Automatic Refactoring-Based Adaptation

Ilie Şavga and Michael Rudolf

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany, {is13|s0600108}@inf.tu-dresden.de

Abstract. Pure structural changes (*refactorings*) of a software component, such as a framework, may introduce component mismatches between it and components that used one of its previous versions. To preserve existing applications, we propose to use the information about refactoring to automatically adapt such mismatches.

1 Introduction

The components comprising a component-based application may have strong inter-dependencies by making assumptions about the provided interfaces of the components they cooperate with. In case one of these components evolves and is upgraded to a new version, the changes applied to it may change its interface. This, in turn, may lead to a *component mismatch* (components do not cooperate as intended) that breaks the existing application. To preserve the latter, the developer must perform *component adaptation* — a set of steps to detect and bridge component mismatches. To reduce the costs and improve the quality of adaptation process, it is crucial to, at least partially, automate it. In many cases, though, it is not possible, because the component specification is usually limited to the Application Programming Interface (API), which is too weak to enable automatic mismatch detection and adaptation.

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 a means of modules containing custom code (*plugins*) [15]. 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. Either plugin developers are forced to manually adapt their plugins or framework maintainers need to write update patches. Both tasks are usually expensive and error-prone.

We argue that most of the changes causing *signature* component mismatch (e.g., mismatch of method, parameter and type names, of method and parameter types, of parameter order) can be automatically detected and resolved by using the information about the code change. More specifically, we are focusing on *refactorings*—behavior-preserving source transformations [19][1] — that are the

[1] In accordance with [19], [12], [20] we consider the addition of functionality behavior-preserving.

major cause of signature component mismatch comprising more than 80% of problem-causing changes [9]. The intuition is that a refactoring operator can be treated as a formal specification of a syntactic change and the information about the component's refactoring can be used to automate the adaptation.

Figure 1 shows, how we use the refactoring history to create adapters [13] between the framework and plugins upon the release of a new framework version. The adapters then shield the plugins by representing the public types of the old version, while delegating to the new version. Adapter generation is not limited to two consecutive versions; they can be generated for any previous API version.

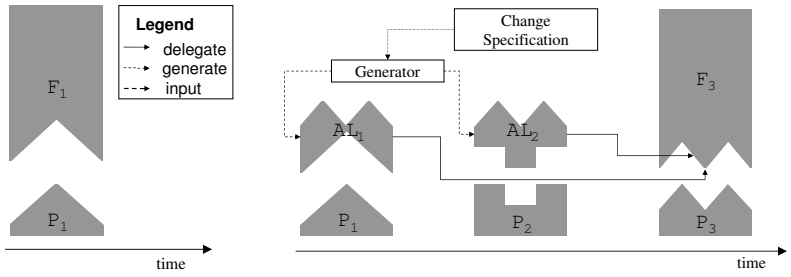

Fig. 1. Plugin adaptation in an evolving framework. In the left part, the framework version F_1 is deployed to the user, who creates a plugin P_1. Later, two new framework versions are released, with F_3 as the latest one. While new plugins (P_3) are developed against the latest version, the existing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

Note the non-intrusive way of adaptation, that is, neither plugins nor the framework are invaded by the adaptation code. Moreover, because we are generating binary adapters, the existing plugins remain *binary compatible* — they link and run with a new framework release without recompiling [11].

The rest of the paper is organized as following: we sketch our approach of using refactoring information to automate adaptation of signature mismatches in Sect. 2, overview related work in Sect. 3, discuss open issues in Sect. 4 and conclude with our main statement in Sect. 5.

2 Refactoring-Based Adaptation

Although refactoring preserves behavior, it changes syntactic component representation, on which other components may rely. Figure 2 shows the application of the *ExtractSubclass* refactoring to a framework class modeling customers. If an existing plugin calling the method *getDiscount()* on an instance of *Customer* is not available for update, it will fail to both recompile and link with the new framework version due to the introduced signature mismatch.

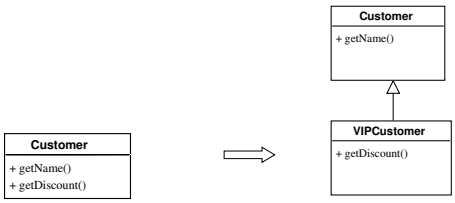

Fig. 2. *ExtractSubclass* refactoring. The method *getDiscount* is moved from *Customer* to its new subclass *VIPCustomer*.

Comebacks. To cope with the signature mismatches introduced by refactoring operators, for each of the latter we formally define a *comeback*—a behavior-preserving transformation that defines how a compensating adapter is constructed. That is, we define an adaptation-oriented pattern problem/solution library of transformations, where a problem pattern is the occurence of a framework refactoring and its solution is the corresponding comeback (adapter refactoring).[2] Note, that a comeback differs from a refactoring *inverse* in that a comeback is applied to adapters, and not to framework types. It is also different from refactoring *undo* - the latter is an un-execution of a (successfully applied) refactoring operation.

Technically, a comeback is realized in terms of refactoring operators to be executed on adapters. For some refactorings, the corresponding comebacks are simple and implimented by a single refactoring. For example, to the refactoring *RenameClass (name, newName)* corresponds a comeback consisting of a refactoring *RenameClass (newName,name)* that renames the adapter to the old name. For other refactorings, their comebacks consist of sequences of refactorings. For instance, the comeback of *MoveMethod* is defined by *DeleteMethod* and *AddMethod* refactoring operators, which sequential execution effectively move the method between the adapters.

For an ordered set of refactorings that occured between two framework versions, the execution of the corresponding comebacks in the reverse order yields the adaptation layer. Figure 3 shows the workflow of refactoring-based plugin adaptation. First, we create the adaptation layer AL_n (the right part of the figure). For each public class of the latest framework version F_n we provide an adapter with exactly the same name and set of method signatures. An adapter delegates to its public class, which becomes the adapter's delegatee. Once the adapters are created, the actual adaptation is performed by executing comebacks backwards with respect to the recorded framework refactorings, where a comeback is derived using the description of the corresponding refactoring. When all comebacks for the refactorings recorded between the last F_n and a previous F_{n-x} framework version are executed, the adaptation layer AL_{n-x} reflects the old functionality, while delegating to the new framework version. Because

[2] A detailed description of our approach including formal comeback definition and current results is presented in [22].

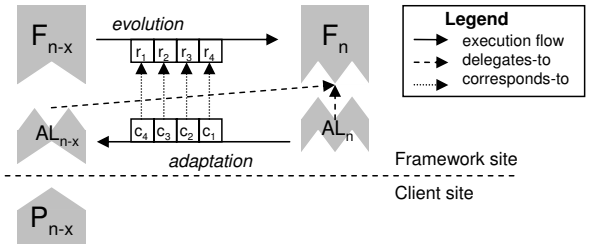

Fig. 3. Adaptation workflow. To a set of refactorings (r_1-r_4) between two framework versions (V_{n-x}, $V_n, n > x > 0$) correspond comebacks (c_4-c_1). Comebacks are executed on the adaptation layer AL_n backwards to the framework refactorings. The resulting adaptation layer AL_{n-x} delegates to the new framework, while adapting plugins of version P_{n-x}.

the adaptation is performed at the framework site, it is transparent for plugins. Neither manual adaptation nor recompilation of plugins is required.

Tool validation. We are evaluating our concept in an operational environment using a logic programming engine. For several refactorings we specified the corresponding comeback transformations as Prolog rules and developed a parser for the CIL code (as stored in .NET assemblies) in order to extract meta-information about API types. This meta-information is used to create a fact base, on which a Prolog engine then executes the required comebacks. Once all comebacks are executed, the fact base contains the necessary information to create adapters and is serialized back to assemblies.

For instance, for the *ExtractSubclass* refactoring introduced earlier in this section, the execution by our tool of the *CbExtractSubclass* comeback yields the corresponding binary adapter, the C# source code of which is provided in Listing 1. The delegation field *delegateeCustomer* is initialized with an instance of *VIPCustomer* and is used to forward the method calls *getName* and *getDiscount*.

Listing 1. C# code of the generated adapter

```
1  public class Customer {
2    protected VIPCustomer delegateeCustomer;
3
4    public Customer() {
5      delegateeCustomer = new VIPCustomer(); }
6
7    public string getName() {
8      return delegateeCustomer.getName(); }
9
10   public float getDiscount() {
11     return delegateeCustomer.getDiscount(); }
12 }
```

The following refactorings are currently supported by our tool: *RenameMethod*, *RenameClass*, *AddMethod*, *AddClass*, *MoveMethod*, *PullUpMethod*, *PushDownMethod*, *ExtractSuperclass*, *ExtractSubclass*, *ExtractClass*. Because we assume, that all API fields are encapsulated (accessed by get/set methods), which is a general requirement in our project, support for field refactorings is implied. The addition of other supported refactorings is discussed in Sect. 4.

Terminology note. According to Becker et al. [4], our approach is a design-time signature adaptation consisting of the following adaptation steps:

1. Detect mismatches. The semantics of refactoring is used to detect (or, more precisely, predict) signature mismatches upon component upgrade.
2. Select adaptation measures. To bridge syntactic component mismatches introduced by refactoring, we rely on the well-known Adapter design pattern [13] that was shown to be "very flexible as theoretically every interface can be transformed into every other interface." [4] More specifically, we use the delegation variant of the pattern to support components written in languages not supporting multiple class inheritance .
3. Configure selected measures. The library of pattern problem/solution is used, where the information about a detected problem (refactoring) is used to configure, or instantiate, the appropriate solution (comeback).
4. Predict the impact. The impact on the functionality is implied by the definition of the comebacks: because a comeback is a refactoring, its execution will not change the behavior, whereas adapting the signature mismatch. The impact on the non-functional properties (e.g. on performance), need to be evaluated for each comeback separately and then for the possible adaptation as a whole.
5. Implement and test the solution. We systematically construct adapters using predefined comeback library. Although the soundness of comebacks is proved, the soundness of the generated adapters needs to be tested, too, to avoid implementation bugs. We are developing a refactoring-driven testing that aims for test generation based on the refactoring history. We will also elaborate the benchmark strategy to estimate the performance pentalties implied by delegation.

3 Related Work

To analyze the nature of the application-breaking changes, Dig and Johnson [9] investigated the evolution of four big frameworks and discovered that most (from 81% up to 97%) of such changes were refactorings. The reason why pure structural transformations break clients is the difference between how a framework is refactored and how it is used. For refactoring it is generally assumed, that the whole source code is accessible and modifiable (the *closed world* assumption [9]). However, the frameworks are used by plugins not available at the time of refactoring. As a consequence, plugins are not updated correspondingly.

The existing approaches overcoming the closed world assumption in case of component evolution can be divided into two groups. Approaches of the first

group rely on the use of a kind of middleware (e.g., [1], [2], [6], [18]) or, at least, a specific communication protocol ([11], [17]) that connect a framework with its plugins. This, in turn, implies a middleware-dependent framework development, that is, the framework and its plugins must use an interface definition language and data types of the middleware and obey its communication protocols.

The second group consists of approaches to distribute changes (including refactorings) and to make them available for the clients remotely. The component developer has to manually describe component changes either as different annotations within the component's source code ([3], [7], [21]) or in a separate specification ([16], [23]). Moreover, the developer must also provide adaptation rules, which are then used by a transformation engine to adapt the old application code. Writing annotations is cumbersome and error-prone. To alleviate this task, the "catch-and-replay" approach [14] records the refactorings applied in an IDE as a log file and delivers it to the application developer, who "replays" refactorings on the application code. Still, current tools do not handle cases when a refactoring cannot be played back in the application context. For example, they will report a failure, if the renaming of a component's method introduces a name conflict with some application-defined method. In addition, the intrusive way of adaptation requires the availability of the application's sources.

4 Discussion

Regarding our technology, several important issues are still open.

Approach generalization. Although we focus on mismatches caused by component upgrade, the refactoring-based adaptation is in general applicable to any occurence of a syntactic mismatch. For example, all adaptable signature mismatches presented in [4] can be described as a result of refactoring. In case of component integration, to automatically adapt such mismatches one could describe them by corresponding refactorings and re-use the comeback library.

Applicability demarcation. We intentionally prohibit some refactorings in our approach. Particularly, we do not allow for *RemoveMethod* and, hence, *RemoveClass* to be applied as standalone refactorings. The main reason is that in our project we consider the pure deletion of functionality that may still be in use by old clients, as a sign of maintenance error and prohibit them (because in fact they cannot be then considered refactorings). In other words, pure deletion that is refactoring under the closed world assumptions cannot be considered as such in a open world. Still, we allow such refactorings to be used in composite refactorings (e.g., *PushDownMethod*), because the semantics of the latter permits adaptation by preventing the information loss.

Limitations. Currently, only common class and method refactorings are supported (listed in Sect. 2). One reason is implied by the state-of-the-art of the refactoring research: some refactorings (e.g., *ExtractInterface*) cannot be specified using existing formalisms as these refactorings imply multiple inheritance and the notion of interface. Moreover, there is no existing work for a certain class of refactorings, namely those applied to generic types. In addition, for

some refactorings (e.g., those splitting/merging types), additional assumptions under which the adaptation becomes possible need to be defined.

Going beyond refactorings. Besides information about the public types and the refactoring history, our approach needs no additional component specifications. However, the latter are required to support modifications that go beyond refactoring (e.g., protocol changes). To broden the range of supported changes, one needs to investigate how to combine other adaptation techniques with the refactoring-based approach.

Adaptation time variation. Although the adapters themselves are generated statically, nothing prevents us from shifting the actual adaptation from design-time to the load- and run-time. In particular, one of our students investigated the use of aspect-oriented techniques for adaptation, where the execution of comebacks ends in an adaptation aspect [5].

Code generation. We deliberately left out the description of our actual code generation. Still, there is a plenty of open issues, such as treating object schizophrenia (implied by delegation), wrapping/unwrapping of API user-defined types, and treatment of reflection calls.

Refactoring history acquisition and representation. As our approach requires the information about refactorings, an importat issue is how it can be obtained (e.g., by recording the applied refactorings in IDE [14], or detecting them in the source code [24], [8]) and stored (e.g., in a refactoring-aware configuration management system [10]).

Verification. As mentioned in Section 2, we need to verify both the adapter soundness and the performance implied by adaptation. Because in general the plugins may not be available for running tests, static verification could be a better choice comparing to run-time testing.

5 Summary

Our technology to retain binary compatibility of existing plugins implies adaptation and requires the information about refactorings occured between framework releases. For each refactoring there is a corresponding comeback that describes how the adapters should be constructed. Execution of comebacks backwards with regard to the framework refactorings yields the corresponding adapter layer. The adaptation is performed at the framework site and is transparent for the clients. We conclude with our main statement:

Treated as a formal specification of syntactic changes, refactoring can foster the automated adaptation of signature component mismatches thus considerably reducing the costs and increasing the quality of component adaptation.

References

1. CORBA homepage. http://www.corba.org.
2. Microsoft COM homepage. http://www.microsoft.com/Com/default.mspx.

3. I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.
4. S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 193–215. Springer, 2004.
5. C. Bitter. Preserving binary backward-compatibility using aspect-oriented techniques. 2007. Master Thesis.
6. J. Camara, C. Canal, J. Cubo, and J. Murillo. An aspect-oriented adaptation framework for dynamic component evolution. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 59–71, 2006.
7. K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.
8. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.
9. D. Dig and R. 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.
10. D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE*, 2007.
11. I. R. Forman, M. H. Conner, S. H. Danforth, and L. 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.
12. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.
14. J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.
15. R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.
16. R. Keller and U. Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–329, 1998.
17. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. 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-Verlag.
18. F. McGurren and D. Conroy. X-adapt: An architecture for dynamic systems. In *Workshop on Component-Oriented Programming, ECOOP, Malaga, Spain*, pages 56–70, 2002.
19. W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Urbana-Champaign, IL, USA, 1992.

20. D. B. Roberts. *Practical analysis for refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1999.
21. S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02: Proceedings of Extreme Programming Conference*, pages 182–185, 2002.
22. I. Savga and M. Rudolf. Refactoring-based adaptation for binary compatiblity in evolving frameworks. In *Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, Salzburg, Austria, October 2007. To appear.
23. I. Savga, M. Rudolf, J. Sliwerski, J. Lehmann, and H. Wendel. API changes - how far would you go? In R. L. Krikhaar, C. Verhoef, and G. A. D. Lucca, editors, *CSMR*, pages 329–330. IEEE Computer Society, 2007.
24. P. Weissgerber and S. 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.