Migration of Applications across Object-Oriented APIs

by

Thiago Tonelli Bartolomei

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

Abstract

Software developers often encapsulate reusable code as *Application Programming Interfaces* (APIs). The co-evolution of applications and APIs may motivate an *API migration*: the replacement of application dependencies to an *original API* by dependencies to an alternative API that provides similar functionality and abstractions.

In this dissertation, we investigate issues associated with API migration in object-oriented systems, with special focus on wrapping approaches. We present two studies and a set of developer interviews that elicit issues in the process and techniques used in API migration in practice. The results suggest that the most pressing issues relate to discovery and specification of differences between APIs, and to assessment of migration correctness. This dissertation introduces techniques and a method to address these issues.

We propose the use of design patterns to support the specification of API wrappers. *API wrapping design patterns* encode solutions to common wrapping design problems. We present an initial catalog of such patterns that were abstracted from programming idioms found in existing API wrappers.

We introduce the concept of *compliance testing for API migration*, a form of automated testing. Compliance testing supports the discovery of behavioral differences between a wrapper and its corresponding original API, as well as assessment of wrapper correctness. Compliance testing uses API contracts and assertion tunings to explicitly capture and enforce the notion of a "good enough" wrapper that is informal in practice.

We present the *Koloo method for wrapper-based API migration.* The method prescribes practical steps to use compliance testing as a means to elicit the requirements for the API migration, and to assess its correctness. *Koloo* fits within the iterative, sample-driven general API migration process usually followed by developers in practice.

We evaluate the *Koloo* method in an empirical study. The subjects cover the domains of XML processing, GUI programming and bytecode engineering. The results provide evidence that *Koloo* is superior to alternative methods in driving the development of a wrapper that is tailored for the application under migration. The results also show that API contracts help driving the evolution of the wrapper, and assertion tuning is necessary to relax the semantics of strict equality contracts, and useful to compromise on features that are difficult to emulate perfectly. Finally, we validate that the proposed design patterns are used in practical wrappers.

Acknowledgments

This project involved over half a decade of almost exclusive dedication. I am sincerely and immensely grateful to Fran for her support, love and patience throughout this time.

I am also very lucky to have been guided by two great mentors. I thank my advisor, Prof. Krzysztof Czarnecki, for his constant encouragement and commitment. His energy, work ethics, focus on details, and competence definitely drove me to improve the quality of my work and to always aim for the best. I am grateful to Prof. Ralf Lämmel for his friendship, his hands-on research approach, almost infinite interest in my research topic, and enormous endurance during research sprints. My visits to Koblenz were surely highlights of my Ph.D. experience.

I would like to thank my external examiner, Prof. Giuliano Antoniol, for finding time to visit Waterloo during the summer, and for his insightful comments on this dissertation. I am very grateful to my internal thesis committee, Prof. Michael Godfrey, Prof. Sebastian Fischmeister, and Prof. Patrick Lam, for helping me shape this research.

I am extremely fortunate to have been raised in a great, supporting family. I cannot, particularly in such a short note, express how much I owe to my parents, Rogério and Sílvia, my brother Rodrigo, and my sister Lívia.

During the Ph.D., the Generative Software Development Lab was my second home. Literally. All you folks at the GSD Lab, I am very thankful for having had the opportunity to work with you. I will forever have fond memories of our passionate discussions about any possible subject. And of the Grad House. A special thanks to Ricardo Terra for proof-reading early versions of this dissertation.

I am also very happy that I was able to make great friends in Waterloo, particularly in the Brazilian community. From Saraus and barbeques to weekend cycling rides to Tuesday morning ski trips, I always enjoyed excellent company and friendship.

This work was partially funded by the IBM Centres for Advanced Studies, Toronto.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
COTS	Commercial Off-The-Shelf
DOM	Document Object Model
DSL	Domain-Specific Language
FSML	Framework-Specific Modeling Language
GoF	Gang of Four
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
MVC	Model-View-Controller
NCLOC	Non-Comment Lines of Code
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Chapter 1

Introduction

Software developers modularize source code to improve extensibility and reusability [60, 40]. In object-oriented systems, code can be modularized by several means, such as methods, classes, packages, components, libraries, and frameworks. These modules provide reusable code encapsulated as *Application Programming Interfaces* (APIs). In this dissertation, API refers to the programming elements exported by a module as well as the implementation details clients may depend upon. Consequently, the term API also denotes the resulting packaged module, usually a component, library, or framework. This focus on modularization fosters separation of concerns and reliance on third-party code. Domain experts can concentrate on the specialized tasks of their APIs, whereas application developers focus on the application logic. It is common for modern applications to rely on dozens of third-party APIs [48].

Useful software is constantly evolving [49]. The dependency of applications on third-party APIs means that significant maintainability issues may arise whenever any of these APIs evolve. New requirements or better designs drive API changes that often break existing dependent applications [26]. To capitalize on the API evolution, such applications need to undergo an *API upgrade*.

Yet, many reasons may motivate an *API migration*: the replacement of an API by an alternate API that provides similar functionality and abstractions. A motivation may be to replace an aged API with a novel API that provides better solutions, new features or improved performance. The development of an API may stagnate, making it prudent to migrate to an actively maintained API. An in-house or project-specific API may be replaced by a standard API. Applications may need to be integrated and their APIs consolidated. Finally, API migration may be motivated by license and copyright issues.

Principled solutions to API upgrade exist. Many API upgrade approaches [37, 91, 25, 70, 90, 44, 77, 78, 27] exploit the fact that most client-breaking API changes are *refactorings* [26]: simple code modifications that preserve behavior [32]. These approaches use comparison algorithms or recording tools to detect the refactorings applied to the API; then, clients are upgraded by replaying the refactorings or generating adaptation layers. Techniques using change rules [19] and semantic patches [67, 14] can also address localized changes more complex than refactorings. Finally, manual upgrade can be supported by recommending modifications based on how the API itself has adapted to its changes [21].

Unfortunately, API upgrade approaches cannot be directly applied to API migration because independently developed APIs may differ substantially. Our studies [10, 9], for example, have revealed that different feature sets must be accommodated, and similar features may be offered in different ways or diverge significantly in their details. Thus, refactorings and localized changes alone do not suffice in this context.

Previous work on API migration is limited to special cases, such as migrating from inheritance to annotation-based frameworks [83] and removing dependencies on legacy classes by migrating to similar replacement classes [8]. No previous work addresses API migration comprehensively.

This dissertation presents a comprehensive approach to API migration. We first studied the process and challenges of API migration in practice; we conducted two studies and a set of developer interviews. The results have suggested that the most pressing issues are related to discovery and specification of differences between APIs, and to assessment of migration correctness. This dissertation thus proposes a method called *Koloo* that addresses these issues and fits within the general process followed by developers in practice.

The remainder of this chapter presents an overview of *Koloo*—its assumptions, novel contributions, evaluation, and limitations—and outlines the structure of the dissertation.

1.1 Technical Approaches to API Migration

Wrapping and *rewriting* are the main technical approaches to API migration. Wrapping is a general re-engineering technique to provide access to existing functionality through a preferred interface. In object-oriented systems it amounts to the systematic use of the *ADAPTER* design pattern [34] to create an adaptation layer—a *wrapper*—between the APIs. That is, the interface of the *original API* is re-implemented using the *replacement API.* Rewriting is the substitution, within an application's source code, of references to the original API by corresponding features of the replacement API.

A hybrid approach is sometimes more appropriate. We have found in our interviews that limited application modifications can be expected when doing so significantly simplifies a wrapper implementation. In rewriting approaches, the *Facade* pattern [34] can be used to encapsulate adjustment code and simplify application code changes [8]. In this case, application source code that cannot be directly rewritten to a corresponding replacement API feature is instead redirected to an adjustment Facade.

In the context of API migration, wrapping has the following advantages over rewriting. First, no automation is necessary. Wrapping may benefit from tools that automate wrapper generation and additional development tasks, but automation is not as crucial as in rewriting. Rewriting approaches must be automated because code changes are scattered in application code, making manual rewriting a tedious, error prone, mostly repetitive task.

Second, wrapping does not modify application source code, except for the limited manual changes described above. In our interviews we have learned that developers may not trust automated rewriting tools because the resulting code may be unfamiliar and, therefore, difficult to maintain. Maintainability issues have also been identified in the context of rewriting for language migration [84], which often include API migration.

Finally, wrapping naturally enables incremental development, which is essential to discover API differences. It is trivial to construct or generate a wrapper with the original API interface but no implementation. Client applications can be compiled against this wrapper skeleton, which is then incrementally extended. An existing wrapper may also be the starting point to migrate additional applications. Automated rewriting specifications could also be reused across applications; however, developing the initial specification is more intricate because abstractions of original and replacement API may temporarily coexist in the application. Wrapping cleanly isolates original and replacement APIs.

Rewriting, on the other hand, truly removes original API abstractions from application source code. This may be necessary, for example, if the goal of the migration is to access new features of the replacement API directly, achieve better standardization in integration efforts, or to avoid license and copyright issues.

Nevertheless, in this dissertation we assume the use of wrapping. The development of a wrapper may be itself the goal of the API migration. If the goal is rewriting, then the wrapper can play the role of a migration specification. That is, wrapping can be used to incrementally discover and specify API differences, and subsequently guide manual or automated rewriting. For example, wrapper methods that simply delegate to the replacement API could be inlined in the client application; complex adjustments in wrapper methods could be encapsulated in a Facade. Thus, our research concentrates on the development and correctness assessment of API wrappers.

1.2 API Migration in Practice

This dissertation proposes the *Koloo* method to address the most pressing issues in API migration. These issues have been identified in two studies and a set of developer interviews.

We first designed an exploratory study to examine *behavioral mismatches* between APIs: differences in behavior occurring in seemingly corresponding features, such as methods with the same name and parameter types. Our goal was to understand the types of behavioral adjustments necessary to adapt the behavior of the replacement API to comply with the expected behavior of the original API, and observe how behavioral mismatches affect client applications. We chose to study XML APIs because they are structurally similar, which allows the focus on the low-level mismatches, such as differences in method contracts, that prevent simple delegation in the wrapper.

Next, we conducted a comprehensive artifact study to examine general wrapper design problems, and design problems caused by *structural mismatches* between APIs. We chose *Graphical User Interface* (GUI) APIs because they differ significantly in interfaces and inheritance hierarchies. We then studied the source code of two large, independently developed, open source GUI wrappers.

Finally, we conducted semi-structured interviews with seven developers who had experience in API migration. We asked general open questions about which development process, migration techniques and criteria for verification of results they used, and about existing and desired tool support. We also validated the findings of our second study with the main developer of one of the GUI wrappers.

We observed three pressing issues in API migration, and insights into their solutions.

The first issue is lack of support for the specification of differences between APIs. That is, there are no design guidelines to the development of wrappers. However, certain wrapper design problems and behavioral mismatches occur very frequently: we noticed some problems occur in multiple instances in a single wrapper and also across wrappers. The insight is that we can leverage programming idioms created by developers to solve these commonly occurring problems to guide wrapper development.

The second issue is how to discover the behavioral differences between APIs that are to be neutralized by a wrapper. Client applications use the original API in many ways and may depend on different implementation details. Thus, it is impractical for a wrapper to cover all and every case in advance. The insight from the studies is that a viable API migration targets a specific *application under migration.* Our interviews have confirmed that, in practice, API migration has usually a single target application.

The third issue is how to assess the correctness of the API migration. A formal approach to correctness is currently impractical because it involves the formalization of many artifacts: two APIs and one wrapper. A pragmatic approach is the use of testing to enforce behavioral equivalence between the original API and the wrapper in limited cases. However, we have learned from the studies that API differences may be too difficult to be neutralized completely. The insight is that, in practice, developers are often content with a wrapper that is "*good enough*". Behavioral equivalence can therefore often be relaxed. Nevertheless, this notion of a "good enough" wrapper is still too informal, and there is limited test automation to enforce it.

1.3 The ***Koloo*** Method for API Migration

The *Koloo* method for API migration addresses the issues discussed above. *Koloo* is based on the concepts of *API wrapping design patterns* and *compliance testing.*

Design patterns [34] are reusable solutions to commonly occurring problems in a particular context. *Koloo* guides developers in the specification of wrappers via a catalog of design patterns tailored to wrapper-based API migration. These *API wrapping design patterns* were abstracted from idioms created by developers of the studied wrappers.

We introduce *compliance testing* for API migration as a form of automated testing. Compliance testing supports the discovery of behavioral differences between original API and wrapper, and explicitly captures and enforces the notion of a "good enough" wrapper that is informal in practice. A compliance test captures the behavior of the original API as an *API interaction trace*: a runtime trace of the interaction between the API and the application under migration. The behavior of the original API embodied in the trace represents a behavioral specification to be verified in the wrapper. The trace is then re-executed against the wrapper while *API contracts*, which are oracles such as return value equality, enforce strict behavioral equivalence. But, since behavioral equivalence often needs to be relaxed, compliance tests accept *assertion tunings* that specify acceptable contract deviations for selected assertions.

We designed the *Koloo* method to evolve wrappers with respect to an application under migration. It can be used to create new wrappers or to evolve existing wrappers. The method prescribes compliance testing as a means to elicit the requirements for the API migration as well as assessment of its correctness; it prescribes design patterns to solve commonly occurring wrapping design problems. *Koloo* fits within the iterative, sample-driven general API migration process usually followed by developers.

A *Koloo* iteration starts with the design of an application scenario. The scenario defines the requirements for the wrapper in this iteration. Developers execute the scenario and capture an API interaction trace to be used as a compliance test. Then, developers execute the compliance test. Contract violations drive the evolution of the wrapper and the refinement of a specification of assertion tunings. The iteration ends when all contract violations are resolved—by development or tuning—and the scenario is asserted as migrated. Previously asserted scenarios can be re-executed to avoid wrapper regressions.

By the end of an iteration, the wrapper is demonstrably compliant with the original API in the exercised scenarios. The specification of contracts and assertion tunings precisely documents the unresolved differences between wrapper and original API. Developers can iterate over this process, creating new scenarios until they are satisfied with the coverage and quality of the wrapper.

1.4 Implementation and Evaluation

We have implemented the *Koloo* toolkit to automate several aspects of the method. The main tools are a tracer to collect API interaction traces, and an interpreter to re-execute traces and verify contracts. The toolkit provides tools to support additional tasks, such as manipulation of stored traces, and analysis of API source code to collect metrics and generate wrapper skeletons.

We performed an empirical evaluation of the method and a validation of the API wrapping design patterns.

The empirical evaluation compared the *Koloo* method against three alternatives. Our hypothesis was that *Koloo* is better than alternative methods in identifying the behavioral differences between the original API and a wrapper that are important to the application under migration. In particular, we hypothesized that *Koloo* is better than *i)* a test suite that targets the original API, because the suite may be too demanding or miss important cases; *ii)* an application test suite, because it may not contain assertions focused on the interaction with the API; and *iii)* the current industry practice, because *Koloo* may detect behavioral mismatches before they cause crashes or visible effects.

In this evaluation we used the XML wrapper we had created for our first study; its development had been driven by an original API test suite. We also used a GUI wrapper from the second study; it had been independently developed using the current industry practice. Then, we designed scenarios for applications of the original APIs and used *Koloo* to evolve the wrappers. Finally, we used *Koloo* to migrate an existing application across bytecode engineering APIs. Since we used application test cases as scenarios—to extract compliance tests—we were able to compare *Koloo* against a method driven by an application test suite.

Throughout the study we recorded contract violations and the use of tunings to relax contracts at selected assertions.

The results of the study provide evidence to support our hypothesis. *Koloo* was able to detect the behavioral differences that are important to the application under migration. Furthermore, we have found that the method is effective in driving the development of a compliant wrapper: violations of API contracts help drive the evolution of the wrapper, and assertion tunings are necessary to relax the semantics of strict equality contracts, and useful to compromise on features that are difficult to emulate perfectly.

The validation of API wrapping design patterns provides evidence of their use in practice. We designed source code metrics that capture instances of the design patterns. Then, we applied these metrics to the wrappers from which the patterns were abstracted, and to the wrappers we developed in our additional studies. The results show that the patterns are indeed extensively used in the studied practical wrappers.

1.5 Limitations

The main limitations of our research arise from limitations of wrapping and compliance testing. Object-oriented wrappers are effective to support the migration across traditional object-oriented APIs at a similar level of abstraction because they naturally encapsulate behavioral adjustments. However, the design of wrappers to migrate across APIs with different programming models, such as from inheritance to annotation-based frameworks or from XML parsers to XML object binding APIs, is often convoluted. Moreover, compliance testing focuses on functional equivalence between wrapper and original API; non-functional aspects are not immediately addressed.

The *Koloo* method focuses on driving the development process via API contract violations on selected scenarios. However, *Koloo* does not support developers in the design of good scenarios: we have not defined quality criteria for scenarios. Nevertheless, in our evaluation, scenario design was mostly simple, and existing application test cases could be leveraged. Generally, the cost of API migration is dominated by the actual wrapper refinement, which often requires a deep understanding of original and replacement APIs. General program comprehension techniques can assist developers in this task.

The catalog of API wrapping design patterns offered by *Koloo* is also limited. The design patterns have been abstracted from programming idioms detected in existing wrappers. The catalog is therefore limited by the choice of subject wrappers, and our ability to identify and abstract the idioms. As we gain more experience in wrapping we expect the catalog to eventually reach a established state with a core set of mature, reusable patterns.

Finally, our evaluation is limited. We have only evaluated *Koloo* in terms of its ability to drive the development of behaviorally compliant wrappers. However, we have not performed user experiments nor assessed the quality of the developed wrappers. Furthermore, we need more extensive studies to understand the generality and usefulness of the proposed API design patterns.

1.6 Research Contributions

The primary research contributions of this dissertation are the following:

- **API Wrapping Design Patterns:** We propose the use of design patterns to encode solutions to common design problems occurring in API wrappers. We present an initial catalog of API wrapping design patterns abstracted from programming idioms found in existing API wrappers.
- **Compliance Testing:** We introduce the concept of compliance testing for API migration, a form of automated testing that supports the discovery of behavioral differences between a wrapper and its corresponding original API. Compliance testing uses API contracts and assertion tunings to explicitly capture and enforce the notion of a "good enough" wrapper that is informal in practice.
- **Method for Wrapper-Based API Migration:** We propose the *Koloo* method for wrapper-based API migration. The method prescribes compliance testing as a means to elicit the requirements for the API migration as well as assessment of its correctness. *Koloo* fits within the iterative, sample-driven general API migration process usually followed by developers in practice.
- **Evaluation:** We evaluate the *Koloo* method in an empirical study. The subjects cover the domains of XML processing, GUI programming and bytecode engineering. The results provide evidence that *Koloo* is superior to alternative methods in driving the development of a wrapper that is tailored for the application under migration. The results also show that API contracts help driving the evolution of the wrapper, and assertion tuning is necessary to relax the semantics of strict equality contracts and useful to compromise on features that are difficult to emulate perfectly. Finally, we validate that the proposed design patterns are used in practical wrappers.

1.7 Outline of the Dissertation

In this chapter we have introduced API migration and presented an overview of our research. The remainder of this dissertation is organized as follows. Chapter 2 describes the two studies and the developer interviews we conducted to improve our understanding of API migration in practice. Chapter 3 presents the design patterns we abstracted from existing wrappers. Chapter 4 introduces a novel concept of compliance testing to assess correctness of API wrappers. Chapter 5 describes the *Koloo* method for API migration, which is grounded on compliance testing and design patterns. Chapter 6 presents evaluation of the method and validation of the proposed design patterns. Chapter 7 examines related work. Finally, Chapter 8 concludes with a discussion of the implications and limitations of our research, and future research directions.

1.8 Publications

This dissertation contains material from the following publications:

- Thiago Tonelli Bartolomei, Mahdi Derakhshanmanesh, Andreas Fuhr, Peter Koch, Mathias Konrath, Ralf Lämmel, and Heiko Winnebeck. Combining multiple dimensions of knowledge in API migration. In *Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM) at the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, March 2011

- Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and Back: Patterns for API Migration by Wrapping. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM)*, pages 1–10, September 2010

- Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an API Migration for Two XML APIs. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE)*, pages 42–61, October 2009

Chapter 2

API Migration in Practice

In this chapter, we identify the most pressing issues in API migration in practice. We first introduce the basics of API migration by wrapping with a simple example. Then, we present an exploratory study on behavioral mismatches between XML APIs. Next, we report on an artifact study on structural mismatches between GUI APIs. Finally, we describe a set of interviews we conducted with developers experienced in API migration.

2.1 Basics of API Wrapping

2.1.1 The Tailored *ADAPTER* Pattern

The *ADAPTER* design pattern is the classic solution for integrating a client class with types that have different interfaces than the client expects. The GoF [34] book describes two variants of adapters: *class* and *object*. In both cases, an adapter class extends the type expected by the client to intercept method calls and redirect them to the new target, called the adaptee. Application constructor calls must still be rewritten to the instantiate the adapter type instead of the original type. Class adapters integrate the adaptee through inheritance whereas object adapters use object composition.

In the context of API migration, adapters do not extend expected types; instead, adapters replace them. We use the term *surrogate* to denote such adapters. A surrogate re-implements an original API type, often using an adaptee of the replacement API, while preserving its interface. Thereby, constructor calls in client applications can also be retained. A *wrapper* is a collection of such surrogates and possibly internal helper types.

```
 1  public class Vector extends AbstractList implements ... {
 2      ArrayList adaptee;
 3      public Vector() {
 4          adaptee = new ArrayList();
 5      }
 6      public void add(Object o) {
 7          adaptee.add(o);
 8      }
 9      public void setSize(int ns) {
10          while (adaptee.size() < ns) adaptee.add(null);
11          while (adaptee.size() > ns) adaptee.remove(ns);
12      }
13      public Enumeration elements() {
14          return new EnumerationImpl(adaptee.iterator());
15      }
16      ...
17  }
18  public class EnumerationImpl implements Enumeration {
19      Iterator adaptee;
20      EnumerationImpl(Iterator i) {
21          this.adaptee = i;
22      }
23      public Object nextElement() {
24          return adaptee.next();
25      }
26      ...
27  }
```

Figure 2.1: Vector and Enumeration surrogates.

Surrogates may populate the same namespace as original API types. In this case, a client application is migrated by simply linking to the wrapper instead of linking to the original API. If the wrapper populates a different namespace, trivial namespace rewriting of the application is required.

2.1.2 A Simple Example

The Java source code in Figure 2.1 is an example of wrapping. The scenario is borrowed from previous work on migration by rewriting [8]. We migrate applications using `Vector` to `ArrayList`; thus, we re-implement `Vector` (line 1). Note that we preserve the superclass `AbstractList`. The `Vector` surrogate acts as an object adapter, maintaining the adaptee in an `ArrayList` field (line 2). Both objects have associated life-cycles: an `ArrayList` object is created in the constructor of a `Vector` (line 4). Because original and replacement APIs are very similar in this example, most client requests can be simply delegated to the adaptee (as in line 7) while some semantic adjustments are encapsulated in the adapter (lines 9-12).

As we migrate clients from `Vector` to `ArrayList`, we also need to migrate them from `Enumeration` to `Iterator`. Since `Enumeration` is a Java interface, we simply copy it to the wrapper (not shown in the figure). We need to provide a concrete implementation of the interface that acts as an adapter to `Iterator` and can be used by other types of the wrapping layer. For instance, `Vector`'s `elements` method (lines 13-15) creates a new adapter around the iterator returned by its `ArrayList` adaptee.

2.2 Challenges of XML API Wrapping

In this section, we present an exploratory study on behavioral mismatches between XML APIs. We first describe our overarching research questions and our general approach. Next, we detail the subjects of the study, our methodology, and the results. We then discuss how the results reveal important issues in API migration and suggest requirements for possible solutions. Finally, we consider threats to validity.

2.2.1 Research Questions

In this study we sought to understand how behavioral mismatches between APIs impact the implementation of a wrapper. Thus, we designed the study to address the following research questions:

RQ1 — What types of behavioral adjustments are necessary to adapt the behavior of the replacement API to comply with the expected behavior of the original API?

RQ2 — How do behavioral mismatches affect client applications?

To answer these questions we first manually developed a wrapper for the XML domain. Then, we analyzed the behavioral adjustments present in the wrapper. Finally, we evaluated the wrapper against a client application of the original API.

2.2.2 Subjects

We chose two APIs that offer a *document object model* (DOM) for XML because they are structurally similar: they define classes for the main elements of the XML specification. This allows us to concentrate on the low-level behavioral mismatches.

Subject$_{version}$	**XOM**$_{1.2.1}$	**JDOM**$_{1.1}$	**CDK**$_{1.2.0}$
	`www.xom.nu`	`www.jdom.org`	`cdk.sf.net`
Top-Level Packages	`nu.xom`	`org.jdom`	`org.openscience.cdk`
Types	110	73	1,079
Visible	50	52	1,008
Top-Level	50	50	998
Methods	884	908	10,643
Visible	358	644	9,291
Implementations	352	603	8,582
Fields	279	256	3,590
Visible	17	52	640
Constants	17	29	315
NCLOC	19,330	8,735	105,855

Table 2.1: Subjects of the XML wrapping study.

Table 2.1 presents the subjects of the study. XOM and JDOM are two of the principal DOM-like XML APIs for Java. They were developed independently, by different architects, in different code bases, and based on different design rationales [86]. CDK is a library for structural chemo- and bio-informatics; in the study, it represents an application to be migrated. We selected CDK because it is the largest open-source application in *sourceforge*[1] that makes substantial use of XOM in test cases.

The table lists the main packages and presents size metrics for the subjects (including sub-packages). *Visible* entities are accessible by client applications. In Java, public types count as visible whereas public and protected members of public types count as visible. We include protected members because clients may access them by inheritance. Top-level types exclude nested types. Method implementations exclude abstract methods. Constant are fields marked with `final`. NCLOC stands for *Non-Comment Lines of Code*.

XOM exposes around the same number of types as JDOM but fewer methods. JDOM provides many "convenience methods" that duplicate access to certain functionality [86]; however, XOM, is substantially more complex in terms of NCLOC. This difference involves several factors, including incidental ones such as programming style; most importantly, XOM makes considerable effort to guarantee XML well-formedness [86] by means of pre-condition checking, which directly affects NCLOC.

In the study, we developed a xom2jdom wrapper. That is, we implemented surrogates for XOM types by wrapping the JDOM API. Although wrapping an older API (JDOM)

[1]Sourceforge is a web-based, open-source code repository—`www.sf.net`

Test Suite	**XOM**	**CDK**
Test Classes	42	593
Test Methods	1,414	4,806
Assertions	3,483	10,478
NCLOC	22,157	60,462
Used Test Classes	17	57
Test Methods	720	484
Assertions	2,860	3,993

Table 2.2: Metrics on XOM and CDK test suites.

as a newer one (XOM) might appear counter-intuitive, this scenario is plausible in practice because migration drivers such as legal issues do not necessarily follow technical criteria. The main reason for selecting this migration direction is the availability of an extensive API test suite for XOM. The test suite represents the expected behavior of the API.

Table 2.2 shows metrics on the test suites provided by XOM and CDK. The upper part of the table shows information about the whole suites. Test classes are *JUnit*[2] test cases; test methods are methods in those classes that conform to JUnit 3's conventions: public, void, parameterless, and name starts with "test". We include the number of statically identified assertions in test classes to indicate properties checked by developers; an assertion is not guaranteed to be executed by a test case, but may be executed multiple times.

The lower part of Table 2.2 shows information about the test cases used in our study. We concentrated the development effort in types of the main XOM package; therefore, only test cases that target those types were used. We excluded test cases that target specialized topics, such as canonicalization and encoding. We also selected only the CDK test cases that use XOM and pass with the original XOM implementation. To this end, we executed the whole test suite and performed dynamic analysis to detect calls and references to XOM methods and fields, respectively. However, the number of assertions is the number of statically identified assertions found on the used test methods.

2.2.3 Methodology

The study was performed in three phases: wrapper development, analysis of behavioral adjustments, and analysis of test suite-based compliance.

[2]JUnit is a framework for unit tests in Java—`www.junit.org`

Wrapper Development

We started the development with an *exception-throwing wrapper*. We re-implemented each class of the original API with the same interface but with exception-throwing method implementations. We declared only visible entities (see Table 2.1), and set visible fields to null. We reused interfaces are as-is. The wrapper is compilable by construction with any client application of the original API.

The next step was the implementation of proper surrogates. We systematically applied the *ADAPTER* pattern as described in Section 2.1. We analyzed the APIs' inheritance hierarchies and made each XOM surrogate an object adapter to corresponding JDOM types. We implemented surrogate methods by delegating to similarly named adaptee methods.

Then, we iteratively evolved the wrapper to comply with the selected XOM test cases. In each iteration we executed the test suite, selected a failing test and tried to implement adjustments to make the test pass. We initially focused on tests that seemed easier to adjust and gradually implemented more difficult adjustments. We arbitrarily stopped the development when no clearly simple adjustments remained.

Analysis of Behavioral Adjustments

In this phase we addressed *RQ1*. We first defined categories of general behavioral adjustments, which we call *adaptation levels*. Then, we classified surrogate methods according to their adaptation levels. We defined the following possible adaptation levels for a given surrogate method m:

Adaptation Level 0 (Missing) — m is not implemented. This occurs when there is no straightforward way to implement the method using the replacement API or by complete re-implementation.

Adaptation Level 1 (Delegation) — m performs only basic delegation to a single adaptee method. Argument positions may be filled by defaults, and wrapping and unwrapping of parameter and return objects is allowed (as in Figure 2.1, line 14).

Adaptation Level 2 (Adaptation) — additional adaptations are involved in comparison to level 1: arguments may be pre-processed (converted or checked); results may be post-processed; exceptions may be translated; error codes may be converted into exceptions and vice-versa; the delegation may also be subject to simple argument checks.

Adaptation Level 3 (Composition) — m may invoke any number of adaptee methods, but without re-implementing any functionality of the original API. In informal terms, a level 3 method is effectively missing in the replacement API, but can be re-composed from some of its methods.

Adaptation Level 4 (Re-implementation) — m is a re-implementation of the original API method. In informal terms, level 4 methods violate the "intention to reuse" the replacement API.

Analysis of Test Suite-based Compliance

In this phase we addressed *RQ2*. We executed the selected CDK test cases with the wrapper instead of XOM. We then contrasted the wrapper compliance as asserted by the XOM test suite with the compliance according to the CDK test suite. We associated each surrogate method m with one of the following *compliance levels*, relative to each test suite:

All — all tests that exercise m pass: m is *always* compliant in the test suite.

Some — some tests that exercise m pass; others fail: m is *sometimes* compliant.

None — all tests that exercise m fail: m is *never* compliant.

Unused — m is not exercised by the test suite.

2.2.4 Results

RQ1 — Analysis of Behavioral Adjustments

Table 2.3 shows the main surrogates implemented in xom2jdom. We include only the surrogates targeted by test cases; additional surrogates were implemented as necessary, but were not systematically tested. Most adaptee types are JDOM counterparts found in `org.jdom`; types between parenthesis are JRE types. All surrogates except `Text` have a single adaptee type. `Text`, as indicated by the | symbol, has *alternative targets*: a single object of either type is used as adaptee (see Section 2.3.4).

Basic delegation (level 1) suffices for about 45% of all methods; around 28% requires some pre- or post-processing (level 2); the remainder need to be composed from other methods (level 3) or developed from scratch (level 4). Some methods of the `Serializer` surrogate were not implemented due to high complexity.

xom2jdom Surrogate	**Adaptee Types**	**Methods**	**Adaptation Level**				
nu.xom.	org.jdom.		0	1	2	3	4
`Attribute`	`Attribute`	28	-	16	7	1	4
`Attribute$Type`	`(java.lang.Integer)`	4	-	2	2	-	-
`Builder`	`input.SAXBuilder`	15	-	-	14	-	1
`Comment`	`Comment`	13	-	7	2	3	1
`DocType`	`DocType`	22	-	12	8	1	1
`Document`	`Document`	27	-	15	8	2	2
`Element`	`Element`	51	-	23	16	10	2
`Elements`	`(java.util.List)`	2	-	2	-	-	-
`Node`	-	2	-	-	-	-	2
`NodeFactory`	`JDOMFactory`	11	-	-	-	-	11
`Nodes`	`(java.util.List)`	9	-	6	-	3	-
`ParentNode`	-	-	-	-	-	-	-
`ProcessingInstruction`	`ProcessingInstruction`	17	-	14	1	2	-
`Serializer`	`output.XMLOutputter`	16	4	2	1	8	1
`Text`	`Text | CDATA`	13	-	4	7	1	1
`XPathContext`	`(java.util.Map)`	5	-	4	-	-	1
Total		235	4	107	66	31	27

Table 2.3: Adaptee types in xom2jdom surrogates and methods at each adaptation level.

Figure 2.2 presents examples for each adaptation level. All methods are from the `Element` surrogate, which has a `org.jdom.Element` adaptee. The `getChildCount` method simply delegates to the adaptee. The `insertChild` method needs to check pre-conditions which JDOM does not enforce, but which are enforced by XOM and its test suite. The `removeChild` method can be composed from JDOM's `indexOf` and `removeContent`, plus additional checking. Finally, JDOM's `Element` does not offer *base URI* functionality. The surrogate attempts to reproduce the behavior by resorting to the URI stored in the XML document that contains the element.

RQ2 — Analysis of Test Suite-based Compliance

Table 2.4 shows the results of executing the test suites using xom2jdom. The first row presents the total number of test method executions. For XOM this number equals the number of declared test methods in Table 2.2. But CDK's test suite contains an inheritance hierarchy of test cases, so that test methods in super types are executed for each sub-type. Thus, the number of test method executions is larger than the number of declared test methods. Subsequent rows present the breakdown into passed and failed executions.

```
// Level 1 − Delegation
public int getChildCount() {
    return adaptee.getContentSize();
}

// Level 2 − Adaptation (pre−processing)
public void insertChild(Node child, int position) {
    if (child == null) {
        throw new NullPointerException("inserting null child");
    }
    if (child instanceof Document) {
        throw new IllegalAddException("documents cannot be childs of elements");
    }
    if (child.getParent() != null) {
        throw new MultipleParentException("child has a parent");
    }
    adaptee.addContent(position, child.getAdaptee());
}

// Level 3 − Composition
public Node removeChild(Node child) {
    int index = adaptee.indexOf(child.getAdaptee());
    if (index == −1) {
        throw new NoSuchChildException("child is not a child of this adaptee");
    }
    adaptee.removeContent(child.getAdaptee());
    return child;
}

// Level 4 − Re−implementation
public String getBaseURI() {
    org.jdom.Attribute base = adaptee.getAttribute("base",
                                        org.jdom.Namespace.XML_NAMESPACE);
    if (base != null) {
        if (base.getValue().equals("")) {
            String uri = adaptee.getDocument().getBaseURI();
            return uri == null ? "" : uri;
        }
        return base.getValue();
    }
    if (getParent() != null) {
        return getParent().getBaseURI();
    }
    return "";
}
```

Figure 2.2: Examples of adaptation levels from `Element` surrogate methods.

Test Suite	**XOM**	**CDK**
Test Method Executions	720	761
Passed	419	761
Failed	301	0

Table 2.4: xom2jdom compliance with respect to XOM and CDK test suites.

All CDK test cases pass, while approximately 40% of XOM test cases fail. This result was achieved without any adaptation of CDK, except for three test cases whose dependence on the order of XML attributes had to be relaxed. One of the reasons for the increased compliance is lower coverage: out of 235 implemented XOM methods, XOM's test suite exercises 156 (about 66%), but CDK's test suite exercises only 35 (about 15%).

Figure 2.3 presents compliance levels for xom2jdom methods, according to the categorization described in Section 2.2.3. For each surrogate type (see Table 2.3), we classified method implementations with respect to XOM and CDK test suites. XOM and CDK bars are side-by-side to show the classification of individual methods in the y axis. For instance, two methods of `Attribute` that were classified as **All** in XOM stayed in **All** in CDK; 11 changed from **All** to **Unused**; one changed from **Some** to **All**; and three went from **Some** to **Unused**. Since all CDK test cases pass, CDK bars show only **All** and **Unused**.

The plot shows that several methods with compliance issues (classified as **Some**) with regard to XOM are used without problems in CDK (classified as **All**): `Attribute`, `Builder`, `Document`, `Element`, `Elements`, `Nodes`, and `Serializer`. Furthermore, some implementations that were not exercised by the API's test suite were exercised and found compliant by CDK's test suite: `Element` and `XPathContext`.

2.2.5 Discussion

RQ1 — Behavioral Adjustments

In the study, approximately half of the surrogate methods were implemented with basic delegation; the other half demanded behavioral adjustments or re-implementation. Three factors are necessary to require a behavioral adjustment: *i)* a behavioral mismatch between the APIs must exist, *ii)* the mismatch must be exposed by the client, and *iii)* the client must enforce the original behavior, say, via a test case assertion. The more the APIs differ and the more the client exercises and enforces the behavior of the original API, the more methods will need more adjustments and, hence, will go up in the adaptation level scale.

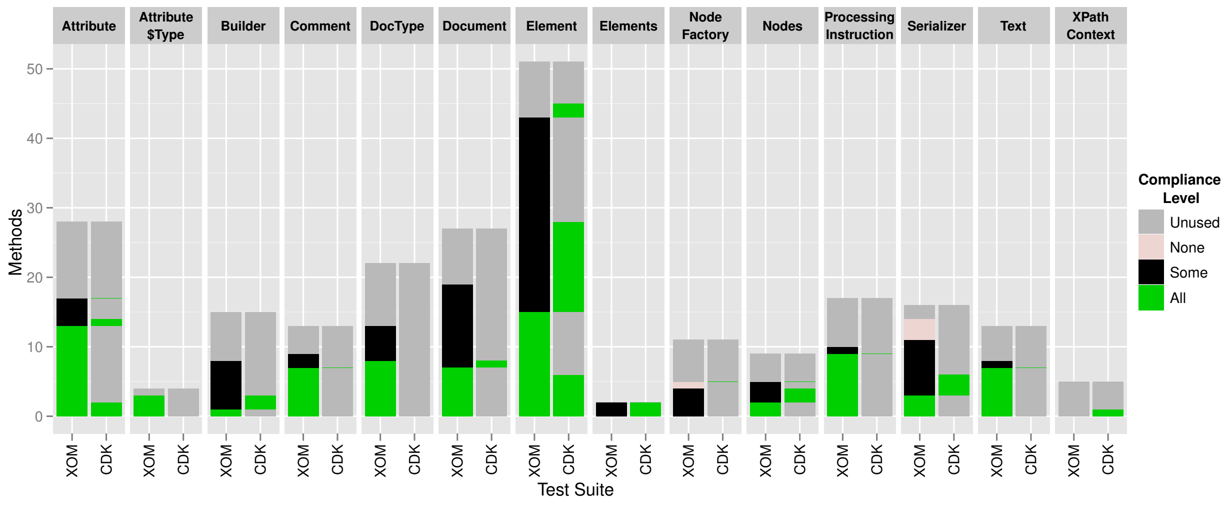

Figure 2.3: Compliance levels of xom2jdom methods w.r.t. XOM and CDK test suites.

The results suggest that a methodology is needed to systematically uncover API behavioral mismatches and, crucially, to identify the adjustments that are relevant to the client. The results also have implications to rewriting approaches. We discussed in Section 1.1 that two options to implement adjustments are via a Facade or by inlining the adjustments at call sites. While it is simple to inline level 1 methods, level 2 methods would bloat call sites with checking code. Also, there is no straightforward solution to inline methods at levels 3 and 4. A Facade can encapsulate these adjustments, but many call sites (about 25% in our study) will be redirected to it instead of the replacement API.

RQ2 — Test Suite-based Compliance

Our results indicate that behavioral mismatches do not necessarily affect client applications. The xom2jdom wrapper passed all client CDK test cases, while it failed many internal XOM test cases. The breakdown in Figure 2.3 suggests that a surrogate method that does not comply with the behavior enforced by one application may participate in the execution of another application without problems, as long as the new application either does not expose the mismatch or the mismatch is not enforced.

The principal insight of this study is that a viable API migration must be performed with respect to an *application under migration*. The results show that applications exercise the original API in different ways and expect different levels of behavioral compliance.

It may be impractical to cover all and every case in advance. But even if a particular application is not available, the migration can target a generalized reference application, created with test cases that exercise the parts of the original API deemed most valuable.

2.2.6 Threats to Validity

The main threats to *internal validity* concern errors in the classification of surrogate methods, and general experimenter bias. We developed the xom2jdom wrapper, defined categories for adaptation and compliance levels, and classified wrapper methods without external validation. Compliance level classification, however, is objective, determined by test cases implemented by other developers. We tried to minimize the errors in adaptation level classification by having two of the authors of the study [10] independently classify methods and then compare the results and resolve differences.

The main threat to external validity relates to our choice of subjects. We studied a single domain, using a single programming language in a single migration instance. We expect the results to generalize to other object-oriented languages and migration instances. The use of the XML domain, however, may have exaggerated the amount of adjustments, with more methods classified in levels 3 and 4 than in other domains. This does not invalidate our claim that applications may exercise the API in different ways and, thus, one should concentrate the migration effort on API behavior needed by specific applications.

2.3 Challenges of GUI API Wrapping

In this section, we report on an artifact study about structural mismatches between GUI APIs. We follow the same organization as the previous study: we describe research questions, approach, subjects, methodology, and results. Finally, we discuss the implications of the results and threats to validity.

2.3.1 Research Questions

In this study we investigated how structural mismatches between APIs impact the design of a wrapper. The study addresses the following research questions:

RQ1 — What are the design challenges faced by developers when implementing wrapping layers around object-oriented APIs?

RQ2 — What are the solutions employed by developers in practice?

To answer these questions we selected existing large, open-source wrappers for the GUI domain. Then, we studied the wrappers and identified key design challenges faced by their developers. Next, we abstracted design patterns from idioms created by developers as solutions to the identified challenges. Finally, we validated the design patterns by demonstrating their use in practical wrappers.

2.3.2 Subjects

The study is based on two open-source projects implementing wrappers for both directions between the principal Java GUI APIs. Table 2.5 presents the wrappers and APIs[3]. SwingWT implements surrogates for Swing on top of SWT; SWTSwing implements the other direction. While SWTSwing preserves SWT's namespace—indicated by the top-level packages—SwingWT declares its own namespace, parallel to Swing's packages. Thus, Swing applications must rewrite their package import declarations to use the wrapper.

We selected these wrappers for the following reasons: they are the most extensive wrappers we could find; they wrap large, complex, varied APIs, which allows us to focus on design issues; they are open-source, so we can study their source code; and we had access to their main developers to corroborate our findings.

Table 2.5 presents size metrics for wrappers and APIs (see the description of metrics for Table 2.1). The table shows that all wrappers have fewer visible elements (types, methods and fields) than their original APIs. That is, the wrappers are not complete. A client application that references some of the missing elements would not compile if used with the wrappers. As expected, the majority of fields exposed by the APIs are constants (about 65%). Even though the wrappers cover only parts of the APIs, the wrapper sizes are substantial: SwingWT has about 34% the NLOC of SWT; SWTSwing has about 28% of Swing. We compare a wrapper with the corresponding replacement API because it indicates the amount of source code needed to adapt and reuse that API.

2.3.3 Methodology

The study was conducted in three phases: analysis of design challenges, abstraction of idioms into design patterns, and validation.

[3]Swing is built on top of AWT; both are core Java APIs. SWT is an independent API under Eclipse.

Subject$_{version}$	SwingWT$_{0.90}$	SWTSwing$_{3.2}$	Swing$_{1.4}$	SWT$_{3.3.2}$
	`swingwt.sf.net`	`swtsing.sf.net`	Core Java API	`www.eclipse.org/swt`
Top-Level Packages	`swingwt.awt` `swingwtx.swing` `swingwtx.accessibility`	`org.eclipse.swt`	`java.awt` `javax.swing` `javax.accessibility`	`org.eclipse.swt`
Types	864	752	2,032	696
Visible	527	361	1,212	432
Top-Level	475	308	791	432
Methods	6.537	6,044	18,702	9,607
Visible	5,650	3,246	12,833	5,966
Implementations	5,124	2,785	11,995	5,851
Fields	2,356	2,636	7,440	5,414
Visible	1,845	1,068	2,965	2,691
Constants	1,196	572	1,943	1,743
NCLOC	33,582	66,104	234,731	99,205

Table 2.5: Wrappers and APIs subject of the GUI wrapping study.

The first phase addressed *RQ1*. It consisted in uncovering important design challenges faced by developers. We initially studied the APIs and detected incompatibilities that would potentially impact the wrapper design. We then talked to the developers about the general structure of their wrappers and the main challenges they experienced. Finally, we investigated the source code to understand the correspondences between types of the original and replacement APIs. The results of this phase are described in Section 2.3.4.

The second phase addressed *RQ2*. It comprised understanding the idioms employed by developers to solve the identified challenges and eventually abstracting idioms as design patterns. To this end, we performed architectural code queries and manual inspection of source code. Our investigation resulted in the five design patterns presented in Chapter 3.

In the last phase we designed simple metrics to provide evidence that the patterns are indeed applied in practice. These results are described in Section 6.2.

2.3.4 Results

In this section, we describe the results of the first phase, which concern *RQ1*. We examine the challenges that emerge when designing wrappers around Swing and SWT. We use two versions of a simple program with a GUI: one version using Swing, another version using SWT. First, we observe the differences between the two versions. Then, we discuss the

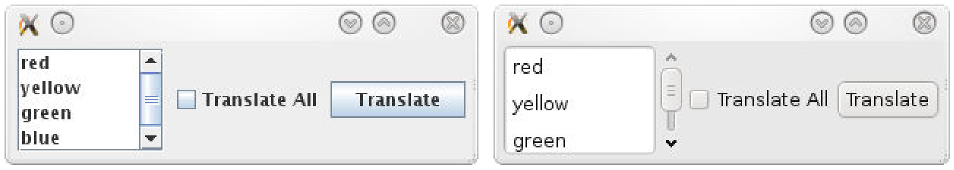

Figure 2.4: Swing and SWT versions of a simple GUI.

issues that arise in defining wrapping layers that serve both directions. We use swing2swt and swt2swing to refer to these wrappers. The swing2swt layer wraps SWT and exposes it with Swing's interface; application code written for Swing can execute with this wrapper and get SWT widgets instead. The swt2swing wrapper implements the opposite direction, wrapping Swing and exposing SWT.

Sample-based Inquiry into API Differences

Figure 2.4 shows the Swing and SWT-based GUIs for a program that translates words from a list box: all words or only the selected ones. Figure 2.5 shows the source code of the program with the Swing and SWT versions next to each other.

Lines 1-34 create the window and the three widgets: a scrollable list of predefined words, a checkbox, and a push button. The handler of the button calls the reusable `Translator` class which is also configured with the value of the checkbox (lines 20-21). The called method (lines 36-40) recursively searches for lists in a window, and depending on the `all` boolean flag, translates all words in the list or just the selected ones.

```java
final JFrame frame = new JFrame();
Container pane = frame.getContentPane();

JScrollPane scroll = new JScrollPane();
scroll.setPreferredSize(new Dimension(100, 70));
DefaultListModel model = new DefaultListModel();
model.addElement(“red”);
model.addElement(“yellow”);
model.addElement(“green”);
model.addElement(“blue”);
JList list = new JList(model);

final JCheckBox check = new JCheckBox(“Translate All”);


JButton translate = new JButton(“Translate”);

translate.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        new Translator().translateLists(
          frame, check.isSelected());
    }
});

pane.setLayout(new FlowLayout());

scroll.getViewport().setView(list);
pane.add(scroll);
pane.add(check);
pane.add(translate);

frame.pack();
frame.setVisible(true);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
...
public class Translator {
  public void translateLists(Container container, boolean all) {
    for (Component c : container.getComponents()) {
        if (c instanceof JList) { ... }
}}}
```

```java
Display display = new Display();
final Shell shell = new Shell(display);

List list = new List(shell,
    SWT.BORDER | SWT.MULTI | SWT.V_SCROLL);
list.setLayoutData(new RowData(100, 70));
list.add(“red”);
list.add(“yellow”);
list.add(“green”);
list.add(“blue”);


final Button check = new Button(shell, SWT.CHECK);
check.setText(“Translate All”);

Button translate = new Button(shell, SWT.PUSH);
translate.setText(“Translate”);
translate.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        new Translator().translateLists(
            shell, check.getSelection());
    }
});

RowLayout layout = new RowLayout();
layout.center = true;
shell.setLayout(layout);

shell.pack();
shell.open();
while (!shell.isDisposed ()) {
    if (!display.readAndDispatch()) display.sleep();
}
display.dispose();
...
public class Translator {
  public void translateLists(Composite composite, boolean all) {
    for (Control c : composite.getChildren()) {
        if (c instanceof List) { ... }
}}}
```

Figure 2.5: Swing (left) and SWT (right) source code versions of the GUI application displayed in Figure 2.4.

The two source code versions are relatively straightforward. In Swing (Figure 2.5 on the left), `JFrame` (line 1) represents the main window and `JScrollPane` (lines 4-5) creates the scrollable area in which the list resides. `JList` (line 11) implements the Model-View-Controller [73] (MVC) *view*, while `DefaultListModel` (lines 6-10) implements the MVC *model* expected by `JList`. `JCheckBox` (line 13) and `JButton` (line 16) are the button widgets. Lines 18-23 register an `ActionListener` with the `JButton`. The listener receives a callback from the API when the button is pressed. A layout is created for the window (line 25) and the widgets are wired together (lines 27-30). Swing maintains its own thread; therefore, lines 32-34 just open the window and set the default operation for when the window is closed, leaving the application thread free from GUI concerns.

The SWT version (Figure 2.5 on the right) is startlingly similar. `Shell` (line 2), `List` (lines 4-5) and `Button` (line 13 and line 16) represent the window, list, and button widgets, respectively. Notable differences include `Button` being configured to behave as a checkbox (line 13) or a push button (line 16), and the lack of a scroll pane widget since `List` can be configured to show a scroll bar (line 5). Furthermore, widgets do not have to be wired since they must always receive a parent in the constructor, except for top level widgets such as `Shell`. Finally, SWT does not maintain its own thread; thus, lines 31-33 implement a loop to control the application's main thread.

Non-trivial Mapping Multiplicities

The correspondences between original and replacement API types form an *API type mapping*. The trivial `Vector` example of Section 2.1 involved only one-to-one correspondences between adapter and adaptee. The GUI wrappers require additional mapping multiplicities, as presented in Table 2.6, that pose challenges to the wrapper design.

By analyzing the code of Figure 2.5 we note that SWT's `Display` (line 1) does not have a counterpart in Swing: this is a case of *No Target*. While swing2swt can simply ignore this type, swt2swing has to re-implement its services completely.

A challenge caused by *Alternative Targets* arises when an original API type implements variants distributed over different replacement API types. SWT's `Button`, for example, maps to both of Swing's types `JButton` and `JCheckBox`, but each specific `Button` object maps to either a `JButton` or a `JCheckBox` object. This represents an issue for swt2swing because a surrogate for `Button` must decide at runtime which type of adaptee object is to be created. This issue also occurs in the `Text` surrogate of the xom2jdom wrapper: it must select between JDOM's `Text` and `CDATA` (see Table 2.3).

Name	**Multiplicity** (type/object)	**Example**
No Target	0/0	Display$\rightarrow \emptyset$
Single Target	1/1	Vector$\rightarrow$ArrayList
Alternative Targets	*/1	Button$\rightarrow$JButton | JCheckBox
Composite Targets	*/*	List$\rightarrow$JList, ListModel

Table 2.6: Mapping multiplicities w.r.t. a single original API object.

Figure 2.5 also suggests mappings from `JFrame` to `Shell` and `JList` to `List`. But closer inspection shows that a `Shell` object in fact corresponds to a `JFrame` plus its content pane, an object of type `Container`. Further, SWT's `List` already encapsulates its model and can be scrollable. It should map, therefore, to a `JList` plus a `ListModel` and a `JScrollPane`. These are examples of *Composite Targets*, which force surrogates to maintain multiple adaptees of potentially many types. The swt2swing `List` surrogate, for example, must keep `JList` and `ListModel` adaptees.

Note that *Composite* and *Alternative Targets* can be simultaneously present in a type mapping. SWT's `List`, for example, can have variants with and without scroll bars. Its mapping with respect to Swing would then be:

$$List \rightarrow (JList, ListModel)|(JList, ListModel, JScrollBar)$$

Table 2.6 shows only multiplicities with respect to a single original API object. The opposite direction of a *Composite Targets* mapping, however, also represents an issue to wrappers because surrogates must coordinate around a single adaptee object. In swing2swt, for example, `JScrollPane`, `JList`, and `DefaultListModel` surrogates have to share a reference to a single `List` object. Finally, mapped types may not completely match in the features they offer and data they carry. `Shell`'s `disposed` flag (line 31), for example, is missing in `JFrame`, and must be re-implemented by the surrogate.

Mapping Varying Type Hierarchies

Figure 2.6 shows an excerpt of Swing and SWT's class hierarchies, with the type mappings conjectured above. Further, `Component` is mapped to `Widget` since they are the supertypes of all widgets in their corresponding APIs. Also, `Container` is mapped to `Composite` because they represent widgets that can have child widgets in the sense of the *COMPOSITE* design pattern [34]. For `List`'s case of a *Composite Target* we use a directed edge to

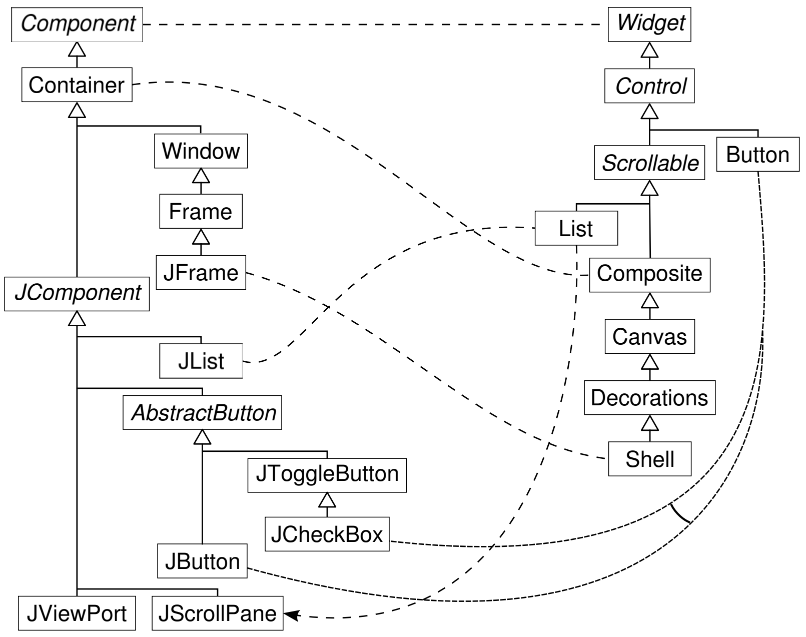

Figure 2.6: One option for Swing and SWT Type Mappings.

`JScrollPane` which expresses uni-directionality. Finally, we note that the mappings of the open-source wrappers SwingWT and SWTSwing are slightly different from Figure 2.6.

Wrappers must provide surrogates for the whole inheritance hierarchy of the original API for two reasons. First, client applications can reference and extend any visible type of the original API. Clients using surrogates that do not comply with the hierarchy may not even compile. For instance, if swing2swt only exposes `JList`, without exposing its super-types, clients such as `Translator` (l.36), which references `JList`'s super-type `Container`, would break.

Second, wrappers can take advantage of the original API's inheritance decomposition to reuse wrapping code. For instance, `JList` and `JButton` have `Component` as a common super-type and could reuse its methods. However, this kind of reuse can only be achieved if type mappings respect covariance. Figure 2.6 shows that `Component`$\rightarrow$`Widget` and `Container`$\rightarrow$`Composite` are covariant mappings. Therefore, swing2swt can implement `Component` methods using a `Widget` adaptee and `Container` will be able to reuse these methods since its adaptee, `Composite`, is a `Widget`. Breaking covariance indicates a potentially serious API mismatch. In Swing, for example, buttons extend `Container` and can, thus, have child widgets. Since SWT's `Button` does not extend `Composite`, Swing clients that attach widgets to buttons cannot be easily migrated to SWT.

Different functional decomposition of types in the inheritance hierarchy can lead to additional challenges, besides broken covariance. In SWT, for example, a widget becomes scrollable through inheritance, by extending `Scrollable`, whereas Swing uses composition with `JScrollPane`. Finer grained functionality can also be offered at different levels in the hierarchy. SWT's `Control.pack` method, called in the shell object (line 29), for instance, maps to `Window.pack` in Swing. Since `Control` maps to `Component`, the surrogate must verify that the adaptee is actually a `Window` before delegating the method call.

Varying Creation and Wiring Protocols

Swing provides distinct methods for creating widgets and wiring them to parents. Swing clients can dynamically create and modify their compositions. In contrast, SWT enforces the parent-child relationship in constructors, which constrains its clients.

Migrating from an API that is relaxed in how it establishes object relationships to one that enforces relationships at construction time presents a challenge to wrappers. Surrogates may need to delay adaptee construction. For example, when a surrogate for `JCheckBox` is instantiated (l.13), it cannot immediately construct the corresponding SWT `Button` adaptee because the parent widget is unknown until the checkbox is wired to the content pane (l.29). This also means that operations cannot be delegated as usual: they may need to be re-implemented.

Mapping from a strict to a relaxed API does not present major challenges: surrogate constructors can create the adaptee and wire it to its parent immediately.

Inversion of Control

Wrappers may need to delegate control from the replacement API back to the application. A common scenario is that an application receives control from the wrapper through callbacks, by implementing suitable interfaces or extending suitable classes of the original API. If there is a correspondence between callbacks of both APIs the wrapper can delegate events generated by the replacement API to callbacks designed for the original API.

For instance, our Swing application registers an `ActionListener` with a `JButton` (lines 18-23), which corresponds to adding a `SelectionAdapter` to an SWT's `Button`. In swing2swt, the `JButton` surrogate must register a `SelectionAdapter` to its `Button` adaptee and keep the `ActionListener`. When SWT generates a `SelectionEvent`, it must be translated into an `ActionEvent` and delegated to the original `ActionListener`. The analogous opposite mapping is implemented by swt2swing.

Object Schizophrenia

A surrogate must wrap all objects of the replacement API before they are returned to the application. For instance, the surrogate method for `Container.getComponents` (line 38) calls the corresponding `Composite.getChildren` on its adaptee and receives objects of replacement API types. The surrogate must then wrap those objects in original API types before returning to the application.

It can happen that the same object flows many times from the replacement API to the application. This may create garbage and potentially cause object schizophrenia [36] if a new surrogate is created each time the method is called. That is, the application will receive two objects with different identities but that represent a single object of the replacement API. Thus, it is necessary to manage the surrogate identities of objects of the replacement API so that new surrogates are only created when necessary. This challenge is exacerbated by the fact that the type of the surrogate is not always fully implied only by the type of the object from the replacement API. For example, an SWT `Button` could be encapsulated in a surrogate for `JButton` or `JCheckBox`, depending on its runtime properties.

Additional Design Challenges

We found additional challenges specific to the studied APIs. First, Swing and SWT approach thread handling differently. In Swing, the application thread is released after creating the GUI, and a Swing-specific daemon thread maintains the windows and sends events to registered handles. In SWT, the application must control the main thread, looping while the window is not disposed and dispatching widget events. The result is that swing2swt must control SWT's thread on behalf of client applications since they are not designed with this concern in mind. On the other hand, swt2swing has to synchronize the application's and Swing's threads.

Second, Swing and SWT use layouts to determine how widgets are arranged in a window, but perfect correspondences between the layout types available in the APIs is rare. Even for the simple layouts used in the example, `FlowLayout` in Swing and `RowLayout` in SWT, the way in which applications provide layout information varies: while in Swing the desired size is set in the `JScrollPane` (line 5), in SWT the information is passed as `LayoutData` to the `List` (line 6). Because layouts are a well modularized concern of the APIs, the studied wrappers re-implemented the original API layouts completely, without trying to reuse replacement API layouts.

2.3.5 Discussion

The artifact analysis shows that developers face considerable design challenges in implementing adaptation layers around object-oriented APIs. The results presented in this section elucidate *RQ1*: we have identified several issues caused by structural mismatches between Java GUI APIs. Unfortunately, developers lack clear guidance on how to solve these issues.

The main insight of this study is that design patterns can be used to guide developers. While studying the wrappers with respect to *RQ2* we noticed that developers often create informal idioms to overcome design issues in their projects. In Chapter 3 we present a catalog of API wrapping design patterns we abstracted from some of the idioms.

2.3.6 Threats to Validity

The main threat to internal validity is related to the identified challenges and idioms. Although we were systematic in analyzing the projects, we may have missed important challenges and idioms.

Threats to external validity concern the generalization of our results and involve our choice of subjects. We think that the challenges, idioms and patterns identified are not dependent on the specific APIs and could be generalized to other settings. The quality of the selected wrappers could also influence the validity of the identified idioms and, therefore, of the abstracted patterns. We hold that the GUI wrappers are successful projects worth being studied given the complexity of the APIs they wrap, their claims of compatibility with the respective original APIs and the extensive user base.

2.4 Interviews

In this section, we describe a set of interviews we conducted with developers experienced in API migration. The overarching goal of the interviews was to understand the process and techniques developers use in practice and to corroborate the findings of our previous studies. First, we describe the interview questions, subjects, and methodology. Then, we present the results, discussion and threats to validity.

2.4.1 Interview Questions

We asked developers the following general open questions:

IQ1 — What is the development process used in practical API migration?

IQ2 — What are the techniques used for verification of results?

IQ3 — What is the existing and desired tool support?

2.4.2 Subjects and Methodology

We conducted interviews with seven developers familiar with some form of API migration: wrapping and/or rewriting. We sought experienced developers from a varied background, but were constrained by familiarity with API migration. We contacted three developers of the GUI wrappers we presented in Section 2.3, two developers we were acquainted with, and two developers from a company specialized in porting applications across mobile devices[4]. Our subjects came from four different countries, had at least five years of professional development experience and participated in at least one API migration effort, in domains such as banking, gaming, GUI, web services, logging and geographical information systems.

We performed five guided interviews by phone and two interviews by email. The phone interviews were recorded and ranged from 30 minutes to 1:30 hours. We organized the interviews around *IQ1-3*, but left developers free to express their views on the subject of API migration. Due to the small sample size, we were only able to perform a qualitative evaluation. We analyzed the recordings of phone interviews and the responses in the email interviews, and created interview summaries. Then, we analyzed the summaries to categorize recurrent statements, and to collect observations we found interesting.

2.4.3 Results

We summarize the outcome of the interviews as follows.

[4]Porting is related to API migration in that mobile devices offer different APIs to access device services.

Specific Applications

All reported API migration efforts used specific applications to drive the development process. Even wrapping projects aiming at general wrappers initially targeted a single application. For instance, the SWTSwing wrapper studied in the previous section, which is one of the wrappers covered by the interviews, was developed initially to migrate the Java *Integrated Development Environment* (IDE) of *Eclipse*[5] from SWT to Swing. One of the developers of SwingWT also reported that his contributions to the wrapper originated from attempting to migrate a personal application from Swing to SWT.

Simple Samples

Even with specific applications in mind, developers usually start with simple samples. In the case of API migration by rewriting, developers reported initially targeting samples so that they can assess the migration effort. In the case of API migration by wrapping, developers reported to create small samples that imitate API usage in the application. Such samples are then used to evolve the wrapper until it becomes possible to execute the target application with it. In addition to hand-coded samples, developers also reported leveraging simple open-source applications that exercise the original API.

Limited Test Automation

Developers reported that, in most cases, they simply try to informally compare the behavior of the application before and after migration. For example, they look at the resulting GUI snapshots, files, or the return value of some important functions. A simple *crash-driven* method is often used: developers try to run the application under migration until it crashes; such crashes are debugged and analyzed as they indicate incompleteness or non-compliance of the wrapper with respect to the original API.

One interviewee reported an approach for a more rigorous testing process in banking applications. Developers first recreate missing, incomplete or outdated requirements documents. Then, they derive use cases from the requirements. Finally, they manually implement automated test cases for the use cases, so that essential business flows can be verified automatically. In essence, the migration is considered correct if tests designed for application use cases pass; no API-specific tests are created. Nevertheless, this approach is restricted to critical systems because it is labor intensive.

[5]Eclipse is an extensive open-source development platform in Java—`www.eclipse.org`

Relaxed Behavioral Equivalence

All developers of wrappers mentioned that strict behavioral equivalence between the original API and wrapper was never a goal. No developer aimed for strict behavioral equivalence between the original and the migrated application. These expectations match the results of our previous studies: differences between APIs are often too hard to be neutralized completely. Strict equivalence is reportedly considered unnecessary. Developers are content with a migration that is "good enough".

In the case of wrapper-based API migration, some developers reported that limited, manual adaptation of the application is sometimes recommended when it simplifies the wrapper implementation or enables better utilization of the replacement API. SWTSwing, for example, recommends that applications add initialization code that greatly improves SWTSwing's synchronization between application and Swing threads (see *Additional Design Challenges* in Section 2.3.4).

Conceivable Tool Support

Developers varied in their opinions about conceivable tool support. Some developers expressed interest in automated comparison of results. In certain domains, such as in the GUI domain, developers expressed concerns about the feasibility of automated comparison. For instance, developers of SWTSwing used a tool for opening side-by-side the same application—once with SWT, once with SWTSwing—thereby simplifying the manual comparison of states of the GUI, including details of appearance. A general, automated comparison was considered infeasible.

One developer argued that automated rewriting tools should not be trusted for complex migration tasks because the code may become unmaintainable, while the same could be true for generated wrappers. Finally, one developer expressed that debugging and regression testing may benefit from the selective use of injected observation functionality to be applied to objects of interest. That is, a tool to observe abstracted state of objects during execution with the original API and then compare with the state during execution of the wrapper.

Wrapper Design

Most developers of wrappers expressed that wrapping is especially well suited for encapsulating and localizing behavioral adjustments. Developers of GUI wrappers also reported that one of the main issues related to the design and organization of source code to implement a mapping between very different APIs.

2.4.4 Discussion

The results of the interviews corroborate the findings of our XML study (Section 2.2): developers confirm the use of a reference application as the main migration target, and report difficulties in achieving strict behavioral compliance. The results also confirm the importance of the design challenges we have found in our GUI study (Section 2.3).

The interviews also reveal the current industry practice, which we call the crash-driven method. First, developers use small samples that exercise the original API to evolve the wrapper until the application under migration can be executed. Then, developers execute the application, linked with the wrapper, in use cases deemed important. The development is driven by application crashes and informal assessment of application behavior.

Furthermore, the interviews suggest requirements for an improved method. Developers express difficulties in uncovering API differences that are important to the application under migration. Tool support for behavioral comparison using relaxed equivalence, and state observation of selected objects is desired. Finally, test automation is mostly missing.

2.4.5 Threats to Validity

The main threat to internal validity relates to our interpretation of the interviews. We mitigated the threat by striving to conduct and analyze the interviews as impartially and systematically as possible.

A major threat to external validity is sampling. We were able to interview only seven developers and we used convenience sampling. We tried to minimize this threat by selecting subjects with varied backgrounds. We think that the subjects adequately represent the current industry practice.

2.5 Summary

In this chapter, we have introduced the basics of API migration by wrapping, and presented two studies and a set of interviews we have conducted to better understand API migration in practice. Together, these activities suggest that the most pressing issues involve discovery and specification of differences between APIs, and assessment of migration correctness. In the next chapter, we tackle the issue of how to specify the differences between APIs. We address the lack of guidelines for wrapper design with a catalog of API wrapping design patterns we have abstracted from the solutions used by developers in practice.

Chapter 3

API Wrapping Design Patterns

In this chapter, we present a catalog of *API wrapping design patterns.* The catalog addresses *RQ2* of the study presented in Section 2.3. The patterns abstract idioms found in the practical GUI wrappers of the study. Each pattern addresses at least one of the challenges described in Section 2.3.4. We abstracted the following patterns:

- *Layered Adapter* decomposes surrogates into layers, providing flexibility to implement mappings with Alternative and Composite Targets (see Table 2.6 and Section 2.3.4).
- *Stateful Adapter* keeps additional state in an adapter to re-implement features missing in the adaptee or to adjust differences in interaction protocols (see discussion of `Shell`'s `disposed` flag in Section 2.3.4).
- *Delayed Instantiation* delays the instantiation of adaptees until enough data is present; placeholders are used to store interim data and execute operations, thereby addressing the challenge of varying creation and wiring protocols.
- *Inverse Delegation* provides adapters in the inverse direction, in order to delegate replacement API events to the application, thereby addressing the challenge of inversion of control.
- *Wrapping Identity Map* maintains the correspondence between adaptees and surrogates in an identity map that is consulted when returning data to the application. This pattern addresses the challenge of object schizophrenia.

In the next sections, we present each pattern in detail. However, we refrain from using the heavy structure of the GoF catalog [34]. Instead, we detail patterns with examples taken from the wrappers, and focus on discussing challenges addressed and design options.

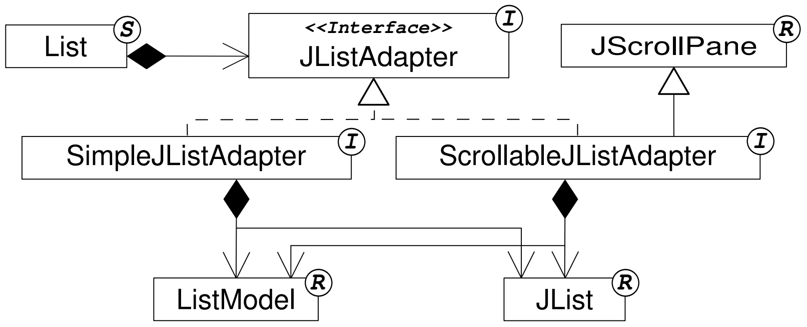

Figure 3.1: *Layered Adapter* for Alternative and Composite targets. Class diagrams throughout the thesis identify Ⓢurrogate, Ⓘnternal, Ⓡeplacement API and Ⓐpplication types.

3.1 Layered Adapter

The classic *ADAPTER* pattern can be directly applied to *Single Target* mappings, as defined in Table 2.6. *No Target* mappings, on the other hand, must be completely re-implemented by a surrogate. Mappings with *Alternative* and *Composite Targets* demand more flexibility because the surrogate may need to chose different combinations of adaptees at runtime.

In practice, we have seen that this flexibility is achieved by decomposing the surrogate into a layered adapter. The class diagram in Figure 3.1 shows the general structure of the pattern in the context of swt2swing's `List` surrogate implementation (swt2swing is the example wrapper described in Section 2.3). Instead of directly referencing replacement API types, `List` maintains an object of an interface type, `JListAdapter`. Alternative mappings are uniformly accessed by the surrogate through the implementation of different variants of the interface. Each variant is an adapter with potentially many *Composite Targets*.

For example, the `SimpleJListAdapter` variant composes `JList` and `ListModel`, whereas `ScrollableJListAdapter` additionally inherits `JScrollPane`. This structure naturally represents the mapping definition for `List` given as a regular expression in Section 2.3.4: $List \rightarrow (JList, ListModel)|(JList, ListModel, JScrollBar)$

3.2 Stateful Adapter

The main goal of the *ADAPTER* pattern is to reuse existing functionality. Unfortunately, delegation is often not possible because mapped types may not agree on the features they

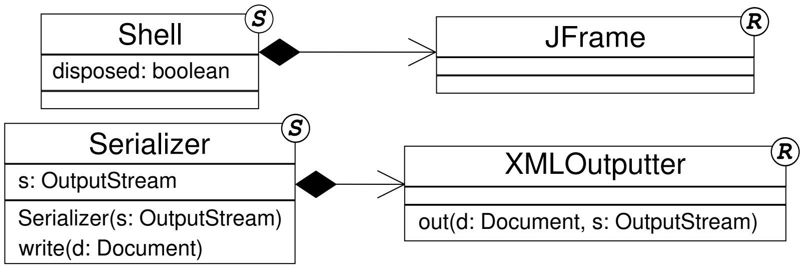

Figure 3.2: *Stateful Wrapper* examples.

offer and data they carry. If an original API type assumes a richer state than its replacement API counterparts, requests cannot be simply delegated to adaptees—a *Stateful Adapter* must carry the missing data.

Figure 3.2 outlines the structure of the two main scenarios in which the pattern is used. In the first, `Shell`'s surrogate re-implements functionality missing from `JFrame`, the disposed flag (see Section 2.3.4). The second scenario comes from xom2jdom, the wrapper described in Section 2.2, and exemplifies adjustment of interaction protocols. XOM clients use `Serializer` to write XML documents to an output stream. Clients first set the output stream in the constructor and then call `write` methods passing only the document to be written. `XMLOutputter`, JDOM's counterpart, uses a different protocol: clients must pass both the stream and the document at each method call. Thus, the surrogate collects the state and adjusts the protocol when dispatching operations.

3.3 Delayed Instantiation

Surrogates usually have their lifecycles bound to adaptees. When migrating to replacement APIs that enforce object relationships at construction time, surrogates may need to delay construction of adaptees until the relationship is established in terms of original API protocols. For example, in the swing2swt excerpt of Figure 3.3, `JButton` cannot create its `Button` adaptee in a constructor because the parent object is yet unknown—the parent-child relationship is only established when the `JButton` object is added to a `Container` with `Container.add` (line 16).

The *FACTORY METHOD* pattern [34] provides infrastructure for delayed instantiation since it decouples surrogate and adaptee lifecycles. In the example, the `Component` surrogate declares the `createAdaptee` factory method (line 3) to be implemented by sub-classes. While no adaptee is available, operations on surrogates must be deferred or re-implemented

```
 1  public abstract class Component {
 2      Widget adaptee;
 3      abstract void createAdaptee(Composite parent);
 4      ...
 5  }
 6  public class Container extends Component {
 7      List<Component> pending = new ...;
 8      void createPending() {
 9          if (pending != null) for (Component c : pending) wire(c);
10          pending = null;
11      }
12      void wire(Component c) {
13          c.createAdaptee((Composite) getAdaptee());
14          if (c instanceof Container) ((Container) c).createPending();
15      }
16      public Component add(Component c) {
17          if(getAdaptee() == null) pending.add(c);
18          else wire(c);
19          return c;
20      }
21      ...
22  }
23  public class JButton extends AbstractButton {
24      String text;
25      void createAdaptee(Composite parent) {
26          Button button = new Button(parent, SWT.PUSH);
27          button.setText(text);
28          setAdaptee(button); ...
29      }
30      public void setText(String text) {
31          if(getAdaptee() == null) this.text = text;
32          else getAdaptee().setText(text);
33      }
34      ...
35  }
```

Figure 3.3: *Delayed Instantiation* example.

without delegation. We call re-implemented operations *safe*, because they can be executed by client applications prior to adaptee instantiation. For example, `JButton` stores interim data in a field `text` (line 24) and re-implements its accessor methods (line 30). The factory method then creates the adaptee (line 25) and ensures interim data is delegated (line 27).

Although `Container.add` sets the parent-child relationship, it cannot trigger adaptee instantiation if the container itself does not have an adaptee—if the adaptee is available, the objects are wired (line 18), otherwise, the object is added to a list of child components `pending` adaptee instantiation (line 17). Eventually a component will be added to a top-level widget, which always has an adaptee; this is when `wire` (line 12) will be executed. This will trigger the factory method on the component, passing the parent widget's adaptee as parameter (line 13). If the component is a container it may have pending children, so `createPending` is called (line 14) to iterate the list and cascade wiring (line 9) of children.

Implementations of delayed instantiation must decide how to store interim data and which operations to re-implement. In SwingWT, interim data is stored directly in surrogate fields, as in our example. Surrogate methods must always check if the adaptee has already been instantiated to decide whether the operation should be delegated or simulated with the placeholder fields (such as `setText` in line 30).

Another option is to use the layered adapter pattern. An internal adapter variant implements a placeholder and the surrogate swaps it to a final variant once the parent widget is known. This solution has the advantage that surrogates delegate to the uniform internal adapter interface without having to check if the adaptee has already been instantiated. In this case, factory methods can be implemented in both layers: the surrogates (to create internal adapters) and the adapters (to create adaptees).

Factory methods in surrogates also prevent object adapters from instantiating multiple adaptees. In Java, every constructor must call a constructor of the super-class (except `Object`). If adaptees are instantiated in all constructors, super-classes will always instantiate an adaptee, which is then stored in a field (as in line 2). Sub-classes can only refine the adaptee by overwriting the field after the fact, causing multiple adaptees to be instantiated when only the refined one would suffice. Factory methods allow smoother refinement since sub-classes can override the method to prevent instantiation on super-classes, even if the factory method is called by constructors.

Delayed instantiation should be used with care because reproducing all operations of a surrogate ultimately violates the intention of reuse. Only simple operations that are often performed in client applications before the trigger should be allowed to be delayed. Wrappers should also provide documentation regarding the operations that are safe. In SwingWT, for example, mostly getters and setters are considered safe.

3.4 Inverse Delegation

This pattern deals with control flowing from the replacement API to the application. In particular, *Inverse Delegation* allows application callbacks designed to work with the original API to be executed when corresponding events occur in the replacement API.

There are two main approaches to implement the pattern. First, it is possible to apply the *ADAPTER* pattern in the inverse direction, as outlined in Figure 3.4. When `MyActionListener` is registered to a surrogate (such as `JButton`), the surrogate creates an

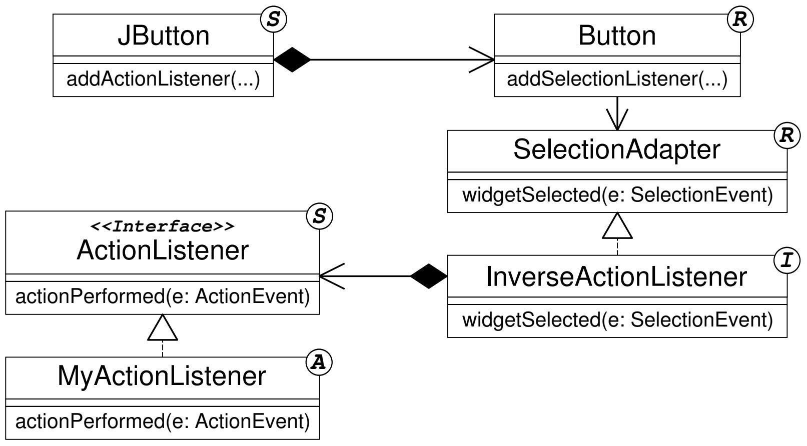
JButton
S
addActionListener(...)
Button
R
addSelectionListener(...)
SelectionAdapter
R
widgetSelected(e: SelectionEvent)
<<Interface>>
ActionListener
S
actionPerformed(e: ActionEvent)
InverseActionListener
I
widgetSelected(e: SelectionEvent)
MyActionListener
A
actionPerformed(e: ActionEvent)

Figure 3.4: *Inverse Delegation* for callbacks.

equivalent listener of the replacement API (`InverseActionListener`), which becomes an inverse adapter to the original listener. When the replacement API generates an event in `Button`, `InverseActionListener` translates and delegates it back to its `ActionListener` inverse adaptee.

Another approach, implemented in SwingWT and SWTSwing, is to use the surrogate itself as inverse adapter. In Figure 3.4, `JButton` would implement `SelectionListener` (the interface equivalent to `SelectionAdapter`) and register itself to `Button`. `JButton` would keep a list of `ActionListeners`. When `MyActionListener` is registered, `JButton` simply adds it to the list. An event in `Button` would then trigger `JButton`, which would then translate the event and dispatch to the listeners in its list, including `MyActionListener`.

The advantage of the approach found in the GUI wrappers is simplicity: the role of `InverseActionListener` is played by `JButton` without an extra class. The main disadvantages are that surrogates implement an additional concern, and that the inheritance hierarchies of surrogates and replacement API get tangled.

3.5 Wrapping Identity Map

The semantics of some original API operations, such as the `elements` method in Figure 2.1 and the event objects in Figure 2.5, is to return a new object at each call. In this case, a surrogate can simply wrap replacement API objects into new surrogates before returning to the application. For other operations it is necessary to maintain the correspondence

```
 1   public abstract class Component {
 2     static Map<Widget, Component> map = new ...;
 3     static Component getSurrogate(Widget adaptee, Class<?> sClass) {
 4         if (! map.containsKey(adaptee))
 5             map.put(adaptee, createSurrogate(adaptee, sClass));
 6         return map.get(adaptee);
 7     }
 8     ...
 9   }
10   public class Container extends Component {
11     public Component add(Component c) { ... }
12     public Component[] getComponents() {
13         Control[] controls = ((Composite) getAdptee()).getChildren();
14         Component[] components = new Component[controls.length];
15         for(int i = 0; i < controls.length; i++)
16             components[i] = getSurrogate(controls[i], Component.class);
17         return components;
18       }
19       ...
20   }
```

Figure 3.5: *Wrapping Identity Map* example.

between adaptees and surrogates to avoid object schizophrenia (see Section 2.3.4).

In Figure 3.5, for example, `Container.getComponents` (line 12) delegates the call to its adaptee (line 13) and gets an array of sub-widgets. It then iterates over the array wrapping the replacement API objects into corresponding surrogates (line 16). Instead of creating a new surrogate at each call, `Component.getSurrogate` (line 3) consults a *Wrapping Identity Map* (line 2) to get the `Component` surrogate already mapped to the `Widget` adaptee. If the adaptee does not have a surrogate yet, the component creates a new surrogate on the fly (line 5).

Note that `createSurrogate` (code omitted) receives as parameter the surrogate class to be instantiated. This allows the method to either instantiate the surrogate object reflectively—in which case surrogates must conform to certain rules, such as having a default constructor and setters for adaptees—or implement a hard-coded type mapping. Furthermore, because the operation requests a surrogate of a certain type (`Composite` in line 16) it is possible to have an adaptee mapped to different surrogates depending on the context, which is necessary when many surrogates share a single adaptee.

The identity map is only required if the surrogate is an object adapter because class adapters are already typed by both APIs and do not require translation. Maps can also be designed on a per-type or system-wide basis; they can be manually implemented using available map data structures, or a library can be designed to maintain the map and instantiate surrogates reflectively if necessary. An additional issue with identity maps is memory leak: correspondences should be weak. That is, garbage collection should not be prevented from cleaning up objects only because they are in the map.

3.6 Summary

In this chapter, we have presented an initial catalog of API wrapping design patterns. The patterns have been abstracted from the idiomatic solutions used by developers in practical wrappers. The catalog addresses the lack of guidance to the specification of wrappers. In the next chapter, we introduce the concept of compliance testing, which addresses two additional issues in API migration: discovery of behavioral differences across APIs and assessment of migration correctness.

Chapter 4

Compliance Testing

In this chapter, we introduce the notion of *compliance testing for API migration*. Compliance testing addresses many of the issues found in practical API migration that we described in Chapter 2. In particular, compliance testing:

- supports developers in uncovering behavioral differences between a wrapper and its original API, focusing on behavior exercised by the application under migration; and
- enables the assessment of wrapper correctness, making explicit the notion of a "good enough" wrapper that is informal in practice.

This chapter is organized as follows. First, we motivate compliance testing by discussing the unique challenges of testing for API migration. We argue that existing techniques fail to address these challenges adequately. Next, we introduce a running example to illustrate how compliance testing can uncover compliance issues between a wrapper and its original API. Finally, we detail the components of compliance testing: API interaction trace, trace interpretation, API contracts, and assertion tuning.

4.1 Unique Challenges of Testing for Wrapper-based API Migration

Testing for wrapper-based API migration involves unique challenges. Compliance testing is a form of automated testing motivated by the inadequacies of existing testing techniques in addressing these challenges comprehensively. In this section, we discuss this unique context and the shortcomings of existing testing techniques. Furthermore, we indicate how compliance testing addresses these challenges. It does so partly by leveraging aspects of existing techniques.

4.1.1 Focus on API Behavior Relevant to the Application

API migration usually targets a specific application (Chapter 2). Therefore, testing must concentrate on the behavior of the original API that is exercised by this application. Traditional testing techniques, however, concentrate on finding most faults, usually by achieving high code coverage [81]. API-specific test suites also focus on exploring most possible usage scenarios. These techniques are of limited use for API migration because they may demand too much of the wrapper—behavior that is not needed by the application under migration—and may still fail to enforce behavior relevant to the application.

Compliance testing focuses on the behavior that is important to the application. To this end, it uses a record and replay technique inspired by approaches for regression test extraction [64, 75, 54, 30]. The idea is to record application execution traces with respect to a certain module—in our case an API—and then replay the execution after the module evolves. The replay tool mocks the application behavior and checks whether the module behaves as when recorded. While regression testing focuses on preserving behavior of an earlier version in a later version of a module, compliance testing focuses on preserving the recorded behavior of the original API in the wrapper.

4.1.2 Focus on API Interaction

Testing for wrapper-based API migration must focus on assessment of API behavior. Application test suites, however, do not necessarily target the interaction of the application with the API. While the ultimate goal may be to satisfy application tests, such tests may be of little use to assess wrapper compliance and to drive wrapper development.

Compliance testing focuses on the API interaction. The record and replay technique captures only the interaction of the application with the API. Still, the technique requires application usage scenarios. However, developers can reuse existing application test cases or rely on some of the many automatic test generation techniques (such as [12, 20, 66, 3]) to generate new application test cases. Then, compliance testing enhances application test cases with assessment of API-specific behavior.

4.1.3 Test Oracle Selection and Limited State Observation

Test oracles [81] determine the correctness of a wrapper. In API migration, the underlying assumption is that the wrapper should strive to reproduce the behavior of the original API triggered by the application. Thus, baseline oracles must enforce in the wrapper the behavior observed in the original API.

Compliance testing uses API contracts to define oracles (see Section 4.5). We use the term contract to suggest that the original API has a contract with the application to behave in a certain way and the wrapper should honor this contract. Here we discuss the main categories of oracles relevant to API migration. Compliance testing, however, can be extended with new oracles via implementation of new API contracts.

Many existing approaches provide test oracles for input/output behavior [75, 64, 69, 38, 54, 93, 41, 85, 66, 89]. They observe the result of functions for a given input, and then generate assertions that enforce the output behavior. Exceptional behavior is often handled as a possible output.

We saw in Section 3.4 that APIs often delegate control back to applications via callbacks. Hence, it is important to verify that the wrapper reproduces the callbacks of the original API that are triggered by the application. Some approaches provide oracles for synchronous callbacks [75, 64, 38, 54], when the callback is executed in the same thread as the triggering call. While approaches to deterministically replay multi-threaded applications exist [39, 74], they are tailored to find concurrency bugs instead of verifying that asynchronous callbacks are executed. Instead, we support a heuristic approach.

The behavior of a module can also be observed by changes to its state. Orstra [89], for example, is a tool for augmenting generated tests with regression oracle testing. It includes a concrete-state contract that observes the state of a module based on its fields. In wrapper-based API migration, however, concrete state observation is impossible because surrogates usually do not contain the same fields as the classes they are replacing. The form of state observation that is most relevant to wrappers, therefore, is interface-based.

4.1.4 Relaxed Behavioral Equivalence

The studies and interviews of Chapter 2 have shown that developers are often content with a wrapper that is "good enough". In most cases, strict behavioral equivalence between wrapper and original API is desired. But in some cases, the wrapper is allowed to produce slightly diverging behaviors while still being considered correct. This may occur, for example, because some mismatch is too difficult to be neutralized by the wrapper, some output or state values should be interpreted differently, some behavior is non-deterministic, or the difference is not important to developers. Current testing techniques, however, focus only on strict behavioral equivalence.

In terms of the framework proposed by Staats et al. [81], the problem is that, in the context of API migration, the oracles described above are not complete. In other words, a deviation detected by an oracle does not imply necessarily that the wrapper is incorrect with respect to the *specification*, which is the idealized behavior expected by developers.

Compliance testing introduces the notion of assertion tunings to overcome this problem (see Section 4.6). Tunings are executable specifications of acceptable behavioral differences. For example, a developer of a GUI wrapper may accept that widget positions diverge by 10% of the original values. They are executable in that they modify the default semantics of API contracts. Tunings capture and document the expectations of wrapper developers, thereby making explicit the notion of a "good enough" wrapper.

4.2 Motivating Example

In this section, we illustrate compliance issues between a wrapper and its original API. As previously discussed, the aim of wrapper development is to reduce such issues until the wrapper is "good enough" for use in applications. Original and replacement APIs may contain arbitrary differences (Chapter 2); the wrapper must neutralize these differences.

Our example concerns SwingWT, the wrapper between Swing and SWT presented in Section 2.3. Using compliance testing and the *Koloo* method—and associated toolkit—presented in Chapter 5, we developed a revision of SwingWT, which we call *evolved SwingWT* in the sequel. The evolution of SwingWT is detailed in Section 6.1.

Figure 4.1 illustrates compliance issues of SwingWT versus Swing; it also illustrates how our systematically evolved wrapper reduces the compliance issues. In the figure, we exercise one scenario of the *Tooltip* demo of SwingSet2[1], which is a set of Swing demos originally distributed by Sun. The scenario displays a specific tooltip as it is triggered when the user positions the mouse over the cow's mouth. We chose this scenario because the *Tooltip* demo is clearly concerned with triggering tooltips.

On the left, the reference Swing behavior is that the cow moos. In the middle, the original SwingWT wrapper, prior to our efforts, is used. Three problems are noticeable in the image: the background color is gray instead of white, the tooltip shows a different

[1]Source code can be found in Sun JDK folder `demo/jfc/SwingSet2/`

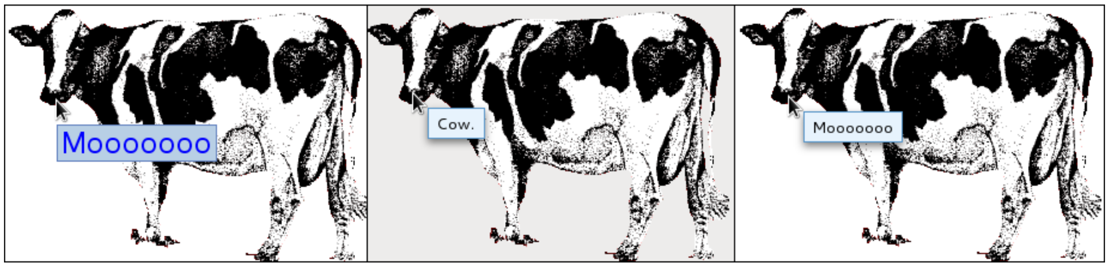

Figure 4.1: SwingSet2's Tooltip demo in Swing, SwingWT, and evolved SwingWT.

message, "Cow." instead of "Mooooooo", and the tooltip font is different. On the right, the evolved SwingWT wrapper is used. The background color and the tooltip message now comply with the original API; the font is still different, however, which demonstrates a limitation of our approach. The example is visually striking, but compliance testing also reveals issues that are not easily spotted by looking at particular screenshots; it also applies to domains other than GUI programming.

Compliance testing reveals behavioral differences automatically. The behavior of the original API represents a baseline behavior that is checked via assertions generated from API contracts. Revealed differences are referred to as *violations* of compliance. Compliance testing can be exercised with varying levels of scrutiny in terms of API contracts to be enforced. In the following we briefly discuss how different contracts implemented in *Koloo* tools helped us evolve SwingWT.

The API contract for return values detects a violation: SwingWT returns a different boolean value than Swing in a call to the method `contains(Point)` of class `java.awt.Polygon`. An investigation reveals that the *Tooltip* demo uses the method to map areas of the image to corresponding messages. The violation is directly linked to the observable fact that a wrong message is displayed in the middle of Figure 4.1.

The API contract for asynchronous calls from the original API to the application also detects a violation: SwingWT does not reproduce an asynchronous callback to the method `contains(int, int)` of class `java.awt.Component`. An investigation reveals that the *Tooltip* demo overrides the method to select the appropriate tooltip to display. The violation is also directly linked to the observable fact that a wrong message is displayed.

The contract for state integrity also flags a violation for diverging background colors. This violation cannot be revealed by looking at return values because the application does not exercise the getter of the background color in any way. We note that additional scrutiny in terms of API contracts may also imply noise in reporting violations (see Section 6.1);

$$
\begin{array}{lrcl}
\textit{event} & E & ::= & (e_{id},\, t_{id},\, p_{id},\, d,\, T,\, sig,\, P[\,],\, R) \\
\textit{execution id} & e_{id} & \in & \mathbb{N} \\
\textit{thread id} & t_{id} & \in & \mathbb{N} \\
\textit{parent id} & p_{id} & \in & \{e_{id}\} \\
\textit{direction} & d & \in & \rightarrow \mid \hookleftarrow \\
\textit{event entity} & T,P,R & ::= & (type,\, content) \\
\textit{entity content} & content & ::= & \_ \mid value \mid id \mid (id,\, entity[\,]) \\
\textit{signature} & sig & \in & string \\
\textit{type name} & type & \in & string \\
\textit{object id} & id & \in & \mathbb{N} \\
\textit{content value} & value & \in & string \mid primitive
\end{array}
$$

Figure 4.2: Event structure of captured traces.

elimination of these violations is not necessarily expected by a "good enough" wrapper.

Finally, a difference in tooltip fonts remains. This issue occurs because Swing handles HTML whereas SWT does not support HTML in tooltips. This API difference eventually led to an observable GUI difference that was manually resolved during wrapper development by stripping HTML off the tooltip. Although an API contract could conceivably observe and enforce this type of side-effect, such a contract is not implemented in *Koloo*.

In the next sections, we detail the components of compliance testing that allow developers to detect such behavioral differences, automate their enforcement, and make decisions regarding their importance to the application at hand.

4.3 API Interaction Trace

The execution of a compliance test is captured as an *API interaction trace*. The trace represents the interaction between the application under migration and the original API as recorded during an application run. It is a sequence of events that denote method calls, constructor calls and field accesses to *API-defined* elements. An element is API-defined if it is declared by the API or it is an application extension of an element declared by the API. The trace is meant to abstract from execution events that do not involve API types or subtypes thereof in the application. A compliance test, thus, consists of the re-execution of trace events against the wrapper, together with an assessment of its behavior.

Figure 4.2 defines the structure of events more precisely. An event is an eight-tuple uniquely identified by its execution id, which captures the order in which events were

```
e_1, t_0   → _:Robot.<init>():0 ...
e_16, t_0  → _:ToolTipDemo.<init>(_):1 ...
e_22, t_0     → _:JPanel.<init>():2 ...
e_29, t_0     → _:ToolTipDemo$Cow.<init>(1):3 ...
e_32, t_0     → _:Polygon.<init>():4 ...
e_62, t_0     → 3:ToolTipDemo$Cow.setToolTipText("Cow"):_ ...
e_66, t_0     → 2:JPanel.add(3):3 ...
e_68, t_0  → _:JFrame.<init>("ToolTip Demo"):5 ...
e_72, t_0  → 5:JFrame.getContentPane():6
e_73, t_0  → 6:Container.add(2, "Center"):_ ...
e_77, t_0  → 5:JFrame.show():_ ...
e_85, t_0  → 0:Robot.mouseMove('342', '312'):_
e_86, t_0  → _:Thread.sleep('3000'):_
e_87, t_1  ↩ 6:Container.contains('155', '112'):'true'
e_88, t_1     → _:Point.<init>('155', '112'):7
e_89, t_1     → 4:Polygon.contains(7):'true'
e_90, t_1     → 3:ToolTipDemo$Cow.setToolTipText(
           "<html><center><font color=blue size=+2>
           Mooooooo
           </font></center></html>")):_
```

Figure 4.3: API interaction trace for the example in Figure 4.1.

detected. A thread id identifies the active thread. The parent id indicates the state of the execution stack: event e_{id} occurred in the control flow of its parent event p_{id}. The event signature identifies the replacement API element, including the static target type. From the viewpoint of the application an event is either outgoing (→), if it originates from an application type, or otherwise incoming (↩). Incoming events express callbacks, when the API calls a method implemented by the application. A callback is *synchronous*, if it is the response to a call in the same thread. It is asynchronous otherwise, that is, if it is a top-level event of a thread that is controlled by the API. In this case, the trace contains only callbacks at the top-level, meaning that the API created the thread.

The remaining items correspond to the target object, a sequence of parameters, and a return object, all encoded as event entities. An entity represents a runtime value or object as a tuple with a string (the runtime type of the entity) and a content item. The content can be _ (null or void), a value (primitive or string), an object reference id, or the reference to an array (which has an id and points to a sequence of entities). Exceptions thrown by events are encoded by the return value being a reference with the exception type.

Figure 4.3 shows an excerpt of the trace underlying the motivating cow example of Section 4.2. This is the trace as recorded with the original Swing API; in Section 6.1

we describe the design of the application scenario that originated the trace. The trace is edited for readability; for instance, type names are abbreviated. No parent id is included because it is clear from indentation. Numbers represent object ids, single quotes represent primitive values, and double quotes represent string values.

The trace shows the construction of GUI elements (e_1-e_{73}), the method call to show the window (e_{77}), and a call to move the mouse over the cow (e_{85}). It also contains calls to the two different `contains` methods that we encountered during the discussion of behavioral differences in Section 4.2: the asynchronous callback (note the different thread ids) missing in SwingWT (e_{87}), and the computation of the area check (e_{89}). The last event assigns the specific tooltip's text using HTML format.

4.4 Trace Interpretation

A trace interpreter re-executes the API interaction trace. The interpreter mocks the application by reproducing its role in the interactions recorded in the trace. Since interpretation is borrowed from regression test extraction techniques [64, 75, 54, 30] we refrain from exhaustive details; instead, we give an overview of the process.

The interpreter builds a model of all types and methods from the trace so that mock application types and methods can be provided for trace re-execution. In particular, receiver objects of incoming events are associated with mock application types. Each mock application method is essentially a sequence of outgoing API interactions, such as method calls and field accesses. Events without parents are stored in a special `main` method. Our implementation in the *Koloo* toolkit uses a custom classloader that generates bytecode for the mock application types with method implementations that call back into the interpreter.

Interpretation starts with the `main` method and proceeds sequentially. Each outgoing event is executed in turn. The interpreter keeps a map from trace object id to runtime object so that it can determine the target and parameters of events; primitive values are used directly. After returning from an event execution, the interpreter registers the return value in the object map and uses the selected contracts to verify the result. Contracts can issue fatal violations that immediately interrupt interpretation or can register the violation but allow interpretation to continue. After the last event has been executed, the interpreter reports the set of detected violations.

4.5 API Contracts

Compliance testing uses API contracts to evaluate wrapper behavior during re-execution of an API interaction trace. As described earlier, API contracts are oracles that check whether the wrapper honors a behavioral contract established by the original API.

API contracts enrich the trace with assertions. Different API contracts may use different *oracle data* [81]. That is, the generated assertions can observe different aspects of the trace, and even data available during trace interpretation. Since developers can select which contracts to apply, simpler contracts can be used in initial stages of development, while more demanding ones are postponed.

The *Koloo* toolkit can be extended with new API contracts via implementation of a simple Java interface. In the following sections we detail the API contracts currently implemented in *Koloo* and used during its evaluation (Section 6.1).

4.5.1 Basic Contracts

Basic contracts represent the simplest level of compliance expected from wrappers. Their oracle data is information readily available in traces, such as data and control flow.

Value Equality — Checks the return value of a method call or field access. The primitive value or string returned by the wrapper must equal the value returned by the original API.

In the motivating example of Figure 4.1, this contract detected a violation in the method call to `Polygon.contains` (e_{89} in Figure 4.3). In this case, the original API returned `true`, but the wrapper returned `false` in the re-execution.

Reference Integrity — Checks the returned object reference of a method call or field access. The identity of the object returned by the wrapper must equal the identity observed with the original API, modulo consistent renaming of identities.

This contract can flag the need for the Wrapping Identity Map pattern (Section 3.5), and detect bugs in its implementation. The contract detects violations if the semantics of wrapper and original API diverge with respect to caching of objects to be returned. Note, however, that a violation is only detected if the application exercises the API in a susceptible way, for instance, by calling a certain method twice.

Value equality and reference integrity are also applied to arrays and their contents. The array must preserve reference integrity and its contents must all preserve value or reference integrity, depending on the element type. Arrays, thus, have list semantics.

Exception Conformance — Checks the exceptional behavior of method calls. If a method call throws an exception with the original API, it must also throw an exception of the same type during re-execution. Furthermore, re-execution must not throw any additional exceptions.

Callback Conformance — Checks callback triggering behavior. If a callback is triggered by the original API, it must also be triggered during re-execution.

To verify callback conformance, the event for triggering a callback must be associated with an actual callback. Invocation of callbacks is tracked globally so that they can be matched with triggering events.

In the case of synchronous callbacks, the association between callbacks and the preceding triggering event is unambiguous: the callback must be reproduced by its parent event. In the case of asynchronous callbacks, the triggering event could date back more arbitrarily. In fact, there is no guarantee that multi-threaded callbacks re-execute deterministically in the order of trace capture [33], and deviations from the captured order do not imply incorrect behavior. Assertions are checked conservatively. A check for re-execution of the callback is issued at the preceding event (which is not necessarily the trigger), and the check does not need to succeed immediately, but it is attempted repeatedly and in parallel to re-execution (in a separate, auxiliary thread), subject to a large timeout.

The callback conformance contract can indicate the need for the Inverse Delegation pattern (Section 3.4) and may detect bugs in its implementation. Missing callbacks indicate that inverse delegation may be missing in the wrapper. However, in this formulation, wrapper callbacks that were not called by the original API are not considered violations.

In our example, this contract detected a violation because the asynchronous callback to `Container.contains` (e_{87} in Figure 4.3) was not performed by the wrapper.

4.5.2 State Integrity

Value equality and reference integrity are applied only to data returned by the API. We can also verify wrapper behavior by observing additional state of objects in the trace. For instance, the diverging background color in our example (Figure 4.1) occurred because the

Modifier	Observer	Description
`set`X(... v)	`get`X()	getter returns v
`set`X(`boolean` v)	`is`X()	predicate returns v
`add`X(... v)	`get`X`s`()	result contains v
`add`(T v)	`get`T`s`()	result contains v
`put`(... k, ... v)	`get`(k)	getter returns v

Table 4.1: The modifier/observer pair heuristic for the set/get contract.

default background color of SwingWT was computed differently than in the original Swing. This behavioral difference was not directly exercised by the application.

The state integrity contract checks the state of receiver objects in the trace: targets of method calls and field accesses. State is collected by observer methods, since it must be interface-based, and compared by value equality and reference integrity contracts.

Koloo allows users to control the behavior of state collection. One supported approach is *selective observation*, in which users declare, in a point-wise manner, the methods to collect state. For instance, a declaration to observe `Container.getBackground.getRed` collects state at every event in which the receiver object is of type `Container`; the state is collected by calling `getBackground` on the receiver object and then `getRed` on the object returned by `getBackground`.

Another supported approach is to select all available observer methods in receiver objects. *Koloo* uses a trivial heuristic for detecting observer methods: they start with *get*, *has*, *is*, or *count*. In this case, users can also control the depth of collection, such that the results of observers are observed recursively. By default, however, collection is shallow.

4.5.3 Set/get Integrity

This contract checks the *set/get law*: a getter following a setter must retrieve the argument of the setter. More precisely, if a method for object modification is executed and there is an associated observer, then the observation must retrieve the argument of the modification.

The set/get integrity contract is an example of an extension that was added to *Koloo*. We implemented this contract because it is simpler than state integrity—it is subsumed by it—and yet may reveal important API behavior to be emulated by the wrapper.

This contract can detect behavioral differences in setters and getters. Such integrity, however, is only required for the re-execution of the trace if the original API respects the law. It may be reasonable that APIs do not satisfy "laws" as this. For instance, a setter may handle *nulls* or perform parameter normalization.

Koloo uses a heuristic to determine methods that are modifiers, as well as corresponding observers. Table 4.1 shows the considered pairs of modifiers and observers; we use X for variable postfixes of method names and T for type names. For each trace event that represents a call to a method that is considered a modifier by this table, the contract generates an assertion using the corresponding observer. For example, a call to a `setEnabled` method is enriched with an assertion that checks whether the value passed as parameter equals the one returned by a call to `isEnabled` on the same object.

4.6 Assertion Tuning

The major source of control in compliance testing is the selection of contracts for which assertions are to be checked, as we discussed above. A further source of control is the tuning of the semantics of assertions. For instance, value equality can be relaxed. Also, tunings can account for non-deterministic and non-repeatable behavior.

A tuning specification is a list of assertion tunings applied during trace re-execution. An assertion tuning overrides the semantics of selected assertions. Thus, an assertion tuning consists of a selection of events to be tuned, and a match operation that defines the modified semantics. The match operation compares expected results (captured with the original API) against actual results (recorded during re-execution).

Koloo implements some default match implementations and provides a framework for extensions. Technically, *Koloo* accepts tuning declarations of the form Package / Type.Method : MatchOperation Parameters. This coarse mechanism for event selection was sufficient in our experiments (Section 6.1); finer event selection mechanisms may be implemented if necessary.

Figure 4.4 demonstrates an example in more detail. The `BytecodeTuning` occurred in a wrapper study in the domain of bytecode engineering. The example is concerned with the comparison of the result returned by the central `getBytes` method of the API, which is supposed to return the bytecode of a given Java class. The following declaration applies a tuning to `getBytes` (no parameter is passed):

org/apache/bcel/classfile/JavaClass.getBytes : BytecodeTuning

The challenge is that the original API and the replacement API may use different orders for the constant pool that is part of the returned bytecode sequence. We need to compare

```
public class BytecodeTuning implements ... {
  public boolean matches(byte[] expected, byte[] actual) {
    return serialize(expected).equals(
            serialize(actual));
  }
  private static String serialize(byte[] bytecode)
    StringWriter w = new StringWriter();
    new ClassReader(bytecode).accept(
      new TraceClassVisitor(new PrintWriter(w)), 0);
    return w.toString();
  }
}
```

Figure 4.4: The match operation for the bytecode tuning.

bytecodes at a higher level of abstraction, where constant pool order is immaterial. By serializing the bytecode into strings the constants are essentially inlined and, thus, this difference is neutralized. The replacement API readily provides a visitor for serialization, which is accordingly used by the match operation in the figure.

Figure 4.5 shows additional examples of tunings that were used in our evaluation (see Section 6.1). These tunings were implemented as needed and the corresponding match operations were incorporated in *Koloo*.

4.7 Summary

In this chapter, we have presented the novel concept of compliance testing for API migration. We have discussed the challenges posed by the unique context of testing for API migration and argued that they are not adequately addressed by existing testing techniques. Compliance testing borrows aspects from many of these techniques and offers a unified, tailored solution. We have then detailed the components of compliance testing: API interaction trace, trace interpretation, API contracts, and assertion tuning.

In the next chapter, we propose a method called *Koloo*. In essence, the method describes the practical steps needed to apply compliance testing and design patterns to API migration while fitting within the general process followed by developers in practice.

org/apache/bcel/generic/InstructionHandle.getTargeters : ArrayIsSet

Compliance of bytecode engineering APIs: a specific method's result of an array type is declared to have set semantics instead of the default list semantics.

java/awt/Component.getHeight : Percentage 10

Compliance of GUI APIs: a percentage-based divergence between expected and actual height of a component is admitted.

java/awt/Point.x : Margin 15

Compliance of GUI APIs: a margin-based divergence between expected and actual value of a point's coordinate is admitted.

javax/swing/JFrame.pack : SkipCallbacks

Compliance of GUI APIs: a tuning that turns off checking of callbacks that are triggered by calling `pack`.

Figure 4.5: Diverse assertion tuning examples.

Chapter 5

The *Koloo* Method for Wrapper-based API Migration

In the previous two chapters, we presented API migration design patterns and the concept of compliance testing for API migration. Together, these approaches address the three key issues identified in Chapter 2, namely: how to design wrappers, how to discover behavioral differences across APIs to be neutralized, and how to assess migration correctness.

In this chapter, we propose a method that integrates these approaches. The *Koloo* method is designed to evolve wrappers with respect to an application under migration. It can be used to create new wrappers or to evolve existing wrappers. The input is an application, a pair of APIs (original and replacement APIs), and optionally an existing wrapper; the output is a wrapper that is demonstrably "good enough" for the application.

Koloo addresses key issues without being intrusive. In particular, *Koloo*:

- fits within the iterative, sample driven general API migration process usually followed by developers in practice (Chapter 2);
- suggests the use of API migration design patterns (Chapter 3) to support the design and implementation of wrappers;
- prescribes practical steps to use compliance testing (Chapter 4) as a means to elicit the requirements for the API migration as well as assessment of its correctness; and
- is supported by a toolkit that automates many of its tasks.

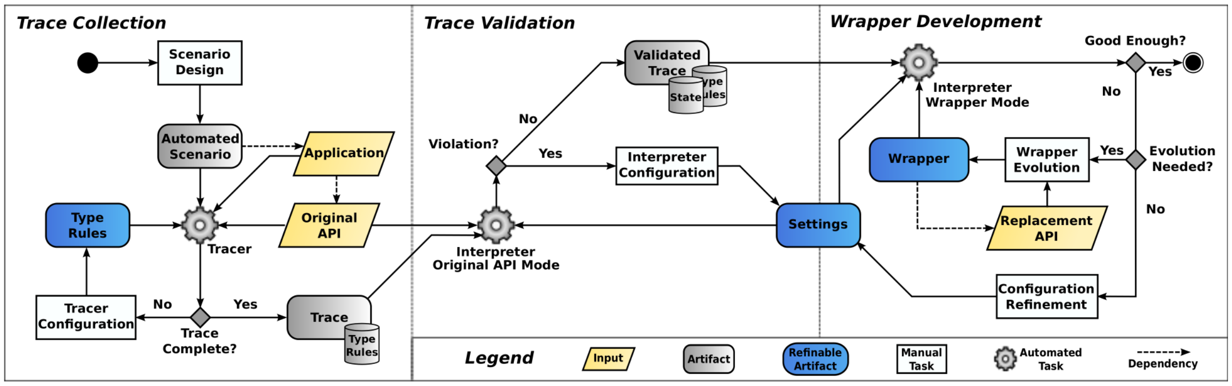

Figure 5.1: An overview of an iteration according to the *Koloo* method.

This chapter is organized as follows. First, we present an overview of the method. *Koloo* is iterative and each iteration is composed of three phases. Next, we present each phase: trace collection, trace validation, and wrapper development.

5.1 Overview

Figure 5.1 shows an overview of a *Koloo* iteration. Each iteration consists of three phases that describe how to apply compliance testing in practice. First, in *trace collection*, developers design an application scenario and extract an API interaction trace (Section 4.3). Then, in *trace validation*, an interpreter (Section 4.4) validates the re-execution of the trace, still using the original API. Validation supports detection of unreproducible and non-deterministic behavior. In this phase, settings are aggregated to decide on API contracts (Section 4.5) to be asserted. Finally, in *wrapper development*, the interpreter re-executes the trace with the wrapper. In this phase, the final decisions on applicable API contracts and corresponding tunings (Section 4.6) are made. Violation of contracts guide the evolution of the wrapper, possibly using the design patterns presented in Chapter 3.

Koloo defines the overall structure of an iteration, but admits adjustments. For instance, initially a migration effort may concentrate on the design of scenarios that cover the most important use cases of the application. Multiple traces could, thus, be collected and validated at once, prior to wrapper development. Most importantly, by the end of an iteration, the wrapper is demonstrably compliant with the original API in terms of the exercised scenarios, and the specification of contracts and tunings precisely documents the unresolved differences between wrapper and original API on the tested executions.

The *Koloo* toolkit implements support for the tasks defined as automated in Figure 5.1. *Koloo*-specific manual efforts concern identification of API types for configuration, selection of contracts to be checked, definition of tunings, and decisions on the relevance of violations. As reported by developers (Section 2.4), scenario design is also common and not specific to *Koloo*. Generally, the cost of API migration is dominated by the actual wrapper refinement, which is also not specific to *Koloo*.

5.2 Phase: Trace Collection

5.2.1 Scenario Design

Trace collection starts with the design of scenarios that will be automated as tests. A scenario is an execution of the application using the original API. It is important to design scenarios that cover the desired uses of the application because they represent the requirements for the migration: the wrapper will be considered compliant when it properly replaces the original API in these scenarios with respect to the verified contracts.

Scenarios usually target the applications at hand, as opposed to directly using the API. Tests for an application, if available, can be used as scenarios. Traditional test generation techniques, such as [12, 20, 66, 3], can be used to generate application test cases to be used as scenarios. Automatically generated or manually developed tests usually contain assertions related to application logic, which might also indirectly check some aspects of API usage. *Koloo*'s interpreter enriches application scenarios with assertions for selected API contracts.

Traces that are derived from scenarios must not depend on external actions and all results should be reproducible; this is a common requirement for automated testing. In the context of API migration, however, this requirement may imply extra challenges. Consider, for example, API migration for GUI APIs: GUI actions would need to be automated by using a GUI record and replay tool, such as GUITAR [59], or by manually writing scripts that automate actions upon the GUI. In the example of Figure 4.1, we designed a scenario that positions the mouse over various areas of the image, and automated the scenario with AWT's `Robot` class and JUnit.

It is the responsibility of the developer to design scenarios that explore interesting interactions with the API. Depending on the selected contracts, those interactions are more or less strongly verified. By using basic contracts (Section 4.5.1), verification is passive: assertions only check the data and control flow ocurring at trace re-execution.

Scrutiny of verification is increased when more active contracts are selected, such as set/get (Section 4.5.3), and the various levels of state integrity (Section 4.5.2). However, in our evaluation (Section 6.1), we have found that actively collecting state can be problematic because it demands more of the wrapper than the application actually needs.

5.2.2 API Tracing

A tracer executes the scenarios and collects API interactions as traces. *Koloo* uses the bytecode engineering framework ASM[1] for tracing. To this end, the tracer is also configured with a description of the types that belong to the application and the API; this metadata is also incorporated into traces. Trace collection should abstract away details of the underlying application that are not important from an API compliance testing point of view. This also helps with automating the scenario and making it reproducible. Such abstraction is similar to other approaches for extracting regressions tests from application traces [30, 64, 75, 69].

5.2.3 Type Rules

Developers need to identify application and API types. Such identification boils down to *type rules* such as regular expressions over their fully qualified names. For instance, `java.awt.*`, `javax.swing.*` and `javax.accessibility.*` identify Swing/AWT types.

The interaction between application and API may also be mediated by objects of *auxiliary types*: objects that are essential for the scenario even though their types should not be migrated. For example, APIs often use collections, iterators, and files. In order to reproduce the scenario in subsequent phases, events for such auxiliary types must be incorporated into traces. Thus, developers must also identify auxiliary types.

The tracer reports unbound object references in API events, which detect missing auxiliary types. Developers need to add type rules for auxiliary types until traces can be constructed completely. In the study of Section 6.1, however, few rules were needed: one scenario needed eight auxiliary types but most did not need any. *Koloo* defines Java's `Thread` and `Runnable` types to be auxiliary types by default, thereby enabling it to mock multi-threaded applications.

[1]`http://asm.ow2.org`

Additional type rules may be needed if wrapper and original API are implemented in different namespaces. These rules rewrite package names. For instance, the SwingWT wrapper presented in Section 2.3 requires the type rule `java.awt` $\Rightarrow$ `swingwt.awt`.

5.3 Phase: Trace Validation

Traces must be validated before they are attempted with the emerging wrapper. By validation we mean that interpretation of the trace is checked to reproduce the captured behavior. We also include the effort for contract selection, including the corresponding aggregation of state that is needed by some contracts but not readily available from the trace. In particular, state collection is needed for checking state and set/get integrity.

5.3.1 Reproducibility

Validation of reproducibility for the original API assures interpretation does not attempt to verify in the wrapper behavior that is even unreproducible with the original API. For instance, if the trace contains a call to a `time` method, then the corresponding event will not be validated and will be accordingly omitted from the verification of the wrapper. Such omission of events that fail to be reproducible also decreases the chance of attempting to verify non-deterministic behavior common in multi-threaded APIs. The reproducibility can depend on the selected contracts. For instance, the broad use of contracts for state integrity may potentially invalidate objects from the trace, thereby providing feedback to developers so that they backtrack on their selection.

5.4 Phase: Wrapper Development

Trace collection and validation permits actual development (evolution) of the wrapper. The trace validation interpreter is used to verify the traces for the wrapper. Type rules for namespace rewriting may now apply, and aggregated state from trace validation can be used as well.

If interpretation does not result in a violation, the validated trace is deemed *asserted*, meaning that the wrapper is known to comply with the corresponding scenario, modulo tunings, with respect to the verified contracts. If desired, the level of scrutiny can be increased by selecting stronger contracts and repeating the process. The set of asserted traces constitutes a regression suite that should be used any time the wrapper evolves, thereby guaranteeing that no regression goes undetected. *Koloo* contains a corresponding regression running tool to automate this part of the process.

5.4.1 Wrapper Scaffolding

The wrapper should provide all types, methods, and fields of its original API (see Section 2.3.4). We have observed three different approaches to implementing methods on the initial wrapper scaffolding. In xom2jdom (Section 2.2), we used exceptions to signal unimplemented methods, the so-called *exception throwing* approach. SwingWT uses *best effort* in that it leaves void methods empty and returns bogus values when necessary in hope that it is "good enough". Finally, from our interviews (Section 2.4) we know that SWTSwing uses a *mixed* approach: best effort, but logging usage of unimplemented methods.

The advantage of exception throwing is explicitness about the reason for method invocation failure: the method is unimplemented. Best effort may be "good enough" in some cases, but is not recommended because the reason for failure is lost. The mixed approach is recommended for production wrappers since they can be exercised in ways not expected by the designed scenarios. Nevertheless, during development, exception throwing fits better with compliance testing since exercised unimplemented methods raise explicit violations.

5.4.2 Wrapper Evolution

Any given violation can imply that developers need to evolve the wrapper. Developers should use the catalog of design patterns presented in Chapter 3 to solve common issues detected by assertions. For example, violations of reference integrity may indicate the need for a wrapper identity map (Section 3.5). Developers can also use additional sources of knowledge to support wrapper evolution, such as structural models of APIs, databases with information about API usage in applications, and API ontologies [11].

Violations can also be addressed by tunings. Developers may decide to add a tuning declaration to relax the API contracts for certain assertions (see Section 4.6). Alternatively, developers may decide to reconfigure the settings to reduce scrutiny by selecting fewer or less demanding API contracts.

By the end of an iteration the wrapper demonstrably complies with the original API in the asserted scenarios. The specification of contracts in use documents the level of scrutiny while the specification of tunings precisely capture the unresolved differences between wrapper and original API.

5.5 Summary

In this chapter, we have presented the *Koloo* method for API migration. It addresses the issues identified in Chapter 2 and fits within the general process followed by developers in practice. Furthermore, it guides developers in the steps needed to apply compliance testing and API wrapping design patterns in practice. In the next chapter, we describe an empirical evaluation of the method and validation of the API wrapping design patterns.

Chapter 6

Evaluation

In previous chapters, we introduced the *Koloo* method for API migration. Chapter 2 motivated our work with two studies and a set of interviews. In Chapter 3, we presented a catalog of design patterns that encode solutions to commonly occurring API wrapping problems. In Chapter 4, we introduced the concept of compliance testing to uncover API differences and assess wrapper correctness. Together, API wrapping design patterns and compliance testing are the foundation of the *Koloo* method, discussed in Chapter 5.

In this chapter, we consider evaluation. First, we present an empirical study in which we compared the *Koloo* method against three alternative methods, and assessed the usefulness of API contracts and assertion tuning. Next, we describe validation of the API wrapping design patterns by identifying their usage in practical wrappers.

6.1 Method Evaluation

In this section, we present an empirical evaluation of the *Koloo* method. We compare the method against alternative methods and evaluate aspects of compliance testing included in *Koloo*. We start with our research questions and follow with descriptions of subjects, methodology, and results. Then, we discuss how the results elucidate the research questions and consider threats to validity.

6.1.1 Research Questions

The overarching goal of this study was to evaluate the *Koloo* method in general and compliance testing in particular. Compliance testing addresses two of the issues identified in Chapter 2: uncovering of behavioral API mismatches and assessment of wrapper correctness. The study evaluates to what extent compliance testing achieves these goals.

Our first research question is the following:

> ***RQ1*** — How does the *Koloo* method compare with alternatives in terms of the ability to produce a wrapper behaviorally compliant with the application under migration?

This question refers to the ability to uncover behavioral API mismatches. Our hypothesis, which is part of the motivation for compliance testing (see Section 4.1), is that *Koloo* is better than:

- API test suites in uncovering the specific behavioral differences between wrapper and original API that are needed by the application under migration;
- application test suites in consistently focusing assertions on the API interaction; and
- application crashes in detecting important compliance issues.

Therefore, we compare *Koloo* against methods that represent these categories and break down *RQ1* into specific questions for each compared alternative:

> ***RQ1.1*** — Can *Koloo* uncover behavioral differences between original API and wrapper that are not uncovered by the original API test suite?
>
> ***RQ1.2*** — Can *Koloo* compliance tests detect more compliance issues than application-specific test suites? In other words, can *Koloo* enrich application test suites with useful API-specific assertions?
>
> ***RQ1.3*** — Can *Koloo* detect important compliance issues in wrappers developed with a traditional crash-driven method?

Note that *RQ1.1* addresses the failure of API test suites in detecting API mismatches that are relevant to the application. The study presented in Section 2.2 already provides evidence in support that API test suites may demand too much of the wrapper.

The following research questions relate to the ability to assess wrapper correctness, while also driving the development. In other words, they address the usefulness of API contracts and assertion tunings.

RQ2 — How do API contracts help in driving the evolution of wrappers in practice?

RQ3 — Which types of assertion tunings may be necessary? How often are they used?

6.1.2 Subjects

Table 6.1 presents general information for the applications, wrappers, and APIs that were subjects of our study; APIs are highlighted with a gray background. The data correspond to artifacts prior to the evolution of the wrappers. We chose representatives of three domains and covered the wide range of development methods for comparison.

For the XML domain we used the xom2jdom wrapper developed in the study presented in Section 2.2. We described the methodology used to develop xom2jdom in Section 2.2.3: it was driven by XOM's test suite. Thus, xom2jdom is a representative of an *API test suite-driven method*. For the application under migration we selected Memoranda[1], a scheduling tool that uses XOM to store data in XML files. We selected Memoranda because it provides a GUI, making it suitable to design scenarios without looking at source code.

We selected Quilt[2] as the application for the Bytecode domain. Quilt relies on the BCEL[3] library to modify the bytecode of applications to perform code coverage analysis. We selected Quilt because it contains a large test suite, allowing the comparison against an *application test suite-driven method*. During the study we developed the bcel2asm wrapper from scratch to migrate Quilt from BCEL to ASM.

Finally, we used the SwingWT wrapper for the GUI domain. As discussed in Section 2.4.4, SwingWT was developed in the traditional *application crash-driven method* and offered the opportunity of consulting its developers for validation of our results. The application under migration was SwingSet2, the set of Swing demos we discussed in Section 4.2. We chose SwingSet2 because SwingWT developers reported that SwingSet2 crashes initially drove the development of SwingWT.

6.1.3 Methodology

Using the *Koloo* method we evolved the two existing wrappers and developed the xom2jdom wrapper from scratch. For each wrapper, we followed the phases of the *Koloo* method: trace collection, trace validation, and wrapper development. We recorded the violations

[1]`http://memoranda.sf.net`
[2]`http://quilt.sf.net`
[3]`http://commons.apache.org/bcel/`

Artifact Kind	**Subject**$_{version}$ Type Definitions	**Types**	**Methods**	**NCLOC**
	XML			
Application	**Memoranda**$_{1.0-RC3.1}$ net.sf.memoranda.*	461	1,684	23,891
Original API	**XOM**$_{1.2.1}$ nu.xom.*	110	884	19,395
Wrapper	**xom2jdom** $_{0.1}$ nu.xom.*	43	602	5,203
Replacement API	**JDOM**$_{1.1}$ org.jdom.*	73	908	8,735
	Bytecode			
Application	**Quilt**$_{0.6-a-5}$ org.quilt.*	74	641	5,393
Original API	**BCEL**$_{5.2}$ org.apache.bcel.*	408	3,386	27,384
Wrapper	**bcel2asm** org.apache.bcel.*	-	-	-
Replacement API	**ASM**$_{3.3.1}$ org.objectweb.asm.*	174	1,526	23,348
	GUI			
Application	**SwingSet2**$_{1.4}$ swingset2.*	131	425	6,421
Original API	**Swing**$_{1.4}$ java.awt.* javax.swing.* javax.accessibility.*	2,032	18,702	234,731
Wrapper	**SwingWT**$_{0.90}$ swingwt.awt.* swingwtx.swing.* swingwtx.accessibility.*	864	6,537	33,582
Replacement API	**SWT**$_{3.3.2}$ org.eclipse.swt.*	696	9,607	99,205

Table 6.1: Metrics on study subjects prior to wrapper development.

detected by *Koloo* and the assertion tunings used in the development. This allowed us to compare *Koloo* against alternative methods and gather evidence with respect to compliance testing (*RQ2*) and assertion tunings (*RQ3*). In the following we detail wrapper-specific procedures.

xom2jdom

We designed a single scenario for Memoranda: we opened and closed a complex schedule document. We performed the scenario manually in the GUI while tracing the interaction between Memoranda and XOM.

Then, we validated the trace. We started with basic contracts and incrementally increased the scrutiny, first with set/get contracts and then with state integrity contracts. The set/get contract implemented in *Koloo* uses the modifier/observer pair heuristic (Table 4.1). The state contract uses by default the shallow observer heuristic (Section 4.5.2).

Finally, we evolved xom2jdom based on the detected violations. We started with basic contracts, and subsequently used contracts for set/get and state integrity. After the wrapper was compliant, we re-executed XOM's test suite. The goal was to check whether the suite was able to spot the same behavioral differences as *Koloo* (*RQ1.1*).

bcel2asm

We leveraged Quilt's test suite to create scenarios. We executed the 65 test cases in Quilt's suite and collected traces for the 33 that actually use BCEL. Next, we validated the traces with increasingly stronger contracts. Whenever necessary, we created tunings for the original API.

Then, we started development of bcel2asm from scratch. We started with an exception-throwing wrapper such that every method throws an `UnsupportedOperationException` (see Section 5.4 for scaffolding considerations). Quilt could already be compiled with this initial wrapper. Next, we developed the wrapper using one trace at a time, and only basic contracts. For each trace we executed the interpreter, picked a violation, and either evolved the wrapper or created a tuning. After each of these iterations we executed the suite of asserted traces to detect regressions. Furthermore, because each trace corresponds to the execution of a Quilt test case, we re-executed the original suite. This let us compare the contract checking of manually written test cases against *Koloo* (*RQ1.2*).

We mostly used basic contracts; sometimes, however, we resorted to selective state-based contracts to verify the state of specific types of objects. In two occasions, we decided

Subject$_{version}$	**Types**	**Δ**	**Methods**	**Δ**	**NCLOC**	**Δ**
xom2jdom $_{0.2}$	43	(0)	602	(0)	5,205	(+2)
bcel2asm $_{0.1}$	102	(+102)	495	(+495)	3,115	(+3,115)
SwingWT$_{0.91}$	892	(+28)	6,658	(+121)	34,400	(+818)

Table 6.2: Metrics on evolved wrappers, and difference w.r.t Table 6.1.

to add a call to an API object in the original Quilt test case because we wanted to observe a certain behavior at a particular point in the trace and the relevant behavior was not exercised by the application. When all traces were asserted for basic contracts, we collected compliance data for contracts for set/get and state integrity.

SwingWT

We designed 59 scenarios, between two and six for each of the 16 SwingSet2 applications. We simulated the practice communicated by the SwingWT developers: exercising GUI elements trying to imitate a user exploring the possible application use cases. For example, in the Button demo we selected all possible push buttons, check boxes, and radio buttons. We automated the scenarios using AWT `Robot`.

We collected a trace for each scenario and validated the traces against Swing. Similarly to the previous domains, we incrementally increased the contract scrutiny, and created assertion tunings when necessary.

We used SwingSet2 validated traces in two ways. First, we verified the initial compliance of SwingWT with respect to all kinds of contracts. This provides an overview of the overall compliance achieved by SwingWT developers for each category of contracts. Second, we selected two applications of the set, Button and Tooltip demos, to evolve SwingWT. We proceeded as usual, evolving by resolving violations and tuning, but using only basic contracts. We validated that the contracts were indeed valuable by submitting a patch to the developers of SwingWT, thereby providing evidence regarding *RQ1.3*.

6.1.4 Results

Table 6.2 shows wrapper size metrics after the evolution performed in the study. It also shows the increase for each metric when compared to the initial wrappers.

Table 6.3 presents an overview of the main results. This table consists of one block for each domain. The first row presents the client application, number of scenarios and number of auxiliary types specified for the scenarios (see Section 5.2 for a description of auxiliary types). The subsequent left column presents data on trace validation against the original API. For XML and Bytecode, the right column presents the data on wrapper development. For GUI, the right column presents data on verification of the compliance of SwingWT, without proper wrapper development. The wrapper was evolved only with respect to two of SwingSet2's applications; these results are presented below in the table.

Table 6.4 and Table 6.5 present the specifications of assertion tunings for the Bytecode and GUI domains, respectively. Tunings were not needed for the XML domain.

In the sequence we describe the results for each wrapper in more detail.

xom2jdom

We needed three auxiliary types and validation occurred without violations. *Koloo* detected four violations that we corrected in the wrapper. By executing XOM's suite before and after the modifications we noticed that a single API test case accounted for one of the state contract violations; three violations had no corresponding API test case. Two of these were related to reference integrity, and one to the execution of an unexpected operation (call to `getParent` on an XML object without a parent). All modifications applied to xom2jdom were simple and localized: only two new lines of code were added.

bcel2asm

Table 6.2 reflects the fact that we developed bcel2asm from scratch. We re-implemented around 25% of the BCEL types and 18% of its methods in around 13% of ASM's NLOC.

We needed eight auxiliary types (Table 6.3) because BCEL extensively uses JRE types like streams and classloaders. During validation we found one trace that our tool could not interpret due to classloader use, which we discarded. The BCEL reference integrity violation is due to a returned array with set semantics. We implemented an assertion tuning (Table 6.4) to accommodate this behavior. State contracts were violated twice due to side-effects caused by `MethodGen.getLocalVariables` (which is called by `MethodGen.getMethod`), and they were tuned with a *Skip* tuning.

During the development of bcel2asm we resorted to seven additional tunings. One was the `BytecodeTuning` presented in Figure 4.4. The other six were methods with reference integrity problems that we ignored with Skip because they were caused by our decision not to implement reuse of handles. In two occasions, state violations helped detecting

XML			
Memoranda	1 Scenario	3 Auxiliary Types	
XOM Validation		xom2jdom Development	
No violations		Reference	1
		State	3
		Tunings/Regressions	0
Bytecode			
Quilt	32 Scenarios	8 Auxiliary Types	
BCEL Validation		bcel2asm Development	
Reference	1	Reference	50
State	2	Exception	5
Tunings	3	State	2
		Tunings	7
		Regressions	19
GUI			
SwingSet2	59 Scenarios	1 Auxiliary Type	
Swing Validation		SwingWT Verification	
Sync	3	Value	72
Async	3	Reference	4
Set/Get	4	Exception	1
State	6	Sync	32
Tunings	14	Async	6
		Set/Get	8
		State	100s
SwingWT Development			
Button	4 Scenarios	***Tooltip***	5 Scenarios
Value	18	Value	1
Sync	4	Async	1
Tunings	6	Tunings/Regressions	0
Regressions	3		

Table 6.3: A summary of the empirical study results.

BCEL Validation	
Value Equality/Reference Integrity Assertion Tunings	
org/apache/bcel/generic/InstructionHandle.getTargeters :	ArrayIsSet
Object State Assertion Tunings	
org/apache/bcel/generic/MethodGen.getMethod :	Skip
org/apache/bcel/generic/MethodGen.getLocalVariables :	Skip
bcel2asm Development	
Value Equality/Reference Integrity Assertion Tunings	
org/apache/bcel/classfile/JavaClass.getBytes :	BytecodeTuning
org/apache/bcel/generic/InstructionHandle.getNext :	Skip
org/apache/bcel/generic/InstructionList.getStart :	Skip
org/apache/bcel/generic/InstructionList.append :	Skip
org/apache/bcel/generic/InstructionList.insert :	Skip
org/apache/bcel/generic/InstructionList.getTargets :	Skip
org/apache/bcel/generic/Select.getTargets :	Skip

Table 6.4: The specification of assertion tunings for the Bytecode domain.

problems in the wrapper. Repeated re-execution of asserted traces resulted in detection of 19 regressions.

In all traces that presented violations—some passed directly due to previous evolution of the wrapper—the corresponding Quilt test case started to pass while *Koloo* still detected violations. That is, while fixing the assertion violations detected by *Koloo* we have also fixed the assertions manually written by Quilt developers.

Furthermore, while trying to reproduce the behavior of BCEL using ASM, we discovered four problems in BCEL. The problems were reported as three bugs and one enhancement proposal[4], and all were acknowledged by BCEL developers to be relevant. Two bugs were detected by comparing the bytecode after `BytecodeTuning` tuned return value contracts and the other was a reference integrity problem that detected a memory leak. The enhancement proposal concerns a custom sort algorithm used by BCEL that we had to reproduce: we proposed the use of a standard sort.

SwingWT

Only one application of SwingSet2 needed an auxiliary type (from `java.bean`). During Swing validation *Koloo* detected six callback violations caused by missed asynchronous

[4]See BCEL bugs 52422, 52433, 54368 and 52441 at `http://issues.apache.org`

callbacks in an application that uses threads to paint a canvas. We added a SkipCallbacks tuning (Table 6.5) to avoid checking this non-deterministic behavior in the wrapper. We found that four methods in Swing do not follow the set/get rule. When receiving a null parameter, one method sets a default and another ignores the call completely. Two other methods enforce bounds on the parameter value. We introduced Skip tunings for these four methods.

Swing Validation	
Value Equality Assertion Tunings	
java/awt/Dimension.height :	SkipIfState
java/awt/Dimension.width :	SkipIfState
java/awt/Point.y :	SkipIfState
Callback Assertion Tunings	
javax/swing/JColorChooser.showDialog :	SkipCallbacks
javax/swing/JFileChooser.showOpenDialog :	SkipCallbacksIfState
javax/swing/JDialog.show :	SkipCallbacksIfState
javax/swing/JDialog.pack :	SkipCallbacksIfState
Set/Get Assertion Tunings	
javax/swing/AbstractButton.setMargin :	Skip
java/awt/Graphics2D.setPaint :	Skip
javax/swing/JInternalFrame.setSelected :	Skip
javax/swing/JScrollBar.setValue :	Skip
Object State Assertion Tunings	
java/awt/Component.getLocationOnScreen :	Observer isShowing
javax/swing/tree/DefaultMutableTreeNode.getLastChild :	Observer getChildCount
javax/swing/tree/DefaultMutableTreeNode.getFirstChild :	Observer getChildCount
SwingWT Development wrt. Button	
Value Equality Assertion Tunings	
java/awt/Dimension.height :	Percentage 1
java/awt/Component.getHeight :	Percentage 10
java/awt/Component.getWidth :	Percentage 10
java/awt/Point.x :	Margin 15
java/awt/Point.y :	Margin 15
Callback Assertion Tunings	
javax/swing/JFrame.pack :	SkipCallbacks

Table 6.5: The specification of assertion tunings for the GUI domain.

Most state contract violations refer to side effects of getting size and position information, or state from unprepared objects. For example, `getLocationOnScreen` throws an exception if the widget is not showing. We added an *Observer* tuning that verifies `getLocationOnScreen` only if the same object returns true to `isShowing`. We also added

Observer tunings to avoid checking tree nodes if the tree does not have children. Finally, we used *SkipIfState* and *SkipCallbacksIfState* tunings to silence violations of value equality and callback contracts that occur due to side-effects of state gathering.

Koloo detected many contract violations in SwingWT verification. 53 value equality violations were about size and positioning, 12 about missing functionality and seven about wrong state or default value in widgets. A total of 38 callbacks were found missing. SwingWT did not comply with the set/get contract in eight methods: six were empty setters, one method wraps its parameter in another object, and another method explicitly deviates from the original API behavior (as described in code comments). Hundreds of violations were detected with state contracts. Many were thread invalid access exceptions and two applications deadlocked; most violations were value equality problems and the side-effects of state gathering caused the GUIs to be distorted.

Most of the violations that drove the evolution of SwingWT were related to size and positioning. We tried to solve them until we were content with the visual result and applied tunings when required. For the Button application we used three Percentage and two Margin tunings to accommodate small differences in size and positioning (Table 6.5). We also added a SkipCallbacks because SwingWT delays the callbacks called by `JFrame.pack` until the frame is shown, which does not affect the application. Asserted traces helped us detect three regressions while developing for Button. Violations detected for Tooltip were discussed in Section 4.2.

The patch we created for SwingWT version 0.90 added 818 NLOC, 121 methods and 28 types (Table 6.2). SwingWT developers used our patch to release version 0.91[5].

6.1.5 Discussion

RQ1 — Alternative Methods

Of the four violations detected by *Koloo* in xom2jdom, only one could be detected by the original XOM test suite. This means that either the application exercises the API in ways that even a very complete test suite, like XOM's, fails to foresee (like calling `getParent` on a document without parent), or that the test case which asserts this behavior may be failing for other causes already (in Section 2.2.4 we showed that around 40% of XOM test cases still fail with xom2jdom). By concentrating on scenarios, *Koloo* can better pinpoint the wrapper evolution that is needed by the application at hand (*RQ1.1*). Furthermore,

[5]`http://swingwt.svn.sf.net/viewvc/swingwt/swingwt/trunk/CHANGELOG?revision=86`

the three additional violations revealed internal wrapper bugs that would seldom be target of assertions in API test suites.

Another result of the study is that manually written Quilt test cases were subsumed by *Koloo*'s basic contracts with respect to the behavior they asserted (*RQ1.2*). This suggests that the type of contracts used by developers in practice may be similar to the basic contracts *Koloo* checks, but are not consistently applied to all interactions with the API.

We also have evidence that *Koloo* leads to a much more compliant wrapper than the customary crash-driven method (*RQ1.3*). The verification of SwingWT showed that many compliance violations were detected, even with respect to an application that is used by the project to showcase itself. The fact that our patch for Button and Tooltip was accepted by SwingWT validates that developers were interested in solving those compliance issues. In conclusion, *Koloo* uncovered previously missing, *important* compliance issues.

RQ2 **— API Contracts**

The study showed that basic contracts indeed represent a minimum level of compliance to be achieved by wrappers, modulo tunings. Basic contracts allowed us to develop bcel2asm to be compliant with Quilt's test suite, to find bugs in BCEL, and to successfully evolve SwingWT. Value equality contracts helped detecting missing functionality, deviations in size and positioning of widgets in SwingWT, and differences in the generated bytecode in bcel2asm. A reference integrity contract helped detect a small memory leak in BCEL. Callback conformance contracts were essential to understand which functionality of the API was being overridden by SwingWT applications.

Set/get contract assertions were useful to better understand the behavior of four methods in Swing. None of the corresponding methods in SwingWT complied with such behavior, and this deviation did not occur in the interaction with applications.

For relatively complete and compliant wrappers, such as xom2jdom, state-integrity contracts helped in finding compliance violations. However, state gathering often triggered side-effects that affected other contracts, even when applied to the original APIs (BCEL and Swing). This is evidence that developers do not always comply with the convention of side-effect-free getters, which is assumed by the interface-based state-gathering heuristic.

Neither bcel2asm nor SwingWT are close to be compliant in terms of state integrity. State contracts naturally demand more of the wrapper than what the application directly needs, and may force the wrapper to implement additional functionality. Nevertheless, they appear to be useful for mature, general wrappers. Selective observation, on the other hand, was useful during the development of bcel2asm to understand the effects of the trace on specific state.

RQ3 — Assertion Tunings

We used a total of 17 tunings for validation and 13 tunings for development. In validation, tunings were used to accommodate non-deterministic behavior and side-effects of state gathering, and to relax the semantics of strict equality contracts, such as `ArrayIsSet` (Figure 4.5). In development, tunings were additionally used to document unimplemented features, such as handle reuse in BCEL, and to compromise on features that are difficult to emulate perfectly, like positioning and sizes of widgets. The results provide evidence of the need and usefulness of tunings for development and documentation of wrappers.

6.1.6 Threats to Validity

The main threats to internal validity concern possible bugs in *Koloo*, errors in the collection of data during the execution of the study, and experimenter bias. Because we manually checked each violation it is unlikely that there are false positives. However, false negatives would not affect our results. We minimized experimenter bias by looking for external confirmation of findings, such as with BCEL and SwingWT developers.

The main threats to external validity involve the subjects of the study and sample size. We mitigated the threat of subject selection by choosing subjects in three different domains of API programming. However, our results are constrained by the nature of the used APIs: object oriented libraries. Also, we were only able to study a single migration instance for each of the compared methods. More studies are thus required to corroborate our findings since the results might be affected by particularities of the studied cases.

6.2 Design Patterns Validation

In this section we turn to the validation of the design patterns proposed in Chapter 3. We validate that the patterns are useful by providing evidence for their presence in practical wrappers. To this end, we designed simple metrics that detect the presence of particular instances of the patterns from source code. We also measure auxiliary properties of wrapper types and adapters.

Wrapper$_{version}$	**SwingWT**$_{0.90}$	**SWTSwing** $_{3.2}$	**xom2jdom** $_{0.1}$	**bcel2asm** $_{0.1}$
Surrogates	474	248	37	97
Interfaces	121	46	0	6
Classic Adapters	19	27	28	62
Layered Adapters	50	49	4	0
Remakes	284	127	5	29
Internal Types	390	504	6	5
Internal Adapters	290	267	4	0
Anonymous Adapters	261	161	0	0
Stateful Adapter	106	184	15	14
Delayed Instantiation	✔	✘	✘	✘
Inverse Delegation	34	76	1	0
Adapted Replacement API Types	21	29	1	0
Wrapping Identity Map	✔	✔	✔	✔
Class Adapters	37	207	1	0
Object Adapters	107	131	31	78
Single Adaptee	22	46	19	78
Multiple Adaptees	85	85	12	0
Other Adapters	240	60	3	2

Table 6.6: Wrapper metrics for types, patterns, and adapters.

Table 6.6 presents measurements for the studied wrappers. We used the same versions of the GUI wrappers used in the study of Section 2.3, from which the patterns were derived, and prior to the SwingWT evolution discussed above. We also measured the xom2jdom wrapper resulting from the study of Section 2.2 and the bcel2asm wrapper we developed for the evaluation of the *Koloo* method in Section 6.1.

In the following sections we discuss the collected data for each pattern and for auxiliary adapter properties. Then, we consider threats to validity.

6.2.1 Layered Adapter

A type present in a wrapper project can be either a *surrogate*, when it represents a type of the original API, or an *internal type*; see the breakdown of types at the top of Table 6.6. A surrogate can be one of four kinds: an *interface*, which is simply a copy of an original API interface; a *remake*, which is a class that re-implements an original API's type without any reference to the replacement API (neither directly nor indirectly through internal types); a *classic adapter*, which is a class that references the replacement API directly but never indirectly; or a *layered adapter*, which references the replacement API indirectly through internal types.

Internal types can provide services to the wrapper, such as `ListTargetAdapter` in Figure 3.1 (Section 3.1), and can themselves be adapters, like `JListAdapter`. Internal adapters in Java can also be implemented as anonymous types.

The measurements show that the majority of surrogates in the GUI wrappers are remakes and interfaces. This is expected since the projects concentrate on wrapping widget types. GUI wrappers also have more layered adapters than classic adapters and encapsulate much of the adaptation code in internal types, which reflects the high complexity of the mappings. Because xom2jdom and bcel2asm map structurally similar APIs, they can rely mostly on classic adapters.

Most of SWTSwing implements the layered adapter as described in Figure 3.1; SwingWT uses a simpler form without the uniform interface, which is indeed not required when no alternatives have to be handled. This is reflected in SWTSwing having many more internal types than SwingWT.

6.2.2 Stateful Adapter

The Stateful Adapter pattern is frequently used in the studied projects. The metric shows all classic adapters, layered adapters, and internal types that carry additional data; we consider additional data a non-constant field whose type is neither of an original API type (a relationship among surrogates) nor of a replacement API type (an adaptee). We exclude fields that are referenced in methods for delayed instantiation because they could represent interim data. Note that the defining characteristic of the stateful adapter pattern is that the replacement API cannot carry the data, whereas in delayed instantiation the replacement API can carry the data but is not ready.

6.2.3 Delayed Instantiation

The Delayed Instantiation pattern is needed for one of the two GUI wrappers. SwingWT needs to account for SWT's strict construction time enforcement of parent-child relationships. All SwingWT widgets implement delayed instantiation by collecting interim data in surrogate fields. On the other hand, xom2jdom and bcel2asm do not need delayed instantiation because the lifecycles of surrogates and adaptees are bound. Nevertheless, other XML wrappers may very well require delayed instantiation. For instance, a migration from

XOM or JDOM to DOM[6] would face the challenge that DOM enforces document-element relationships at construction time.

6.2.4 Inverse Delegation

The Inverse Delegation pattern is implemented by both GUI wrappers and xom2jdom. The GUI wrappers use surrogates as inverse adapters. The table shows a lower bound on the number of inverse delegation instances in that it only counts class adapters to replacement API *listeners* (name ends with `Listener` or `Adapter`). We show the number of wrapper types that implement inverse delegation and the number of unique replacement API types being adapted. The xom2jdom wrapper implements a single one-to-one inverse adapter approach for node factories used in de-serialization, while bcel2asm does not require any inverse delegation.

6.2.5 Wrapping Identity Map

The Wrapping Identity Map pattern is implemented by all wrappers. SwingWT uses a static hash map field for the whole system hosted by `Component`, whereas SWTSwing uses a hash map per `Display` instance. The xom2jdom and bcel2asm wrappers use a dedicated library that maintains a map per adaptee type and instantiates surrogates reflexively when necessary.

6.2.6 Other adapter properties

Table 6.6 also summarizes how wrappers reference replacement API types: *class adapters* reference through inheritance; *object adapters* reference through object composition; *other adapters* reference in some other way, such as through calling replacement API static methods or having methods with replacement API return types. A given wrapper type may be class and object adapter simultaneously. We have not found any classic class adapter in the wrappers: all class adapters were part of a layered adapter pattern. This is also expected because Java does not allow multiple class inheritance and surrogates already implement the original API class hierarchy.

Further, the table summarizes the multiplicity of field-based references for adaptees in object adapters. A *single adaptee* is an object adapter type has a single field with a

[6]DOM is a standard W3C API for XML: `www.w3.org/DOM`

replacement API type; if there is more than one, then it is a *multiple adaptees* adapter. We count both immediate and inherited fields. The high number of multiple adaptees in GUI wrappers indicates a high structural mismatch between APIs: surrogates need multiple replacement API types to reproduce the behavior of one original API type. While xom2jdom implements a more uniform mapping that needs fewer multiple adaptees, bcel2asm uses only single adaptees, which suggests a simple type mapping between BCEL and ASM.

6.2.7 Threats to Validity

The main threat to internal validity is related to the metrics for detecting the presence of patterns. The measurements were extracted by custom made tools and may present small deviations. Because the data only support our claim that patterns are indeed used in practice small errors do not compromise the conclusions.

Threats to external validity concern the generalization of our results. We validated the patterns using projects from which the patterns were derived. This provides evidence that the patterns we abstracted are indeed present in those wrappers, which is expected. We tried to generalize our findings by measuring additional wrappers, xom2jdom and bcel2asm. Although these wrappers are simpler than the GUI wrappers, most of the proposed design patterns were still needed, which attests their generality.

6.3 Summary

In this chapter, we have presented an empirical evaluation of the *Koloo* method, and a validation of the API wrapping design patterns. The results provide evidence that *Koloo* is founded on empirically valid assumptions: that API- and application-specific tests are inadequate for API migration. The empirical study also shows that API contracts support the development of wrappers, and that assertion tunings are in fact necessary for several reasons, such as accommodating mismatches that are difficult to overcome and relaxing strict equivalence in API contracts. In the next chapter, we examine previous research on areas related to this dissertation.

Chapter 7

Previous Research

In this chapter, we discuss previous research work related to the migration of applications across APIs. Because API migration finds itself in the intersection of many disciplines, we limit our discussion to the research that is most relevant to our work. We begin with an examination of existing work on software re-engineering, of which API migration is a particular form. Next, we consider previous research on API upgrade and API migration. Then, we discuss research on software testing. We conclude the review with a brief discussion of previous research on wrapping.

7.1 Software Re-engineering

API migration is a particular form of *software re-engineering*, the "*examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*" [18]. As such, it involves some sort of *reverse engineering*, a careful analysis of a system, usually at a higher level of abstraction, aiming at a better understanding of its structure and behavior [23]. Re-engineering is also concerned with the modification of the system via *forward engineering*, whereby higher level abstractions are implemented in code, and *restructuring*, when the system is modified without changes to its external behavior [18]. In object-oriented settings, restructuring is commonly called *refactoring* [32].

Software re-engineering is probably as old as programming itself. As soon as there was a software system providing value, there was need for its maintenance and evolution [50, 68]. Of special interest to us is the more recent proposition of utilizing patterns to encode expertise in software re-engineering [82, 24, 23]. Patterns are well established abstract solutions to recurrent problems [1], and were popularized in software engineering by design patterns [34]. Stevens and Pooley [82] argued that organizations often employ modern techniques and practices to support re-engineering but have difficulties in transferring this knowledge. As a consequence, new developers have a steep learning curve before becoming productive in re-engineering tasks. They proposed systems re-engineering patterns as a "*unit of transferable expertise*" [82], and introduced a set of example patterns [24].

Demeyer et al. [23] went further and created a catalog of re-engineering patterns based on the experience of the FAMOOS project [28]. They suggested a method based on reverse engineering and re-engineering activities. For each activity type they presented small pattern languages with varying levels of abstraction, from code to requirements. Each language contains a set of related re-engineering patterns, which communicate best practices identified in the project. Their patterns are process oriented; whereas design patterns describe the design solutions as structures, re-engineering patterns describe solution processes.

Extensive work exists on tools and techniques for software re-engineering. One of the most comprehensive toolsets is Moose [29, 61], an extensible environment for re-engineering. Moose offers a common infrastructure to provide reverse engineering, refactoring, and IDE support in a language-independent fashion.

Recent research has focused on Domain Specific Languages (DSLs) to support re-engineering. DSLs provide constructs at the appropriate abstraction level for a given task, such as querying code and designing refactorings, thereby releasing developers from lower level concerns. *.QL* [22], for instance, is an object-oriented language that allows developers to express domain-specific queries over a relational representation of programs. JunGL [87] is a powerful functional language for expressing refactorings. Kniesel and Koch [46] proposed formal definitions of refactorings so that they can be statically composed by tools. Also, Framework-Specific Modeling Languages (FSMLs) [4] were proposed as an approach to capture the abstractions provided by frameworks, and to encode how these abstractions are represented in code. Antkiewicz et al. [5, 6] showed that reverse engineering framework abstractions with FSMLs is efficient and can result in high quality models. A similar approach is Metanool [13], a Moose extension that uses domain-specific annotations to support reverse engineering. Finally, Verbaere et al. [88] provided a recent account on the use of query languages for program comprehension, including re-engineering.

Reverse engineering approaches have also been used migrate applications across programming languages, such as from PL/IX to C++ [47], or from C to Java [56]. Language migration also involves the migration of the APIs used by the application and, therefore, is inherently more complex than API migration. In Malton's *barbell process* [53], for instance, API migration is restricted to the *normalization* and *optimization* phases, which are deemed much simpler than the language *translation* step. Zhong et al. [94] proposed an approach to deal specifically with API migration during language migration. It infers simple API mappings by comparing structure and names of an application before and after language conversion; the mappings can later be useful to migrate other applications. Finally, an excellent report on the major problems facing language migration attempts was given by Terekhov and Verhoef [84].

Our work builds on the research presented in this section in many ways. We use design patterns to encode domain knowledge on API wrapping techniques (Chapter 3). Compliance testing (Chapter 4) can be seen as a reverse engineering technique to query the behavior of the original API, and to understand how it differs from the wrapper. Finally, the *Koloo* method (Chapter 5) is essentially a re-engineering methodology that may benefit from the techniques and tools that support understanding and modification of applications and APIs.

7.2 API-related Application Migration

Most research on migration of applications with respect to the used APIs focus on API upgrade, the migration of an application across subsequent versions of the same API. Dig and Johnson showed that during evolution of object-oriented APIs, around 80% of the client-breaking changes are in fact refactorings [26]. The authors suggest [25] that tools should concentrate on detecting API refactorings and then replaying on client applications or generating adaptation layers. CatchUp! [37] is possibly the first tool to record refactorings while they are being performed by API developers and then replaying at client side. A disadvantage of this approach is that API developers must use this specific tool to record refactorings. Additional work on structural comparison focused on detecting refactorings or relationships between elements across dAPI versions. UMLDiff [91] creates UML models of programs and applies a comparison algorithm which results in a tree of differences. The same authors described Diff-CatchUp [90], a system that uses UMLDiff to detect differences between versions of a framework and store them in a database. Later, when clients upgrade to the new version and are confronted with compilation errors, the system uses the database to suggest plausible replacements. The comparison algorithm is a simple top-down traversal enriched with similarity metrics, which makes it difficult to detect refactorings involving different entities, such as moving a method between classes [44]. Kim et al [44] proposed an algorithm to generate change rules by detecting structural changes from method signatures, therefore mitigating the problem of a top-down traversal. Origin analysis, the determination of the origin of program elements during system evolution [35], was also used to detect relationships between elements of subsequent versions [45].

Replaying refactorings on client applications is complicated because of the unexpected contexts that may require complex analyses [77]. Some approaches, therefore, concentrate on generating adaptation layers instead. Şavga and Rudolf [77] considered recording refactorings and generating adapters through *comebacks*. They formally proved that comebacks are the reverse operators of the corresponding refactorings. The same authors presented a practical implementation of their approach and perform evaluations on realistic projects [78]. ReBa [27] records not only the refactorings but also deletions. It replays by inlining an adaptation layer in the API itself, such that the two versions of the API are merged, with the old version redirecting to new features. Clients are preserved intact, but can still take advantage of the new features if desired. Pointer analyses on the client code are performed to eliminate portions of the layer that are unnecessary for a particular application, which may improve size, performance and understandability of the API.

Some approaches go beyond refactorings, concentrating on upgrading applications in face of arbitrary API upgrade. Chow and Notkin [19] proposed that library developers should write rules that specify how clients are to be adapted in response to API changes. The main drawback is that developers must always document their changes explicitly. Perkins [70] suggested a simple mechanism for upgrading clients when API methods become deprecated. The API developer marks the method as deprecated and changes its implementation to contain the recommended fix. The system then inlines the solution into clients that depended on the deprecated method. Dagenais and Robillard [21] proposed an algorithm that looks at how the API itself adapted to changes. The assumption is that when the API changes, other portions of its code also have to adapt to the changes, and this provides valuable hints to client application developers. The implemented system, SemDiff, gathers this information and recommends adaptations to clients.

The Coccinelle system [67] was devised to support the co-evolution of Linux drivers. When Kernel developers change some aspect of the API, they apply a diff algorithm to generate a patch file. Kernel developers must then abstract away syntactic differences and control flow variations, crafting manually a *semantic patch*. Driver developers can use these patches on their own code. The main novelty is the use of computational tree logic with variables and witnesses (CTL-VW) [14] as a query language to match source code in an abstract fashion. Although proposed in a different context, Tracematches [2] offers similar matching mechanisms for Java, using AspectJ pointcuts and an automaton-based specification language.

The approaches discussed so far are related to API upgrade, when a client application needs to be migrated across API versions. We turn now to the few attempts to tackle the problem of migrating applications across different APIs. Balaban et al. [8] implemented a static analysis based on type constraints and rewrites to support the migration of applications across very similar classes. The migration between two classes is specified as a mapping between their methods, with bindings to describe parameter correspondence. Source methods that lack direct correspondents on the replacement side can be redirected to arbitrary static methods, usually encapsulated as a *FACADE*, that must be implemented with the correct migration code. The authors showed that the approach is very effective for simple migrations, such as the `Vector` to `ArrayList` example presented in Section 2.1.2. The system, however, assumes that original and replacement types are very similar, and that developers can easily determine the behavioral differences between the APIs to specify a migration script. Furthermore, it does not consider callbacks, fields, and type extensions. Another similar approach is twinning [62], a simple technique that rewrites applications or generates adaptation layers from simple type and method mappings between APIs.

A common recent trend in frameworks is the migration from inheritance-based to annotation-based means to ascribe domain roles to application entities. For example, while in JUnit 3 a test case must inherit from the framework's `TestCase` class and implement test methods starting with the string `test`, in JUnit 4 one can use any class as a test case, provided that test methods are annotated with `@Test`. Tansey and Tilevich [83] propose an approach to migrate applications from inheritance to annotation-based frameworks. A rule-based migration DSL is used to specify how to detect instances of framework concepts in source code, and how to modify those instances. A machine learning algorithm creates a migration specification from examples of applications that were already migrated, alleviating the burden of learning the specification language. This approach, nevertheless, has a very narrow scope, since it only tackles the specific inheritance to annotation migration.

In Section 7.1 we discussed how FSMLs may be used to reverse engineer framework-specific concepts from application code. Cheema [17] extended FSMLs by enriching the meta-models with code transformations that migrate concepts to corresponding elements of a different framework. The approach, however, relies on simple custom transformations, which were only implemented for a migration across the Struts[1] and JSF[2] web frameworks.

Finally, there are industrial examples of applying wrapping to migrate across APIs. Saillet [76] developed an adaptation layer between small parts of Java's Swing and SWT GUI libraries, and provided a detailed description of his experience. The open source projects SwingWT and SWTSwing, discussed throughout this dissertation, provide more comprehensive wrappers across the Swing and SWT. We have shown in our evaluation in

[1]`http://struts.apache.org/`

[2]Java Server Faces—`http://java.sun.com/javaee/javaserverfaces/`

Section 6.1 that compliance testing can help improve the compliance of such wrappers with respect to particular client applications.

7.3 Software Testing

Software testing is a broad sub-area of software engineering that has been extensively researched. In this section, we focus on research relevant to API migration, and extend the discussion on related techniques presented in Section 4.1.

Differential testing, as initially proposed by McKeeman [57], is a special case of random testing whose goal is to detect differences between different implementations, such as different compilers for the same language. As a random testing approach, differential testing lacks the focus on the behavior relevant to the application under migration. More recently, a similar approach called compatibility testing was applied to Commercial Off-The-Shelf (COTS) systems. BCT [55] is a technique to capture the behavior of components in a COTS system and generate invariants that represent the input/output and interaction behavior of components. These invariants represent an oracle that can be verified on competing implementations for compatibility [54]. BCT, however, cannot verify properties such as referential integrity because it uses object flattening to store state. It also does not allow for relaxed behavioral equivalence: violations can only be ignored. BCT's approach targets large components and it is not clear whether it will be effective for APIs that allow many relationships among fine-grained objects, and whether the extracted input/output invariants capture asynchronous callbacks.

There is a large body of work in regression testing, which is focused on preserving the behavior of an earlier version in a later version of a system. Elbaum et al. [30] proposed differential unit tests to detect differences between versions of the same unit. In this approach the system state is serialized prior to the execution of a method. During regression testing, the state is deserialized, the method is executed and results are compared. Hoffman et al. [38] proposed a framework for analyzing traces and present RPRISM, a tool that analyzes traces to identify the cause of regressions.

Koloo is mostly inspired by the test factoring [75] and SCARPE [64] techniques. These approaches capture the interaction of an application with a certain module and mock the module's behavior via a replay system that reproduces return values and callbacks. These approaches can be used, for example, to abstract costly APIs during regression tests. GenUTest [69] further generates smaller unit tests from the traces, one for each trace event that returns some value. *Koloo* presents many differences with these systems, including the verification of asynchronous callbacks, contracts for state integrity, and support for assertion tunings. The approach of [30] is state-based in that it stores the necessary state of the system prior to a method execution; *Koloo*, test factoring, SCARPE and GenUTest are action-based since they restore the state by reproducing the original events.

GUITAR [59] is a framework for automated model-based GUI testing. GUITAR analyzes a GUI application at runtime and builds a model of the available user interactions. A tool then generates regression test cases by traversing the GUI in various ways and collecting state information. A replayer can re-execute the regression tests to detect differences in the state information across versions of the application. In contrast to *Koloo*, GUITAR is domain specific and needs platform-specific rippers to create GUI models.

Traditional testing generation techniques concentrate on finding most faults. Jartege [63] performs random generation of unit tests based on JML specifications. The Korat framework [12] generates systematically all non-isomorphic test cases for the arguments of a method under test using an advanced backtracking algorithm that monitors the execution of predicates for class invariants. For these methods to be applicable to API migration, the API implementations would need to provide detailed contracts; such contracts are not available for APIs in the migrations we have performed or encountered.

JCrasher [20] generates random sequences of calls to a given API in an attempt to make it crash, thereby detecting faults. Eclat [65] performs dynamic inference of certain (API) properties (manifested as assertions) from the execution of a given test suite, and it uses the inferred properties to generate further possibly fault-revealing test cases. Randoop [66] also generates random sequences of calls, but improves efficiency by using dynamic feedback to guide the generation of additional sequences. Randoop checks default contracts (exception throwing and equals) and allows users to implement additional contract checking code. Orstra [89] is a tool for augmenting generated tests with regression oracle testing. *Koloo* implements the interface-based object-state contract proposed by Orstra; the concrete state contract is not implemented since wrappers usually do not contain the same private fields as the classes they are replacing.

Various approaches were proposed with the main goal of improving structural branch coverage: Nighthawk uses a genetic algorithm to find parameters for randomized unit testing. Ocat [41] captures and stores object state during execution and mutates the state during random testing. Palus [93] uses a combination of static and dynamic analyses to generate behaviorally diverse tests. Finally, DyGen [85] combines dynamic trace analysis with symbolic execution. All these methods are not well aligned with the objectives of API migration, where testing should be limited to behavior relevant to the client application.

Finally, some related techniques may be used to improve *Koloo*. The empirical evaluation (Section 6.1) showed that the violations uncovered by *Koloo* helped in understanding the deviation from the original API. However, finding the cause of the deviation in the wrapper and fixing it are orthogonal problems for which *Koloo* gives little help. Delta-debugging [15] might be promising in helping determine the root cause, although its applicability to APIs with long start-up times like Swing is not clear. The same work proposes a technique to automatically determine which objects to trace from an initial trace. This technique could also be adapted to support the discovery of needed auxiliary types in the type rules (see Section 5.2). Techniques for removing non-determinism from replaying of multi-threaded applications [39, 74] could also be used to better support the verification of asynchronous callbacks.

7.4 Wrapping

Wrapping is an established re-engineering technique to encapsulate existing functionality in a new interface. In object-oriented systems wrapping is the systematic application of the *ADAPTER* design pattern [34] to create an adaptation layer.

The emergence of the object-oriented paradigm created interest in migrating old procedural source code to object-oriented abstractions; this is commonly achieved via wrapping [80]. Wrapping received renewed interest recently as an approach to encapsulate legacy systems as services, from single procedures [51] to interoperable GUI elements [92]. Canfora et al. [16] propose wrappers to encapsulate the interaction with interactive legacy systems as web services. The service provides a simple request/response protocol, but it actually implements a wrapper that interacts with the legacy system. The interaction is specified as state machines. In the context of API upgrade, we have seen that wrapping has been used to migrate applications across API versions [77, 27, 78].

In this dissertation, we advance substantially the state-of-the-art in wrapping for object-oriented systems. In Chapter 3, we present a catalog of API wrapping design patterns that encode solutions to design problems specific to the API migration context. We address layered adaptation, delayed adaptee instantiation, and other peculiarities that were not addressed by previous research.

7.5 Summary

In this chapter, we have discussed previous research work related to the migration of applications across APIs. We have positioned the contributions of this dissertation while considering research in software re-engineering, API-related software migration, software testing, and wrapping. In the next chapter, we conclude the dissertation with a discussion of implications and limitations of our contributions, and future work.

Chapter 8

Conclusion

In this dissertation, we have presented our research on migration of applications across object-oriented APIs, also known as API migration. In Chapter 1 we introduced the problem and discussed the main technical approaches: wrapping and rewriting. In Chapter 2 we described two studies and a set of interviews we conducted to explore API migration in practice. The results of these activities informed the remainder of our work.

We introduced a catalog of API wrapping design patterns in Chapter 3. The patterns guide developers in the design and implementation of wrappers across APIs. Then, we presented the novel concept of compliance testing for API migration in Chapter 4. Compliance testing supports the discovery of behavioral mismatches between a wrapper and its original API, and the assessment of wrapper correctness. Compliance tests may use assertion tunings to document acceptable deviations a wrapper makes from the behavior of its original API, thereby making explicit the notion of a "good enough" wrapper, which exists informally in practice. The *Koloo* method for API migration, discussed in Chapter 5, prescribes a methodology for developing an API wrapper using compliance testing and design patterns. The method fits within the iterative, sample-driven approach used in practice. We evaluated the method, and compliance testing in particular, in an empirical study presented in Chapter 6. Finally, we discussed related research in Chapter 7.

In this chapter, we consider the implications and limitations of the research presented in this dissertation. Then, we conclude with a discussion of future research directions.

8.1 Implications and Limitations

API migration is complex. Developers must understand how the application uses the original API. Then, they must discover how the replacement API can reproduce the behavior of the original API in the use cases exercised by the application. Next, they must specify this translation, either by wrapping or rewriting. Finally, developers must determine the level of equivalence considered "good enough" and enforce this equivalence, thereby verifying migration correctness.

Our work proposes techniques and a method to manage this complexity. We focus on a wrapping approach because it may be the objective of the migration or may guide a subsequent rewrite. We showed that compliance testing supports the discovery of behavioral differences between a wrapper and its original API. Compliance testing also makes explicit the compromises and assumptions of a wrapper, and automates the enforcement of behavioral equivalence under these assumptions. Developers can follow the steps of the *Koloo* method to incrementally develop a wrapper tailored for the application under migration. The method guides the extraction of compliance tests from scenarios that encode the requirements for the migration. The *Koloo* toolkit executes compliance tests that drive wrapper development and support detection of regressions. Wrapping design patterns guide the design and implementation of the wrapper.

We now discuss some limitations of our research. We identify limitations related to our choice of wrapping as the technical approach, limitations of the compliance testing technique, of the *Koloo* method, and of our evaluation.

Object-oriented wrappers effectively support migration across traditional object-oriented APIs because they can naturally encapsulate behavioral adjustments; however, modern APIs increasingly employ declarative specifications, such as annotations and configuration files, to assign domain roles to application entities. In this situation, the design of the wrapper is convoluted. Thus, the *Koloo* method supports the development of wrappers that may be used, for example, to derive rewriting specifications across traditional class libraries [8]. The special case of migration from inheritance to annotation based frameworks [83] is not immediately handled by *Koloo*.

Compliance testing focuses on functional equivalence between a wrapper and the original API. Non-functional requirements, such as performance and memory consumption, are not immediately addressed. Nevertheless, we can extend compliance testing with new types of contracts, including contracts that observe side-effects. For example, a contract that observes memory allocation performed by API methods can enforce memory consumption requirements on the wrapper.

The *Koloo* method guides the wrapper development process by focusing on the extraction of test cases to enforce behavioral equivalence on selected scenarios. However, much is still left to developers. In particular, *Koloo* does not provide quality criteria for scenario design. For example, developers must design scenarios that cover important migration requirements, but *Koloo* offers no guidance on how to design scenarios for best coverage, or how to select among potential scenarios. Furthermore, scenario design may involve technical challenges, such as test automation for GUI APIs. Finally, the cost of API migration is largely dominated by the actual wrapper refinement, which often involves deeply understanding original and replacement APIs. *Koloo* does not provide any program comprehension technique specific to API migration, so developers must rely on existing, general approaches.

Our evaluation also contains limitations. We compared *Koloo* against alternative methods in terms of the ability to drive the development of behaviorally compliant wrappers. Our empirical study provides good evidence that *Koloo* is superior in this regard, but many important aspects were not evaluated. For example, we have not performed a user experiment so we have no evidence on how *Koloo* works in practice. We also have not assessed the quality of the wrappers developed with the different techniques. Lastly, the validation of the wrapping design patterns used the projects from where they originated, so we need more studies to understand their generality and usefulness.

Finally, throughout this thesis we have focused on migration across object-oriented APIs. We have not investigated the challenges that may arise in other paradigms, such as APIs in logic or functional programming languages. We have also assumed that original and replacement APIs provide domain concepts at a similar level of abstraction. For example, in the XML domain we investigated migration across document object model APIs, such as DOM, XOM, and JDOM. However, we have not looked at migration across APIs with a completely different programming model, such as from XOM to an XML parser like SAX or StAX, or to an object binding API like JAXB.

Despite its limitations, the research presented in this dissertation represents a step towards effectively managing API migration. The *Koloo* method guides the wrapper development life-cycle. The patterns guide the design and implementation of solutions to some common wrapping problems. While these contributions improve the state-of-the-art in API migration, compliance testing is particularly important to enable further research. Compliance tests precisely define the requirements of an API migration and support the assessment of migration correctness. Therefore, compliance testing determines the quality of a wrapper. This, in turn, enables new research to focus on support for wrapper development: new approaches can count on compliance testing to evaluate the results of the proposed techniques.

8.2 Future Work

We next outline research directions enabled by the contributions of this dissertation.

API Matching

Matching is an operator common to the management of database schemas [71, 52, 79, 7], model-driven engineering meta-models [31, 43], and ontologies [42]. Matching takes two distinct representations of the same data and produces an alignment: a set of mappings between corresponding elements.

We believe that matching can play an important role in supporting wrapper development. We could create models of original and replacement APIs, and use matching algorithms to find corresponding elements at various levels of granularity. For example, we could bootstrap the wrapper skeleton using a mapping over API types to specify adaptee types for surrogates; and during surrogate method implementation, the mapping over API methods may be used to suggest replacement API methods for that particular context.

We performed some experiments with the general similarity flooding matching algorithm [58] to match API models. The early results were promising, particularly when the APIs are structurally similar, as is the case for the XML and Bytecode domains. Other work on semi-automatic identification of an API mapping by a combination of API matching techniques [72] could also be leveraged.

Language Support

The API design patterns proposed in this dissertation guide the design and implementation of wrappers to neutralize certain API differences. However, the implementation of some patterns may involve a large amount of boilerplate code, such as layered adapter or code that is difficult to get correct, as in delayed instantiation.

Programming language support may help developers overcome these difficulties. For instance, a construct that allows explicit type mappings may be used to generate code for layered adapter and API wrapping identity map. Also, explicit classification of delayed instantiation operations enables generation of difficult code to detect when the adaptee may be instantiated and interim data transferred to it.

Static Program Analysis

Wrappers make assumptions about how client applications interact with the original API. For example, if delayed instantiation is used, only a sub-set of surrogate operations are safe; the wrapper assumes only those operations will be called prior to adaptee instantiation. Compliance testing also assumes that the wrapper will be used only with the data and call sequences exercised by the designed scenarios.

Wrappers are implemented with respect to a client application. However, developers may desire to leverage existing wrappers to migrate additional applications. They can follow the *Koloo* method, design scenarios, collect traces and verify the wrapper. But static program analyses may provide better guarantees (that the wrapper will work in all cases) and be cheaper to execute (no need to design scenarios) for an initial assessment. A program analysis can verify whether a client application calls any unsafe operation prior to adaptee instantiation.

Wrapper Inlining

This dissertation assumes the goal of the API migration is to produce a compliant wrapper that may guide rewriting if desired. As discussed before, rewriting needs automation to be effective, because it is an error prone, repetitive task. Thus, we need an approach to leverage wrappers as specifications to automate client application rewriting. We call this approach *wrapper inlining*.

Wrapper inlining is a generalization of the API upgrade approach by Perkins [70]. While his work proposes to inline the implementation of deprecated methods at call sites, we would like to inline the whole wrapper implementation. That is, application variables of surrogate types need to be translated to corresponding adaptee types; surrogate method implementations need to be inlined at application call sites; and additional data carried by surrogates may need to be stored by the application.

We see two main challenges in wrapper inlining. First, wrappers may contain complex structures that may be difficult to inline. A possible solution is to identify patterns in these structures and provide programming language support for their specification. This may restrict how wrappers are implemented, but enable more powerful analyses to support wrapper inlining. Second, the resulting application code can be unmaintainable after rewriting because the complexity hidden in the wrapper is spread over application code. In Section 2.2.5 we discussed the implications of a large number of complex adjustments to rewriting approaches. We believe partial evaluation might be employed to simplify inlined code for specific calling contexts. For example, a partial evaluation analysis could decide that a given compliance issue does not need to be dealt with at a particular call site and the handling code can be removed.

IDE Support

Based on our experience developing wrappers, we believe the following IDE features have potential to greatly improve development productivity and deserve further research:

Recommendation system — controlled by a developer, a recommendation system may use information from API matching algorithms to recommend replacement API elements to be used in the implementation of a given wrapper type or method.

Trace analysis — compliance tests are encoded as dynamic API interaction traces. IDEs may support analysis of traces by, for example, providing GUIs with syntax highlighting and views for selective trace slices based on control and data dependencies. While compliance testing supports the detection of wrapper problems by raising assertion violations, trace analysis should support root-cause analysis of the detected violations to guide wrapper evolution. Trace minimization [15] techniques are promising in this area.

Multiple dimensions of knowledge — IDEs could integrate multiple dimensions of knowledge to support API migration [11]. For instance, API documentation, sample applications, and API usage data should be integrated in editors to assist developers with relevant information for a specific task.

References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364, October 2005.

[3] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 144–153, November 2007.

[4] Michal Antkiewicz. *Framework-Specific Modeling Languages.* PhD thesis, University of Waterloo, September 2008.

[5] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Automatic Extraction of Framework-Specific Models From Framework-Based Application Code. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 214–223, November 2007.

[6] Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Fast Extraction of High-Quality Framework-Specific Models From Application Code. *Journal of Automated Software Engineering*, 16(1):101–144, March 2009.

[7] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 906–908, June 2005.

[8] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 265–279, October 2005.

[9] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and Back: Patterns for API Migration by Wrapping. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM)*, pages 1–10, September 2010.

[10] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an API Migration for Two XML APIs. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE)*, pages 42–61, October 2009.

[11] Thiago Tonelli Bartolomei, Mahdi Derakhshanmanesh, Andreas Fuhr, Peter Koch, Mathias Konrath, Ralf Lämmel, and Heiko Winnebeck. Combining multiple dimensions of knowledge in API migration. In *Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM) at the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, March 2011.

[12] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[13] Andrea Brühlmann, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Enriching Reverse Engineering with Annotations. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 660–674. September 2008.

[14] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking. In *Proceedings of the 36th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 114–126, January 2009.

[15] Martin Burger and Andreas Zeller. Minimizing reproduction of software failures. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 221–231, July 2011.

[16] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480, April 2008.

[17] Aseem Paul Singh Cheema. Struts2JSF: Framework Migration in J2EE Using Framework Specific Modeling Languages. Master's thesis, University of Waterloo, May 2007.

[18] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[19] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 359–368, November 1996.

[20] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice Experience*, 34(11):1025–1050, September 2004.

[21] Barthélémy Dagenais and Martin P. Robillard. Recommending Adaptive Changes for Framework Evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):481–490, September 2011.

[22] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-Oriented Queries Made Easy. In *Proceedings of the 2nd Generative and Transformational Techniques in Software Engineering Summer School (GTTSE)*, pages 78–133. July 2008.

[23] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.

[24] Rick Dewar, Ashley D. Lloyd, Rob Pooley, and Perdita Stevens. Identifying and Communicating Expertise in Systems Reengineering: A Patterns Approach. *IEEE Software*, 146(3):145–152, June 1999.

[25] Danny Dig and Ralph Johnson. Automated Upgrading of Component-Based Applications. In *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 675–676, October 2006.

[26] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, April 2006.

[27] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 441–450, May 2008.

[28] Stéphane Ducasse and Serge Demeyer. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, October 1999.

[29] Stéphane Ducasse, Tudor Girba, Michele Lanza, and Serge Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. *Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series*, pages 55–71, 2005.

[30] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving Differential Unit Test Cases from System Test Cases. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 253–264, November 2006.

[31] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Model Driven Engineering Languages and Systems*, pages 326–340. September 2008.

[32] Martin Fowler. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[33] Martin Fowler. Eradicating Non-Determinism in Tests, April 2011. Blog post `http://martinfowler.com/articles/nonDeterminism.html`.

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

[35] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering (TSE)*, 31(2):166–181, February 2005.

[36] William H. Harrison and Harold L. Ossher. Member-group relationships among objects. In *Workshop on Foundations Of Aspect-Oriented Languages (FOAL) at the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 02–06, April 2002.

[37] Johannes Henkel and Amer Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 274–283, May 2005.

[38] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 453–464, June 2009.

[39] Jeff Huang, Peng Liu, and Charles Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 207–216, November 2010.

[40] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse : Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[41] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: Object Capture-based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170, July 2010.

[42] Yannis Kalfoglou and Marco Schorlemmer. Ontology Mapping: The State of the Art. *The Knowledge Engineering Review*, 18(1):1–31, 2003.

[43] Kelly Garces, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Adaptation of Models to Evolving Metamodels. Research Report RR-6723, INRIA, October 2008.

[44] Miryung Kim, David Notkin, and Dan Grossman. Automatic Inference of Structural Changes for Matching Across Program Versions. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 333–343, May 2007.

[45] Sunghun Kim, E. James Whitehead, and Jennifer Bevan Jr. Properties of Signature Change Patterns. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM)*, pages 4–13, September 2006.

[46] Günter Kniesel and Helge Koch. Static Composition of Refactorings. *Science of Computer Programming*, 52(1-3):9–51, August 2004.

[47] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code Migration through Transformations: An Experience Report. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 1–13, December 1998.

[48] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC), Technical Track on Programming Languages*, March 2011.

[49] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[50] Meir. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.

[51] Maozhen Li, Bin Yu, Man Qi, and Nick Antonopoulos. Automatically Wrapping Legacy Software into Services: A Grid Case Study. *Peer-to-Peer Networking and Applications*, 1(2):139–147, 2008.

[52] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic Schema Matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 49–58, September 2001.

[53] Andrew J. Malton. The Software Migration Barbell. In *ASERC Workshop on Software Architecture*, August 2001.

[54] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezzè. Compatibility and regression testing of COTS-component-based software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 85–95, May 2007.

[55] Leonardo Mariani and Mauro Pezzè. Behavior Capture and Test: Automated Analysis of Component Integration. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, June 2005.

[56] J. Martin and H.A. Muller. C to Java migration experiences. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 143–153, March 2002.

[57] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[58] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: a Versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 117–128, March 2002.

[59] Atif Memon, Adithya Nagarajan, and Qing Xie. Automating regression testing for evolving GUI software. *Journal on Software Maintenance and Evolution*, 17(1):27–64, January 2005.

[60] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[61] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Grba. The story of moose: an agile reengineering environment. In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 1–10, September 2005.

[62] Marius Nita and David Notkin. Using Twinning to Adapt Programs to Alternative APIs. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 205–214, May 2010.

[63] Catherine Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *Quality of Software Architectures and Software Quality*, pages 242–256, 2005.

[64] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *Proceedings of the 3rd International Workshop on Dynamic Analysis (WODA)*, pages 1–7, May 2005.

[65] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *Proceedings of the 19th European Conference Object-Oriented Programming (ECOOP)*, pages 504–527, July 2005.

[66] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *Proceedings of the 22th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 815–816, October 2007.

[67] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. *SIGOPS Operating Systems Review*, 42(4):247–260, 2008.

[68] David Lorge Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 279–287, May 1994.

[69] Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai. GenUTest: a unit test and mock aspect generation tool. *International Journal on Software Tools for Technology Transfer*, 11(4):273–290, October 2009.

[70] Jeff H. Perkins. Automatically generating refactorings to support API evolution. In *Proceedings of the 6th ACM SIGPLAN SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 111–114, September 2005.

[71] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001.

[72] Daniel Ratiu, Martin Feilkas, Florian Deissenboeck, Jan Jürjens, and Radu Marinescu. Towards a Repository of Common Programming Technologies Knowledge. In *Proceedings of the International Workshop on Semantic Technologies in System Maintenance (STSM)*, 2008.

[73] Trygve Reenskaug. Models-views-controllers. Technical report, Xerox PARC, 12 1979. Available at: `http://heim.ifi.uio.no/~trygver/mvc/index.html`.

[74] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, 2003.

[75] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 114–123, November 2005.

[76] Yannick Saillet. Migrate your Swing application to SWT. IBM DeveloperWorks, January 2004.

[77] Ilie Şavga and Michael Rudolf. Refactoring-Based Support for Binary Compatibility in Evolving Frameworks. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 175–184, October 2007.

[78] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical Refactoring-Based Framework Upgrade. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 171–180, October 2008.

[79] Pavel Shvaiko and Jérôme Euzenat. A Survey of Schema-Based Matching Approaches. In *Journal on Data Semantics IV*, volume 3730, pages 146–171. 2005.

[80] H.M. Sneed and R. Majnar. A case study in software wrapping. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 86–93, 1998.

[81] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 391–400, May 2011.

[82] Perdita Stevens and Rob Pooley. Systems reengineering patterns. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 17–23, November 1998.

[83] Wesley Tansey and Eli Tilevich. Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 295–312, October 2008.

[84] Andrey A. Terekhov and Chris Verhoef. The Realities of Language Conversions. *IEEE Software*, 17(6):111–124, 2000.

[85] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. DyGen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proceedings of the 4th International Conference on Tests and Proofs (TAP)*, pages 77–93, June 2010.

[86] Bill Venners. A Design Review of JDOM: A Conversation with Elliotte Rusty Harold, Parts I-VII. `http://www.artima.com/intv/xmlapis.html` (March 26, 2012), 2003.

[87] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 172–181, May 2006.

[88] Mathieu Verbaere, Michael W. Godfrey, and Tudor Girba. Query Technologies and Applications for Program Comprehension. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC)*, pages 285–288, June 2008.

[89] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings of the 20th European Conference Object-Oriented Programming (ECOOP)*, pages 380–403, July 2006.

[90] Zhenchang Xing and E. Stroulia. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering (TSE)*, 33(12):818–836, 2007.

[91] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE)*, pages 54–65, November 2005.

[92] Bo Zhang, Liang Bao, Rumin Zhou, Shengming Hu, and Ping Chen. A Black-Box Strategy to Migrate GUI-Based Legacy Systems to Web Services. In *Proceedings of*

the International Symposium on Service-Oriented System Engineering (SOSE), pages 25–31, December 2008.

[93] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 353–363, July 2011.

[94] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API Mapping for Language Migration. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 195–204, May 2010.