in which developers are not restricted in how they evolve frameworks. Although in general such development may decrease the probability of refactorings, we show that when developers intend to improve the framework, most of API changes can be seen as refactorings. As another result of the case studies, we classify problem-causing API refactorings by the mechanics of the problems introduced.
- **Technology realization and delimitation**. We realize our technology in a rule-based tool ComeBack! designed to be platform-independent and easily combinable with other tools (e.g., for acquiring refactoring information). We support adaptation of both black- and white-box frameworks, also in the presence of callbacks typical for the latter. Our adapters are fairly efficient (at most 6.5% performance overhead implied by delegation) and preserve object identity. To support side-by-side plugin execution we developed dedicated linking policies in Java and .NET. Finally, we delimit the applicability of our approach by discussing the cases, when we either cannot adapt or require additional means for adaptation.

We discuss the case studies in Sect. 2, continue with the background of our rigorous refactoring-based adaptation in Sect. 3, elaborate on technology realization and limitations in Sect. 4 and Sect. 5, overview related work in Sect. 6, conclude the paper and outline future work in Sect. 7.

2. Re-Exploring Plugin-Breaking API Changes

The applicability of refactoring-based adaptation depends directly on the ratio of refactorings in the API evolution. To estimate the impact of framework refactorings on existing applications, Dig and Johnson [16] investigated the evolution of five big frameworks (e.g., Eclipse [5]). They discovered that most (from 81% up to 97%) plugin-breaking changes were refactorings. Since plugins are usually developed by third-party companies, they are 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 case of other evolving frameworks and thus anticipate the general feasibility of refactoring-based adaptation? To an extent, Dig attempts to answer 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." [15, pp. 33–34] In other words, if the framework developers are concerned with backward compatibility (and usually they are), they are likely to evolve the framework API through small changes, especially refactorings.

However, as an evolutionary activity,[2] maintenance often conflicts with preserving backward compatibility. Framework developers usually face a tradeoff between updating the framework's API and not breaking existing plugins. The degree to which backward compatibility must be preserved dictates the range of allowed API changes and varies depending on the development policies. To which extent can maintenance changes by themselves (not restricted by compatibility preservation) be modeled as refactorings? By answering this question we should be able to reason about the general feasibility of 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 frameworks. We address this question by performing two *critical* case studies.

According to Yin [42, 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. Arguably, framework maintenance that ignores backward compatibility implies unfavorable circumstances for refactorings to occur. Not being afraid of "clients abandoning the component," framework developers might not refrain from changing the framework API in a more drastic way.

[2] According to Lehman [26] we consider software maintenance a form of evolution.

Framework	Versions	Applications	Coverage
JHotDraw	5.2, 6.0, 7.0	4 (sample)	36% (15%)
SalesPoint	1.0, 2.0, 3.1	3 (student labs)	18% (21%)

Table 1. Frameworks selected for the case studies. Coverage reflects the usage of API classes (in parentheses, of methods) by the most advanced application selected.

If the results of such case study are similar to the ones reported by [16], we could expect the broad applicability of refactoring-based adaptation. Otherwise, we should at least be able to reason about the circumstances in which refactorings are less probable.

Study setup. We chose two open source Java frameworks with at least three major versions released (Table 1). The developers of both frameworks were never restricted in changing the framework API and performed all intended changes. For the first case study we used JHotDraw [6]—a well-known white-box framework for developing 2D structured editors. We used its three framework versions 5.2, 6.0 and 7.0 that JHotDraw developers confirmed to be evolved without preserving backward compatibility.[3] For the second case study we used SalesPoint [8] (versions 1.0, 2.0 and 3.1) developed at our department for teaching framework technologies. The framework models a purchase-selling business model with associated concepts, such as customer, shop, and catalog. Because one of the investigators has been directly involved in the development of SalesPoint, we can affirm that backward compatibility was never considered.

Because the change log of JHotDraw was too coarse-grained (most of the changes were undocumented) and there was no change log for SalesPoint, we had to perform *application-driven* change discovery. That is, 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 adaptation was performed by refactoring. For JHotDraw we used four sample clients (JavaDraw, Pert, Net and Nothing) delivered with the framework version 5.2. For SalesPoint we used two applications (student exercises) developed with the framework version 1.0 and another one developed with version 2.0.

For each breaking change detected we analyzed why it was applied by developers. In such a way we discovered exactly the backward-incompatible framework changes affecting the chosen applications and, if possible, modeled those changes as refactorings. Admittedly, we could not detect all breaking changes of the framework, because each application only used a subset of the framework API. Moreover, the same framework change could be detected repeatedly in different applications. However, our aim was not to find the total number of changes, but rather to understand why and how problem-causing changes occurred and to which extent they could be modeled as refactorings.

Results. We were not able to adapt JHotDraw sample applications 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 LoC its recompilation produced more than 60 errors, most of which could not be adapted manually.

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

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. Types of plugin-breaking API changes by their intent. For each type the overall number of changes detected (in parentheses, of refactorings) is specified.

The results for the other JHotDraw and two SalesPoint versions are summarized in Table 2.[4]

- **Shift of responsibility.** Several changes in SalesPoint shifted responsibilities among API classes. For instance, in SalesPoint 1.0 each *Catalog* is saved separately; in SalesPoint 2.0 all catalogs must be attached to a *Shop* that is responsible for saving.

  Such responsibility reallocation could usually be modeled as refactorings (e.g., moving method *save()* from *SalesPoint* to *Shop*) combined with changes of *protocols* (i.e., message exchange [9]) between the framework and plugins. Although typically not affecting the actual framework functionality (e.g., ability to store catalogs), protocol changes introduce permuted message sequences, surplus, or absence of messages between communication parties [12]. In the example of SalesPoint, the protocol change is the absence of the method call to *Shop.attach(Catalog)* attaching catalogs before storing the whole shop.

- **Concept addition.** Adding a new API concept usually implied a set of additive refactorings crosscutting the API. 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.** Several previously existing API concepts were refined to better reflect their semantics and extend their specific functionality. This implied, for instance, renaming methods to better reflect their meaning and extracting interfaces to separate them from implementing classes. In other cases, refinement implied changing from Java built-in types to framework-specific user-defined types. For example, in SalesPoint 2.0 a new class *NumberValue* modeling all kinds of numerical values replaced plain (string and integer) method parameters and return values of existing methods dealing with such values. In JHotDraw 6.0 the Java *Enumeration* was replaced by the framework-specific *FigureEnumeration* that enabled further addition of concept-specific methods (e.g., *hasNextFigure()*). Although we were able to compensate for this change, we did not consider it a refactoring, because of the lack of semantics of such transformations.

  An advanced case going beyond refactorings was replacing one SalesPoint concept with a completely different one with richer functionality. In SalesPoint 1.0 the concept of transaction was modeled by *Transaction*—a thread that could be suspended and resumed. In SalesPoint 2.0 it was replaced by *SaleProcess* to

[4] The protocol of case studies including detailed statistics is available at `http://comeback.sourceforge.net/protocol.pdf`.

explicitly model a state machine with states (*Gate*) and transitions (*Transition*). 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.** In SalesPoint 2.0 the class *User* serves 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 of *Customer* exists in class *SalesPoint*, they removed *Customer*. All changes involved in adapting plugins but one could be modeled as refactoring to call the (semantically equivalent) methods in *SalesPoint*; the exception was a protocol change to call a sequence of three methods instead of one initial method.
- **Refactoring to patterns.** Class *MenuSheet* of SalesPoint 2.0 was restructured to follow the Composite pattern—a refactoring to pattern [25] that involved a set of refactorings to participating types, such as class and method addition, method rename and generalization of argument types.
- **Language evolution.** In SalesPoint 2.0 the parameter type of the *FormSheet* constructor changed from *Panel* of AWT to *JComponent* of Swing. This change was visible in the API and could not be considered 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()*).

Discussion. Although backward compatibility was not considered, our pattern of changes discovered (about 85% consisting of refactorings) repeated the pattern reported by Dig and Johnson for more conventional framework evolution. 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 framework's API (e.g., replacing *Transaction* by *SalesProcess*), developers first analyze existing 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 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* [35, 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 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 [38] 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.

We must admit that our results are not ripe to stronger postulate a general feasibility of refactoring-based adaptation. Possibly, there is a number of frameworks for which their maintenance does not repeat the pattern of refactorings discovered. For example, compared 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 built-in types, the changes of which (when upgrading the implementation language) are not encapsulated and cannot be modeled as refactorings. Because we did not know the details of the development process (e.g., how Eclipse was used to evolve the frameworks), we could not estimate the impact of using an IDE's refactoring engine on the ratio of refactorings discovered.

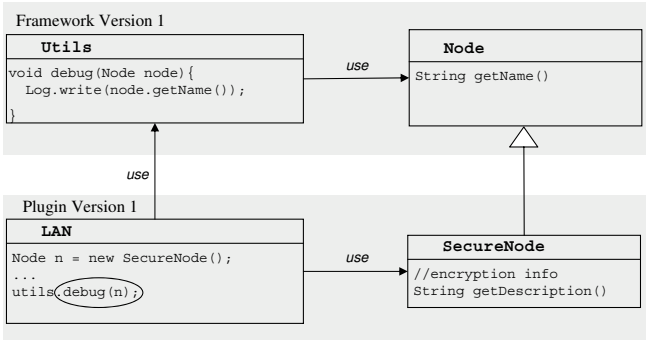

Figure 3. Before Method Capture. Node info debugged correctly.

However, what makes us optimistic is the fact that our results were obtained in critical case studies. Preserving backward compatibility will furthermore increase the probability of refactoring. Breaking changes applied otherwise in unrestricted API maintenance will be avoided to the greatest extent possible. 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 to translate from the old to the new API. In such cases, most API changes are small enabling refactoring-based adaptation.

Classification of Plugin-Breaking API Refactorings

While performing the case studies we noticed that plugin-breaking API refactorings differed in the mechanics of the problem they introduced. In general, after refactoring the framework's API, certain functionality cannot be found by a caller (i.e., the framework in case of callback types, or plugins otherwise) because of:

- **Misplaced functionality.** Required functionality exists, but cannot be located by the compiler or the linker or both. Examples are changing method signatures, moving methods, and renaming classes. Such changes lead to the *syntactic mismatch* [12] between the framework and plugins.
- **Missing functionality.** Required functionality does not exist: either it was removed (e.g., deleting a framework method) or has not been introduced yet (e.g., adding an abstract hook method for which no implementation in plugins exists). The latter case is also known as Unimplemented Method [37].
- **Unintended functionality.** Found functionality is not the one intended. In the presence of dynamic linking, new methods added to framework classes may be overridden accidentally by existing methods of plugin subclasses—the so-called Method Capture problem [37]. Consider class *SecureNode* of plugin version 1 that subclasses *Node* of the framework (Fig. 2) and encrypts/decrypts messages. Its method *getDescription()* returns the details of the encryption strategy used. The *LAN* class of the plugin uses the framework's *Utils* class to debug node info. In the next framework version (Fig. 2), developers add a hook method to *Node* allowing its subclasses to include node-specific

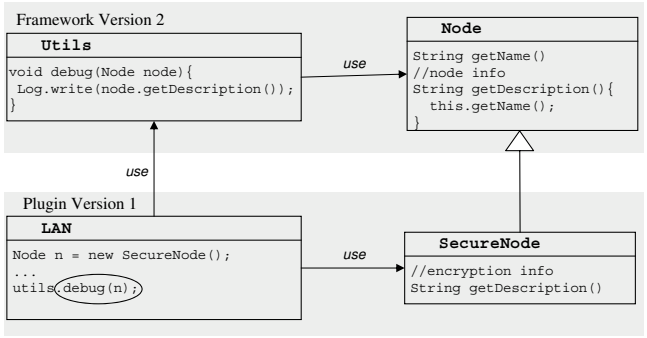

Figure 4. After Method Capture. Instead of node info, for *SecureNode* its encryption description is unintentionally retrieved.

information in the description. By default, this hook method (accidentally called *getDescription()*) delegates to *getName()*, that is, the two are semantically equivalent. In addition, framework developers refactor *Utils* to call *getDescription()*. Now the *LAN*'s call to *Utils.debug()* with the instance of *SecureNode* will produce unexpected behavior, logging the description of the encryption algorithm and not of the node.

In general, any refactoring introducing new or changing existing classes and methods (e.g., adding and extracting methods, changing method signatures, moving methods) may lead to the well-known Fragile Base Class Problem (FBCP) [28], by which apparently safe modifications to a base API class in a white-box system may cause the derived classes to malfunction.

3. Background on Rigorous Refactoring-Based Adaptation

To automatically construct adapters compensating for API refactorings and ensure their soundness, in [32] we formally define our refactoring-based adaptation. Effectively, we roll back the changes introduced by framework refactorings by executing their inverses. We cannot inverse directly on framework types, because we want plugins to use the latest, improved framework. Instead, we create adapters (one for each framework API type) and then inverse refactorings on adapters. We call these inverses *comebacks*.

Technically, a comeback is realized in terms of refactoring operators executed on adapters. It is defined as a template solution and instantiated to an executable specification by re-using parameters of the corresponding refactoring. For some refactorings, the corresponding comebacks are simple and implemented using a single refactoring. For example, the comeback that corresponds to the refactoring `RenameClass(name, newName)` renames the adapter to the old name. As another example, the comeback of `AddClass` is defined by `RemoveClass` removing the adapter. 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. 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`.

In practice, we focus on the adaptation of user-defined API types (e.g., *Node*). Figure 5 shows the workflow of refactoring-based plugin adaptation performed before upgrading a framework to the version F_n. First, we create the adaptation layer AL_n (the right part of the figure). For each user-defined API class of the latest framework version we provide an adapter class with exactly

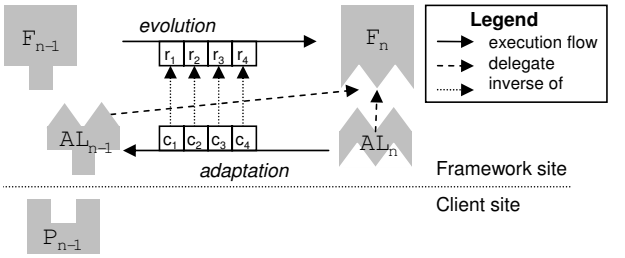

Figure 5. Adaptation workflow. To a set of refactorings (r_1–r_4) between two framework versions (F_{n-1}, F_n) correspond comebacks (c_1–c_4). Comebacks are executed on the adaptation layer AL_n backwards to the framework refactorings. The resulting adaptation layer AL_{n-1} delegates to the new framework, while adapting plugins of version P_{n-1}.

the same name and set of method signatures. Each adapter delegates to its corresponding framework class, which becomes the adapter's delegatee. Once the adapters of AL_n are created, the actual adaptation is performed by executing comebacks derived using the description of the corresponding refactorings. The comebacks are executed *backwards*, that is, starting with the one for the last refactoring and continuing in the reverse order of the initial refactoring execution. When all comebacks for the refactorings recorded between the last and the previous framework version F_{n-1} are executed, the produced adaptation layer AL_{n-1} reconstructs the API of F_{n-1}, while delegating to the newest framework version. The framework versions need not be consecutive: given version number F_{n-2} and the refactoring history between F_{n-2} and F_n, the adapter layer AL_{n-2} delegating to F_n will be derived in the same manner.

4. Practical Refactoring-Based Adaptation

We are implementing our adaptation tool ComeBack! [3] using a Prolog logic programming engine. 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. To support Java and .NET binaries we developed the corresponding Prolog/Java and Prolog/CIL parsers. After examining the history of framework refactorings, the corresponding comebacks are loaded into the engine and executed on the fact base as described in Sect. 3. Once all comebacks have been executed, the fact base contains the information necessary for generating adapters (it describes the adapters) and is serialized to the adapter binaries using reflection in .NET and the ASM code generation library [1] in Java. Thereby we extract and transform the information about the program and not the program itself; the adapter generation using this information is the final step.

4.1 Insights into Binary Adapters

To ensure type safeness in the presence of adapters, we perform *exhaustive adaptation*, that is, every (user-defined) API type of the latest framework version is wrapped into its corresponding adapter and there is no standard way for a plugin to bypass the adaptation layer. An important implication is that at run-time plugins cannot observe any framework type (e.g., returned by a method call) and the framework cannot observe any plugin type (e.g., a callback): in their communication both parties use only adapter and built-in types. This is achieved by *wrapping* and *unwrapping* of framework types inside adapters. For method calls on ordinary (non-

comeback) types, adapters internally wrap (adapt) all user-defined method return values before sending them to plugins and unwrap (adapted) method arguments before sending them to the framework. For callbacks and extended framework classes, the opposite way of wrapping/unwrapping is performed. Since exception types also represent a (possible) return type, they are wrapped similarly.

The pseudocode of Listing 1 summarizes the (simplified) algorithm used for wrapping and unwrapping of types. The full algorithm coping with collections, arrays and exception chaining is available on the tool's homepage [3]. The helper functions used are:

- **`isAdapter`**`(Object)` checks whether the given object is an adapter. The type information needed is added as annotations (calling *Class.isAnnotationPresent(Class)* in Java or adding *attributes* in .NET) when generating adapters.
- **`isUserDefined`**`(Class)` checks whether the given class is defined in the framework. If not, it is considered to be a built-in type. A special handling is required for third-party libraries not subject to adaptation (currently not supported).
- **`wrap`**`(Object)` replaces the given object with an instance of its corresponding adapter. It retrieves the corresponding adapter, if any, from the adapter cache, or creates a new one otherwise.
- **`unwrap`**`(Adapter)` is a placeholder for an access (either via a method call or via direct field access) to the adapter's delegation variable.

Listing 1 *Anatomy of a delegating adapter method.*

```
<AccessModifier> <ReturnType> Name(Parameters) {
  //(un-)wrap the arguments
  foreach (argument in Parameters) {
    if (argument != null) {
      if (isAdapter(argument)) {
        unwrap(argument);
      } else if (isUserDefined(argument)) {
        wrap(argument);
      }
    }
  }

  Object value;
  try { //cast to issue non-virtual call (.NET)
        //otherwise, use reflection (Java)
    value = ((TargetType)delegatee).Name(arguments);
  } catch(Throwable t) {
    //(un-)wrap the exception object
    if (isUserDefined(t)) {
      AdaptedThrowable at = wrap(t);
      throw at;
    } else if (isAdapter(t)) {
      t = unwrap(t);
      throw t;
    } else {
      throw t;
    }
  }

  //(un-)wrap the return value
  if (value != null) {
    if (isAdapter(value)) {
      value = unwrap(value);
    } else if (isUserDefined(value)) {
      value = wrap(value);
    }
  }
  return value;
}
```

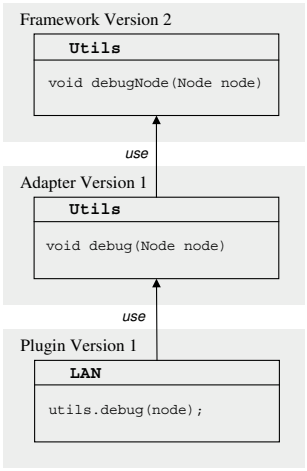

Figure 6. Black-box Class Adaptation.

Adapter structure is influenced by the way framework types are reused. In black-box reuse, *class* adapters are merely reconstructed old classes that forward methods to new framework classes (Fig. 6). In white-box reuse, class adapters must also access the plugins' overriding implementation, while avoiding FBCP problems discussed on page 5. Figure 7 shows the adapter for the framework's *Node* class of our running LAN example. The adapter consists of two classes: public *Node* to serve as the superclass for plugin subclasses and internal *NodeDispatcher* to dynamically dispatch on plugin's overriding methods called from the framework. Whenever an instance of a *Node*'s subclass (e.g., *SecureNode*) is passed to the framework, it is wrapped into *NodeDispatcher*. The latter investigates the plugin and, depending on what methods are intentionally overridden by the plugin, dispatches method calls either to the plugin or to the framework. Method Capture (Fig. 2) is solved by the comeback of `AddMethod` deleting the *getDescription()* method from *Node* adapter; the method is no longer dynamically dispatched to *SecureNode*. Since intentionally overridden hook methods appear in plugins after they appear in the framework's API, no corresponding entry exists in the refactoring log (hence, no comeback is executed and these methods are found dynamically).

A similar adaptation strategy applies to reconstructing refactored API *interfaces*. If a black-box framework instantiates a type

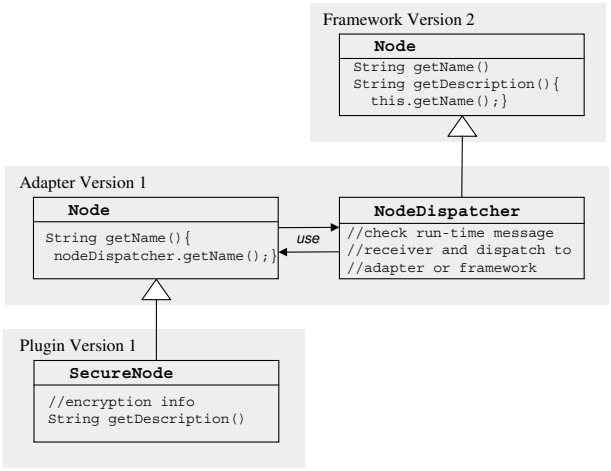

Figure 7. White-box Class Adaptation.

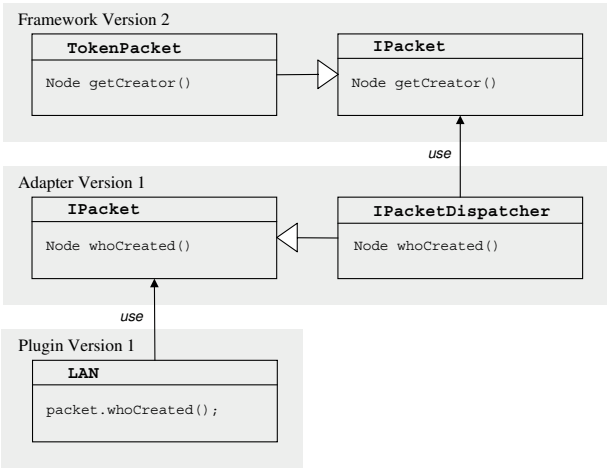

Figure 8. Black-box Interface Adaptation.

implementing the interface (e.g., in a factory method) and returns it to the plugin, the adapter wraps up the (new) framework implementation and "hides" it behind the reconstructed old interface (Fig. 8). If, however, plugins implement the interface provided and used by the framework (*callback* type), then the adapter reuses the plugin's implementation of the old interface and presents it to the framework as an implementation of the actual interface (Fig. 9). As in general we cannot derive from the API the type of use (e.g., how a public framework class is used or which party implements an interface), we generate adapters for both black- and white-box cases—appropriate adapters are instantiated at run-time automatically in the adapter layer depending on the call direction.

4.2 Custom Linking Policies

In our approach, all adapter layers are deployed as binary components with exactly the same name as the framework they delegate to. Although the framework type names are embedded in the plugin binaries, because the names of the (old) framework and the adapter components are the same, the dynamic linking mechanism of both Java and .NET still works. From the point of view of a single plugin, the framework upgrade is replacing one component by another component with the same name: an old version of either the framework itself or an existing 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 [19]. 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.

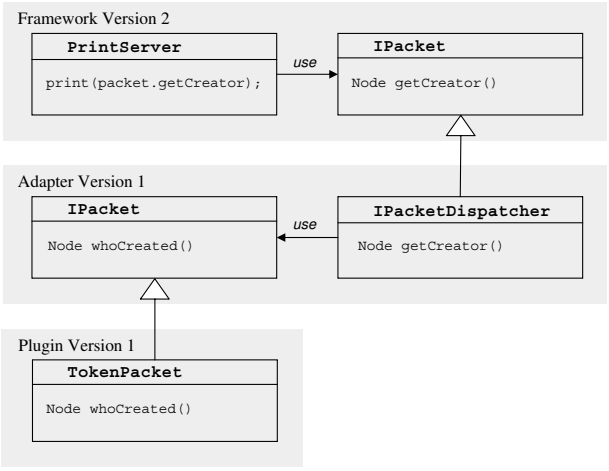

Figure 9. White-box Interface Adaptation.

However, as adapters can be generated for any previous framework API (Fig. 2, page 2), plugins of different versions *simultaneously* use the framework either directly or through the corresponding adapter components. That is, an application may encompass plugins developed with different framework versions. Because all adapter components and the framework have the same name (say, F) differing only in version numbers, we must also support *side-by-side execution* [10]—when multiple versions of F are deployed, any existing component has to find its "right" version of F. In particular, the run-time system must differentiate among versions of F to achieve:

- *linking plugins* that use their corresponding adapter components, and
- *linking adapters* that use the latest version of F (i.e., the framework itself).

Not meeting these two requirements usually results in linker errors. In the former case, a plugin will attempt to use types or methods not present in another version of F. As an example of the latter, more subtle case, assume for Fig. 6 on page 6 an adapter *Utils* is created that has as its delegatee the class *Utils* of the latest framework version. When the adapter type is loaded and its dependencies are resolved, the linking mechanism of both Java and .NET will fail, because the type of the delegation field is considered to be the adapter and not the framework type.

In .NET the system looks for each program's component in a default location called the global assembly cache (GAC), where components intended to be shared are installed. We rely on the fact that the system is able to distinguish different versions of the same component in the GAC, if to each component's deployment unit (assembly) a strong name is assigned—an encrypted label of its four-part version number. For *linking plugins* it is enough to sign the adapter and to direct the binding of its plugins in an application-specific XML configuration file that contains the version number of the adapter assembly to be used. For *linking adapters* the names of the framework types to be created inside adapters are prepended (in CIL, at adapter generation time) the strong name of the framework assembly.

In Java our solution is based on generating custom classloaders and uses the fact that, since classes are identified in the Java virtual machine by their names *and* the classloader that loaded them, classes with the same names loaded by separate classloaders can be distinguished effectively. For *linking plugins* the old framework JAR file is replaced by the generated adapter JAR file, so that the plugin's class path does not need to be adjusted. For *linking adapters* each adapter gets a custom classloader locating the latest framework version and uses this classloader whenever an instance of a framework type needs to be created reflectively.

4.3 Tool Evaluation

Functionality. The following refactorings are currently supported by our tool: `AddPackage`, `RenamePackage`, `MovePackage`, `AddClass`, `RenameClass`, `ExtractSuperclass`, `ExtractSubclass`, `ExtractClass`, `ExtractInterface`,

`RenameInterface`, `AddMethod`, `RenameMethod`, `MoveMethod`, `PushDownMethod`, `PullUpMethod`, `ExtractMethod`, `AddParameter`, `RemoveParameter`, `AddCheckedException`, `DeleteCheckedException`, `AddAbstractHookMethod`, `DeleteMethod`. For the latter three refactorings we provide a default (empty) implementation, which can be changed on demand.

This comeback library can adapt all refactorings discovered by Dig and Johnson [16] (reported in detail in Dig's dissertation [15, p. 21]), except for moving and renaming fields. It can also adapt all refactorings discovered in our case studies, because all API fields of JHotDraw and SalesPoint are encapsulated.

In our case studies, we made the most advanced sample application of JHotDraw execute with version 6.0 of the framework and one of the student applications of SalesPoint execute with the latest framework version. Since after our application-based change discovery we knew exactly how the adaptation code for all changes should look like, we used this knowledge to manually adapt changes beyond refactorings. First, we executed all comebacks and got the adapter fact base. Then, we augmented it with 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. We discuss how to automate this currently manual adaptation in the conclusion (Sect. 7).

***Efficiency*.** The delegation used in our approach inevitably implies performance penalties of at least one message redirection call per method. The penalties increase when user-defined types are present in method signatures and, hence, need to be wrapped and unwrapped as described in Sect. 4.1. In such cases, in addition to constructor calls and cache lookups (for wrapping) and accessor calls (for unwrapping), time is also consumed for type investigation (i.e., **isAdapter** and **isUserDefined**). For collection types this is especially important, because each element of a collection needs to be investigated and possibly wrapped/unwrapped.

However, we are able to considerably reduce the performance penalties by generation- and run-time optimizations. At generation-time, since all final (e.g., *java.lang.String*) and value (e.g., *int*) built-in types in the method signatures need no wrapping and unwrapping, the checking code of Listing 1 for them is not emitted decreasing the overhead. At runtime, we provide a fast adapter cache (implemented as a hash map) to speed up the dynamic adapter lookup when wrapping (and to avoid object schizophrenia, as discussed in Sect. 5). Since strong references in the adapter cache may hinder garbage collection, we use weak references to the framework objects cached and avoid thus memory leakage (i.e., unused objects remaining in memory). Whenever a framework object and its adapter are not used anymore they get collected automatically. For the cache to work correctly, it is required that the methods *hashCode()* and *equals()* are overridden according to the rules in the Java class documentation.

We performed the performance benchmarking under Windows Vista 32bit on an Athlon Turion TL60 (dual core with 2GHz each, 64bit) and 2GB RAM. The measurements were obtained by simple manual code instrumentation (to print the elapsed time). For the adapters generated in the JHotDraw case study, the overhead reached 6.5%, because of the collection types commonly present in method signatures. At the same time, the SalesPoint adapters implied less then 4.5% of overhead. Comparing to Java, the overhead of adapters we generated for the .NET version of SalesPoint was 10–15% less because of the more efficient way the linking policy is implemented (Sect. 4.2).

5. Delimiting the Technology

For certain refactorings, comebacks cannot be defined because of limitations of their execution context (i.e., adapters). For instance, one cannot define comebacks for field refactorings, because fields are not accessible in the adapters. In our approach, we require that all API fields are encapsulated (accessed by get/set methods), so that support for field refactorings is implied by the adaptation of the corresponding accessor methods. We also cannot implement comebacks for `InlineMethod` and `InlineClass`, which require adaptation means (e.g., to access *this*) beyond the adapters' message forwarding. However, adapters suffice for the reverse refactorings `ExtractMethod` and `ExtractClass`.

Delegation as a reuse mechanism introduces *object schizophrenia* [36] due to the dichotomy of the adapter and adaptee objects: what should appear as a single object is actually broken up into two or more, each possessing its own identity, state, and behavior. In our case, if not handled properly, adapters may expose their own object identity different from the corresponding framework objects (i.e., adaptees). As mentioned in the description of the **wrap** function on page 6, an adapter cache guarantees that there is at most one adapter instance for a given framework object, ensuring that a framework object possibly referred to from multiple contexts will always be wrapped in exactly the same adapter. So far we did not evaluate this approach in the presence of object serialization.

Some problems may be caused by differences in the structure of adapters containing adaptation-specific members (e.g., the delegation field) and of original framework types. Assume an instance of a framework class is serialized and then in a new framework version the class is adapted. The default serialization mechanism of both .NET (*System.Runtime.Serialization* namespace) and Java (*java.io.Serializable*) fail to restore the serialized instance due to the structural mismatch between the adapter and the original class. Therefore, whenever serialization is needed, a custom serialization process has to be provided to properly initialize the adapter. In .NET this is achieved by implementing the interface *ISerializable* and in Java by providing the private instance methods *readObject()* and *writeObject()*.

These structural differences may also lead to run-time problems, in case developers misuse reflective method calls to the framework API types (e.g., by relying on the relative order of methods). Since adapters introduce new methods (e.g., adapter constructors) and may change the initial method order, a wrong method may get called. In [34] we identify several cases that may potentially invalidate our adaptation approach and suggest appropriate solutions.

The main requirement of our approach is the availability of the totally ordered refactoring history. For Java it comes "for free" in Eclipse and JBuilder. We developed a query facility that reads in and serializes the Eclipse log into a set of corresponding Prolog facts to be re-used in ComeBack!. Since there is no similar tool for .NET, we developed a prototypic annotation language [40] to demarcate applied refactorings in the C# code. The task of manual annotation is alleviated by a version control system that silently assigns unique identifiers to API types and signatures and is able to detect some refactorings automatically (e.g., class and method rename, method move). For other refactorings (e.g., method extraction, addition of method parameters) it prompts developers for additional information. In case of framework branching, the total order of the refactoring trace is achieved by merging separately recorded refactorings [15, p. 76].

6. Related Work

In general, three main groups of approaches cope with component change in case depending applications are not available for analysis and update.

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 instance, des Rivières enumerates which changes of Java components will not break applications and under which conditions [14]. Mikhajlov and Sekerinski [28] formally define conditions and prescribe a set of requirements to avoid the Fragile Base Class Problem. However, some of these requirements (e.g., never start or stop calling a new method from an existing base class) are too restrictive for framework evolution.

Prevention. A group of approaches relies on the use of a legacy middleware [2, 4] or, at least, a specific communication protocol [20, 27] to interconnect components. The middleware prevents applications from observing component changes, often in a transparent manner. However, most of the compensated changes are trivial (additive changes), with more complex ones being prohibited by the middleware. Moreover, such approaches imply 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.

Facilitation. The third group consists of approaches to distribute the change information and facilitate the remote component update. In most of the cases, component developers have to manually provide such information either as different annotations within the component's source code [11, 13, 31] or in a separate specification [24, 30, 41]. Annotations are then used by a transformation engine to adapt the old application code. The main advantage of these approaches, also comparing to our approach, is their power: given the required specifications, potentially any kind of change can be adapted. However, in case of large and complex legacy applications the cumbersome task of writing specifications and adaptation rules is expensive and error-prone. Moveover, once written, specifications need to be maintained along the component evolution.

The idea of recording the refactorings applied to a software component for their later application to the client code is the basis of CatchUp! [22] that is able to listen for, capture and record refactorings applied in Eclipse. The recorded refactorings are then "replayed" on the application code to synchronize it with the changed component. However, the tool fails when a refactoring cannot be played back in the application context (e.g., if the renaming of a component method introduces conflicts with some application-defined method). Furthermore, this intrusive way of update requires available application sources and implies a new application version for each component upgrade.

Instead of adapting clients, in [18] Dig et al. adapt the framework (or software library). For each breaking refactoring they define a compensating refactoring that *inlines* the corresponding code directly in the library. For example, for `RenameMethod(oldMd, newMd)` compensates `AddMethod(oldMd)` inserting the method that delegates to `newMd`. Given an old library and a refactoring trace, they execute the compensating refactorings in the same order as the original refactorings. Effectively, instead of putting a wrapper around the library, they give it two (the old and the new) interfaces. Similarly to our approach, object identities are preserved and the side-by-side execution is supported. Performance penalties are reported to be less then 1%. Moreover, having access to the old implementation they can recover deleted methods. However, their adaptation does not handle refactorings contradicting each other in the scope of a class (e.g. deleting a method *M* and then renaming another one to *M*). Moreover, their implementation is Java-centric, cannot be re-used for other languages and supports only several refactorings (e.g., no interface refactorings). In addition, the tool cannot cope with the FBCP (Sect. 2). Finally, the compensating refactorings are defined ad hoc, implementation-specific, and not formally validated.

7. Conclusions and Future Work

We stipulate that treating refactoring as a specification of syntactic changes it is possible to support sound and practical adaptation of most of the problem-causing API changes. In this paper we discuss how our refactoring-based adaptation scales. In two case studies of unconstrained API evolution we show that, when developers want to improve the framework, most of their changes are refactorings. For certain API changes breaking plugins of white-box frameworks with dynamic linking we argue that our adapter-based approach is a more appropriate short-term solution than invasive adaptation. On the long run, if it is possible to carefully analyze, update, and test plugins, one could switch to invasive adaptation.

The binary adaptation performed by our tool ComeBack! is efficient and unobtrusive for existing plugins: they need neither manual adaptation nor recompilation. It is also undemanding by alleviating the process of writing and maintaining adaptation specifications. Moreover, the refactoring-based adaptation not only reduces the costs and improves the quality of framework upgrade but also relaxes the constraints on the permitted API changes allowing for more appropriate framework evolution.

We will extend our tool by combining it with protocol adaptation [33]: given a valid protocol adaptation specification, it is converted into corresponding facts and rules inside ComeBack! that generate the adaptation code.

References

[1] ASM homepage. `asm.objectweb.org`.

[2] Microsoft COM homepage. `www.microsoft.com/Com/default.mspx`.

[3] ComeBack! homepage. `comeback.sf.net`.

[4] CORBA homepage. `www.corba.org`.

[5] Eclipse homepage. `www.eclipse.org`.

[6] JHotDraw homepage. `www.jhotdraw.org`.

[7] LAN-simulation lab session homepage. `www.lore.ua.ac.be`.

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

[9] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[10] Dennis Angeline. Side-by-side execution of the .NET framework. `msdn2.microsoft.com/en-us/library/ms994410.aspx`, December 2002.

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

[12] 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.

[13] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 359–368, Washington, DC, USA, 1996. IEEE Computer Society.

[14] Jim des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/Evolving_Java-based_APIs`, 2001.

[15] Daniel Dig. *Safe Component Upgrade*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, October 2007.

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

[17] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *ECOOP'06: European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[18] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *ICSE'08: International Conference on Software Engineering*, pages 441–450, Leipzig, Germany, May 2008.

[19] Susan Eisenbach, Chris Sadler, and Vladimir Jurisic. Feeling the way through DLL hell. In *ECOOP'02: European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 404–428. Springer, 2002.

[20] Ira Richard Forman, Michael H. Conner, Scott Harrison Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[22] 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.

[23] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June 1988.

[24] Ralph Keller and Urs Hölzle. Binary component adaptation. In *ECOOP'98: European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[25] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[26] Meir M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *Software Process Technology – Proceedings of the 5th European Workshop*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, October 1996.

[27] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361, London, UK, 2000. Springer.

[28] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. *Lecture Notes in Computer Science*, 1998.

[29] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[30] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. *SIGOPS Operational System Review*, 2008.

[31] Stefan Roock and Andreas Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02: Proceedings of Extreme Programming Conference*, pages 182–185, 2002.

[32] Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatiblity in evolving frameworks. In *GPCE'07: Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, pages 175–184, Salzburg, Austria, October 2007. ACM.

[33] Ilie Şavga and Michael Rudolf. Refactoring-based adaptation of adaptation specifications. In *SERA'08: Best Paper Selection Proceedings of Software Engineering Research, Management and Applications*, Prague, Czech Republic, August 2008. Springer.

[34] Ilie Şavga, Michael Rudolf, and Jan Lehmann. Controlled adaptation-oriented evolution of object-oriented components. In *IASTED SE'08: Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, February 2008. ACTA Press.

[35] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[36] 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.

[37] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285. ACM Press, October 1996.

[38] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, January 2001.

[39] 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 (ASE'06)*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Ulf Wemmie. Using version control system for tracking adaptation-relevant software changes. Bachelor Thesis. Dresden University of Technology, December 2006.

[41] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM TOPLAS: ACM Transactions on Programming Languages and Systems*, 1997.

[42] Robert K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, December 1994.