CONTROLLED ADAPTATION-ORIENTED EVOLUTION OF OBJECT-ORIENTED COMPONENTS

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

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

Jan Lehmann
Comarch Software AG
Chemnitzer Str. 50, 01187 Dresden, Germany
email: jan.lehmann@comarch.com

ABSTRACT
By introducing syntactic and semantic changes, the upgrade of a software component may invalidate existing applications that use one of its previous versions. Existing adaptation approaches to compensate for such changes rely on and, hence, are limited to certain change specifications. In addition to using an adaptation technology, the developer needs to be guided in the way the component should be evolved in order to enable automatic adaptation and avoid semantic inconsistencies. Based on our experience, we describe problems common to different adaptation techniques and give advice on how to control yet not restrict component evolution.

KEY WORDS
Refactoring, Adaptation, Software Components.

1 Introduction

If a component of an existing application is upgraded, its new version may not cooperate with other components as intended, thus breaking the application. To achieve component *backward-compatibility* (i.e., its new version can substitute a previous one without affecting existing applications), the developer is forced either to limit the changes applicable to a component or to manually solve component incompatibilities. For example, consider a software framework — a software component that embodies a skeleton solution for a family of related software products and is instantiated by modules containing custom code (*plugins*) [1]. A framework may evolve considerably due to new requirements, bug fixing, or quality improvement. As a consequence, existing plugins may become invalid; that is, their sources cannot be recompiled or their binaries cannot be linked and run with a new framework release. Framework maintainers must either restrict themselves to a limited set of repair changes or manually adapt plugins. The former decreases the value of component evolution quickly resulting in software entropy [2], the latter is usually expensive and error-prone.

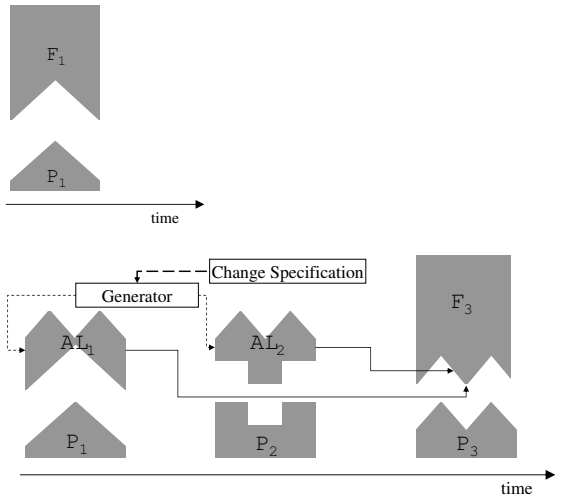

Figure 1. Plugin adaptation in an evolving framework. In the left part, the framework version F_1 is deployed to the user, who creates a plugin P_1. Later, two new framework versions are released, with F_3 as the latest one. While new plugins (P_3) are developed against the latest version, the existing ones (P_1 and P_2) are preserved by creating adapter layers AL_1 and AL_2.

To preserve backward-compatibility, while still permitting developers to apply a wide range of evolutionary changes, a number of approaches (e.g., [3], [4], [5] and [6]) rely on change specifications, to automatically adapt components upon upgrade. For instance, our approach presented in [7] uses the history of structural changes (*refactorings*)[1] to automatically preserve *binary* framework backward-compatibility — existing plugins link and run with a new framework version without recompiling [10]. Upon the release of a new

[1]Refactoring plays an indispensable role in framework evolution [8] and, according to [9], is the cause of more then 80% of application-breaking changes.

framework version we create adaptation layers that are placed between the framework and plugins (Figure 1). The adapters [11] bridge the mismatches of component interfaces caused by framework refactorings and are automatically derived based on the semantics of the latter. As we aim for binary backward-compatibility, our adaptation is performed on the binary level, which a priori enables more powerful adaptation strategies (e.g., by directly specifying method receivers, binary versions and linker-specific instructions).

We stipulate, that although specification-based component adaptation is a powerful means to support backward-compatibility, it alone is not enough to guarantee a sound software component evolution. Due to various particularities of used object-oriented programming languages and different possible implementation strategies, the developers additionally need a carefully elaborated guideline on how to define and change the framework, while still permitting effective and efficient component adaptation. The main contribution of this paper is to identify and make the developers aware of several important problems and their corresponding solutions characteristic to the upgrade of object-oriented components, in particular frameworks. The paper is an important step towards one of our main research goals — the elaboration of an *adaptation-aware guide* that will support developers in defining and evolving backward-compatible frameworks. We argue furthermore, that although the following discussion was driven by problems identified in our refactoring-based adaptation, it is also relevant for most of the existing adaptation approaches.

The following Sect. 2 sets the context of our discussion and presents the main contribution of the paper — a list of problems and solutions important for object-oriented framework adaptation. We overview related work in Sect. 3, discuss additional issues in Sect. 4 and conclude in Sect. 5.

2 Controlled Adaptation-Oriented Framework Evolution

In this section we demarcate issues relevant for component upgrade (focusing on white- and black-box .NET and Java frameworks with dynamic binding, method overriding and callback types), describe associated problems and suggest appropriate solutions. They were discovered and developed in the course of a collaboration project with our industrial partner Comarch [12].

2.1 Controlled Change

In our refactoring-based approach most of the refactorings are adapted automatically; however, several cases must be considered additionally:

2.1.1 Available Functionality

Although most of the refactorings are adaptable, there are at least two, namely Remove Method and Remove Class, which are not and must therefore be prohibited. The main reason is that the pure deletion of component functionality, the use of which by clients cannot be checked, as in the case of a framework, should be considered a maintenance error.

Solution. Apply only refactorings which semantics prevent information loss. For example, the two refactorings mentioned can be used in composite refactorings (e.g., the use of Remove Method in Push Down Method), because the semantics of the latter permits adaptation without loosing framework functionality.

2.1.2 Verified Semantics

If the framework developer intentionally changes the framework's behavior, a decision has to be made, whether the change should be visible to the existing plugins. For example, if the default tax rate used in the framework has changed, it is usually desirable to reveal it to the clients. In contrast, if the order of framework events, on which the application functionality may rely, has changed, the application can break.

Solution. In the first case the developer who changes a method's visible behavior must also reflect the change in a test, to check that the behavior changed as intended. Whereas such test is developed against the new framework functionality, it must run on and verify adapters. In the second case the application-breaking changes need to be compensated for using appropriate adaptation techniques (e.g., protocol adaptation).

2.1.3 New Abstract API Methods

A sign of framework maturing is the appearance of new abstract methods in its API types (abstract classes and interfaces). Because an abstract method represents a part of the required contract [13], which its implementee must fulfill, the introduction of new abstract methods makes the required contract stronger, thus breaking existing clients.

Solution. A solution is to specify a default implementation of the new method. If it is in an abstract class, the developer should declare it as protected, indicating that it can be overridden. If it is in an interface, the implementation must be provided separately and the adapter tool instructed correspondingly. The same approach applies to more general changes from concrete to abstract API types.

A different solution would be to follow the "2" convention [14] as it was done, for instance, with the Java interface `java.awt.LayoutManager`: in order not to change the contract between the framework and

clients, a new interface or abstract class extending the old one is introduced and the use of the older one is discouraged through deprecation. Furthermore, the name of the new artifact equals the name of the old interface or abstract class extended by a "2" (as in `LayoutManager2`).

2.1.4 Modification of Generic Types

In case of changes involving generic types, two main scenarios have to be considered. First, a non-generic type can be transformed into a generic one [15]. Old plugins are then not aware of how to use the new generic type lacking appropriate information about its type parameters. Second, a generic type can be changed into a non-generic one, thereby possibly narrowing its use context. For example, if the signature of a method has changed from `T method()` to `String method()`, the call will fail in case `T` was instantiated with `Integer`.

Solution. The first case represents a type widening conversion and, hence, can be adapted in general. The main difference from the adaptation of ordinary types is that the adapter needs to provide a default type parameter in order to instantiate its adaptee (i.e., the generic type). For that, it is usually sufficient to pass the upper bound of the type parameter (which is the root of the language's inheritance tree, e.g., `Object`, in case of implicit upper bound). The second case must be prohibited in general, unless there is a way to ensure the safeness of type narrowing conversion. For example, it is safe to change the signature of a method `T method()`, where `T` has the upper bound `Customer`, to `Customer method()`.

2.2 Controlled Design

The adapter design pattern relies on the method forwarding technique to wrap the functionality of an adaptee into an adapter. This technique may be rendered insufficient by the way the adaptee type is defined (too much of its information is revealed) or used (too strong assumptions about its structure are made).

2.2.1 Encapsulated Fields

In accordance to common object-oriented principles, classes' fields must not be publicly accessible. This rule becomes especially important in case of method forwarding used, for instance, in our adaptation approach — if not respected, changes applied to the fields in a new component version are directly visible to the clients and cannot be shielded by an adapter.

Solution. Fields ought to be encapsulated properly by declaring them private and providing accessor methods. In this case field adaptation is entailed by the adaptation of the corresponding accessor methods.

2.2.2 Careful Reflection

Reflection is a technique for working with the meta level of an object-oriented system. This implies that the structure of objects, that is, of classes, fields, and methods, is laid open to the user of reflection. Often, having access to that meta level can considerably ease implementing certain functionality. However, in many cases this introduces a dependency on certain properties of an object's class thus breaking encapsulation. For example, a plugin might read raw string data and use them to reflectively create an instance of a type not known at compile time. If this is not done carefully (e.g., by just assuming the type has only one constructor), a runtime error might arise, because the type adapter adds an additional constructor dedicated for wrapping objects, invalidating thus the client assumption. This scenario is depicted in Fig. 2.

Solution. A proper use of reflection entails the complete verification of obtained references to meta-level structures, that is, a program has to check, whether all parameter types (and not just the name) of a method or constructor match the expected ones before an invocation.

2.2.3 Custom Serialization

Assume, an instance of a framework class is serialized; in a new framework version, the class is changed and, hence, adapted. The default serialization of neither .NET (provided by the `System.Runtime.Serialization` namespace) nor Java suffices to restore the serialized instance, due to the structural mismatch of the adapter that contains adaptation-specific members (e.g., the delegation field) and of the original class.

Solution. Whenever serialization is needed, a custom serialization process has to be provided to properly initialize the delegation field. For example, in .NET this can be achieved by implementing the interface `ISerializable`, while in Java the two additional methods `readObject` and `writeObject` have to be provided.

2.3 Controlled Adaptation

Any adaptation technology has certain limitations, according to which adaptation decisions have to be made. In other words, the adaptation itself must also be controlled and the framework developer must be made aware of the adaptation decisions that may influence eventual framework evolution.

2.3.1 Controlled Object Schizophrenia

Delegation as a re-use mechanism introduces *object schizophrenia* [16] due to the dichotomy of the adapter

Framework type

```
public class Customer {
  private String name;

  public Customer(String name) {
    this.name = name;
    //initialize other fields
  }

  //other methods
  //             and fields
}
```

Adapter

```
public class Customer {
  //framework type for delegation
  private _Customer d;

  public Customer(_Customer d) {
    this.d = d;
  } //wrapping constructor

  public Customer(String name) {
    d = new _Customer(name);
  } //initializing constructor
  //delegation methods
}
```

Plugin

```
public class MyPlugin {
  public void DoSomething() {
    String value;    //read in value
    Type target;    //find target type
    object o = target.GetConstructors()[0].Invoke(value);
  }
}
```

Figure 2. C# code of a dangerous use of reflection. The upper part shows the original framework type and the generated adapter. The client code in the lower part will invoke the wrong constructor when run on the adapter.

and adaptee objects: what is intended to appear as a single object is actually broken up into two or more, each 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 our case, if not handled properly, adapters may obtain their own object identity different from the corresponding framework objects (i.e., adaptees). As a consequence, the applications relying on object identity may malfunction.

Solution. Our solution makes use of the specific implementation of the object identity concept in Java and .NET, where it is exposed by predefined methods. In our adaptation technique the adapters also forward these methods to the adaptees thereby preserving object identity. However, comparison using object references instead of these methods may invalidate our approach and should be prohibited.

2.3.2 Adapting User-defined Types

In any object-oriented framework there are API methods that accept and/or return user-defined types — types other then those built into the framework implementation language. Because these types can also be apt for change and, hence, adaptation, the wrapping/unwrapping strategy is needed.

Solution. Whenever such a type is passed by a plugin as an argument to an adapter method call, it has to be unwrapped (to the corresponding framework type) before the adapter will delegate to the appropriate framework method. Analogously, when a method call on the delegatee returns a (framework) type, it has to be wrapped to the corresponding adapter type before returning it to the plugin. Whereas wrapping can be done by using the adapter constructor, unwrapping needs a dedicated adapter method returning the delegatee. This method has to be embedded into the adaptation strategy and protected from being overridden by a plugin.

Adapter Method

```
public string FormatData(object data, string format) {
 try {
   return delegatee.FormatData(data, format);
 } catch (Exception ex) { //catch every exception
   if (isFWException(ex)) { //check for custom exceptions
     throw(adapt(ex)); //wrap FW exception in adapter
   } else {
     throw(ex); //just rethrow ex
   }
 }
}
```

Figure 3. Adapting exception handling.

2.3.3 Exceptions

There are three issues to be considered with regard to exception adaptation. First, the exception classes may change in the same way as ordinary types (e.g., renaming of exception classes and methods). Second, the exception objects also need to be adapted at run-time, like the instances of user-defined types. Third, the influence on a particular adaptation strategy is driven by the specific implementation language. For instance, in Java exceptions may belong to the signature of the

methods (*checked* exceptions), whereas in .NET they never do.

Solution. Adaptation of exception types is implemented similarly to adapting user-defined types as described in the previous subsection. That is, for the changed exception classes adapter classes are created, the instances of which are used at run-time for the wrapping and unwrapping of exception objects (Fig. 3). Moreover, the adaptation strategy must be aware of language particularities with regard to exceptions and handle them appropriately. For example, if there are checked exceptions, they have to be treated as a part of the method signature.

2.3.4 Runtime Type Checks

When generating the adapter for a generic type, the generator has no information about which type will actually be used upon adapter instantiation. The adapter itself has to find out at run-time, whether it was passed a custom (and, hence, adapted) type or a built-in type. In the former case, the adapter has to unwrap the type before sending it to the framework, while in the latter the type is forwarded as it is.

Solution. To perform this runtime check, the adapter needs a common criterion to decide, whether a parameter type is an adapter itself. This can be done with the help of meta-data (e.g., custom attributes in .NET or annotations in Java), which can be created and used to mark the adapters automatically by the adapter generator.

3 Related Work

Usually the way developers should change a component is defined either implicitly (e.g., implied by the programming practice in general, or by some shared knowledge in a certain developer community in the form of "do not do that, unless you do it in that way") or described in the form of best practice tutorials and handbooks.

Basing on Design By Contract paradigm [13], des Rivières showed how changes of package, type, method and variable contracts can potentially break a client [14]. A method contract is an agreement between the method provider (e.g., a framework) and a method client (e.g., a plugin).[2] It consists usually of a precondition (what a method requires from its clients), a postcondition (what a method guarantees to its clients), and an invariant (what is guaranteed not to change during the method execution). If the precondition of a framework method becomes stronger in a new release, it may break existing clients (intuitively, now there is more required from the client than before, and the client may not meet the new requirements). Weakening the postcondition of a method may also break the client — the client now gets less than before, and some of its previous assumptions may become invalid. Des Rivières classified changes applied to Java components according to their impact on the existing clients and for some client-breaking changes suggested appropriate solutions — workarounds [14].

In [17], Mikhajlov and Sekerinski investigated the well-known Fragile Base Class Problem (FBCP), by which seemingly safe modifications to a base API class in a white-box system may cause the derived classes to malfunction. They formally defined the conditions, under which any change to a base class may be considered safe and prescribe a set of requirements (i.e., a guide) of how a base class has to be defined and used by the programmers to avoid the FBCP.

[2]The use of callbacks switches exactly the contract parties.

4 Further Discussion

The decision of which changes can be applied to a framework is also influenced by the particularities of the applied adaptation tool and technology, by the availability and efficiency of testing, and by non-functional requirements to the final applications.

4.1 Adaptation Recall

The range of permissible changes to the API directly depends on the adaptation recall — the ratio of changes supported by a particular adaptation tool. The higher the recall value is, the more changes are (automatically) adaptable and the less restrictions on the component evolution are implied. The developer can then estimate the impact of other, non-adaptable changes and either exclude them from the list of allowed modifications or apply them in a controlled manner. An example of the latter is how we combine behavior-changing modifications (controlled by tests) with the automated adaptation of refactorings (see Sect. 2.1.2 "Verified Semantics").

4.2 Adaptation Verification

The soundness of generated adapters needs to be verified either statically by a program verifier, at run-time by a test suite, or by their combination. The concrete decision is itself driven by the adaptation technology. For example, in our approach we rely on the work of [18], which uses the semantics of refactorings to automatically test the refactored programs.

4.3 Non-Functional Requirements

Non-functional requirements, especially performance issues, may affect the applicability of certain modifications. For instance, if the adaptation of a change

implies performance overhead unacceptable for a real-time application, this change is not permitted. In such cases, the developer decisions are driven by application-specific non-functional requirements and results of the adapter benchmarking.

5 Conclusion

Although a plethora of adaptation technologies to compensate for component upgrade exists, the adaptation alone does not suffice to guarantee backward-compatibility of evolving components. In addition, the developers must be guided in the way they modify components to avoid potential incompatibilities that cannot be coped with by adaptation. In this paper we identified a set of such problems typical for the evolution of object-oriented components and suggested, where appropriate, corresponding solutions. The final goal is to elaborate an adaptation-aware developer guide — a set of instructions of how to evolve and maintain a software component, while preserving its backward-compatibility.

Acknowledgements

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

References

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

[2] F. P. Brooks Jr., The mythical man-month: After 20 years, *IEEE Software*, 1995, *12*(5), 57–60.

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

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

[5] R. Keller & U. Hölzle, Binary component adaptation, *ECOOP'98: European Conference on Object-Oriented Programming*, vol. 1445 of *Lecture Notes in Computer Science*, Springer, 1998, 307–329.

[6] S. Roock & A. Havenstein, Refactoring tags for automatic refactoring of framework dependent applications, *XP'02: Proceedings of Extreme Programming Conference*, 2002, 182–185.

[7] I. Şavga & M. Rudolf, Refactoring-based adaptation for binary compatiblity in evolving frameworks, *GPCE'07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, Salzburg, Austria: ACM, 2007, 175–184.

[8] L. Tokuda & D. Batory, Evolving object-oriented designs with refactorings, *Automated Software Engineering*, vol. 8, Hingham, MA, USA: Kluwer Academic Publishers, 2001, 89–120.

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

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

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

[12] Comarch homepage, `http://www.comarch.com`.

[13] B. Meyer, Applying "design by contract", *Computer*, 1992, *25*(10), 40–51.

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

[15] A. Kieżun, M. D. Ernst, F. Tip, & R. M. Fuhrer, Refactoring for parameterizing Java classes, *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007.

[16] K. C. Sekharaiah & D. J. Ram, Object schizophrenia problem in object role system design, *OOIS '02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, London, UK: Springer, 2002, 494–506.

[17] L. Mikhajlov & E. Sekerinski, A study of the fragile base class problem, *Lecture Notes in Computer Science*, 1998, *1445*, 355–373.

[18] F. Bannwart & P. Müller, Changing programs correctly: Refactoring with specifications., J. Misra, T. Nipkow, & E. Sekerinski (eds.), *FM*, vol. 4085 of *Lecture Notes in Computer Science*, Springer, 2006, 492–507.