Swing to SWT and Back: Patterns for API Migration by Wrapping

Thiago Tonelli Bartolomei and Krzysztof Czarnecki
University of Waterloo
Waterloo, Canada

Ralf Lämmel
University of Koblenz-Landau
Koblenz, Germany

***Abstract*—Evolving requirements may necessitate *API migration*—re-engineering an application to replace its dependence on one API with the dependence on another API for the same domain. One approach to API migration is to replace the original API by a wrapper-based re-implementation that makes reuse of the other API. Wrapper-based migration is attractive because application code is left untouched and wrappers can be reused across applications. The design of such wrappers is challenging though if the two involved APIs were developed independently, in which case the APIs tend to differ significantly. We identify the challenges faced by developers when designing wrappers for object-oriented APIs, and we recover the solutions used in practice. To this end, we analyze two large, open-source GUI wrappers and compile a set of issues pervasive in their designs. We subsequently extract design patterns from the solutions that developers used in the GUI wrappers.**

I. Introduction

Wrapping is an established re-engineering technique to provide access to existing functionality through a preferred interface. One well-known form of wrapping serves the purpose of migrating old procedural code to OO abstractions [1]. In this case, one designs a convenient interface and implements it in terms of legacy procedures and associated state.

Wrapping also has been used for *API migration*, and the present paper follows this path. API migration by wrapping means that an existing API is re-implemented in terms of another API. As a result the original implementation is no longer needed. As a benefit of wrapping, application code does not need to be changed. Such wrapping has been specifically researched for the purpose of API upgrade [2], [3], where the wrapper allows application code to continue using the interface of the earlier API version re-implemented on top of the new version of the API.

Our efforts are not focused on API upgrade; instead we are interested in wrapping as a means to perform API migration across different APIs. Consider the following scenario which is representative for the practical importance of such wrapping:

> *An effort on application integration requires to merge several components which exercise different GUI APIs, say x and y. For the sake of a uniform and amalgamated user interface, one of the two APIs has to be eliminated, say y. This can be achieved by API migration such that a wrapper API y' is used instead of y where y' is a wrapper around x.* When compared to an approach that is based on source-code rewriting, wrapping requires few (ideally, no) adaptations of components using y.

In previous work [4], we studied API migration for two different XML APIs in the Java platform: XOM and JDOM. In terms of their interfaces, the chosen APIs are very similar, but the actual implementations differ systematically in the contracts for many methods. Our study was meant to measure the effort needed to develop a sufficiently compliant wrapper in such a situation. The overall result of that work is twofold: a) wrappers need to involve a lot of tuning in order to achieve semantical equivalence in terms of pre- and post-conditions; b) full compliance of the wrapper's implementation with the original API may be impractical; compliance relative to an 'application of interest' is achievable though.

In this paper, we address a different aspect of API migration by wrapping. That is, we try to understand the *design of the wrappers* themselves. As long as the involved APIs are very similar in terms of their interfaces and type hierarchies, the wrapper design is arguably straightforward—the classic *ADAPTER* design pattern [5] has to be applied systematically. The wrapper's design of our earlier work is indeed relatively straightforward but a few special cases had to be dealt with. We have studied wrapping for other APIs, and we have encountered wrapper designs that are considerably more involved due to differences in deep type hierarchies, instantiation protocols, and multiplicities in the adapter/adaptee relationships.

Contributions

- We report on a study of two large open source projects, SwingWT and SWTSwing, which implement wrappers between the most prominent Java GUI APIs, Swing and SWT. This is the first design study of API wrappers.
- Based on the wrappers of the study and our earlier work, we identify a set of design challenges faced by developers when implementing wrappers around OO APIs.
- We analyze the solutions employed by developers for the aforementioned challenges, and generalize these solutions as (sketches) of design patterns, which can be used by developers facing similar problems.
- We show that the obtained patterns occur frequently in the wrappers of our study and our previous work. The patterns are not specific to our subjects; they involve mismatches that can occur between any pair of APIs.

Road-map of the paper §II sets up the basic notion of API wrapping. §III describes the design of our wrapper study. §IV identifies the challenges of API wrapper design. §V presents the abstracted design patterns. §VI presents measurements regarding the presence of the patterns in actual wrappers. §VII discusses threats to validity. §VIII discusses related work. §IX concludes the paper.

II. Basics of API wrapping

A. Terminology

We say that we migrate from the source API to the target API. Here *source API* refers to the original API that is to be replaced, and *target API* refers to the API that is to be re-used in re-implementing the source API. The term *surrogate* is used for a type that is meant to re-implement a type of the source API while preserving its interface. A *wrapper* is a collection of such surrogates and possibly internal helper types.

B. The tailored ADAPTER pattern

The *ADAPTER* design pattern is the classic solution to integrate a client class with types that have different interfaces than it expects. The GoF [5] book describes two variants of adapters: class and object. In both cases, an adapter class extends the type expected by the client in order to intercept requests and redirect them to the new target (the *adaptee*). Class adapters integrate the adaptee through inheritance whereas object adapters use object composition.

In the context of API migration, the adapter is the surrogate object, and the adaptee is an object of the target API. Surrogates do not extend the expected types, i.e., the classes of the source API; instead they replace them. Thereby the constructor calls in the client application can also be retained. In principle, the surrogates may even populate the same namespace as the source API. For generality's sake, and in order to avoid issues with multiple inheritance, surrogates may prefer the object adapter over the class adapter.

C. A simple example

Let us migrate applications using Java `Vectors` to `ArrayLists`—a simple scenario borrowed from [6]. In the source code of Fig. 1, 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 (l.2). Both objects have associated lifecycles: an `ArrayList` object is created in the constructor of a `Vector` (l.4). Because source and target APIs are very similar in this example, most of the client requests can be simply delegated to the adaptee (as in l.7) while some semantic adjustments are encapsulated in the adapter (l.9-12).

As we migrate clients from `Vector` to `ArrayList`, we also need to migrate them from `Enumerations` to `Iterators`. Since `Enumeration` is a Java interface, we simply copy it to the wrapper (not shown in the figure). We also 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 (l.13-15) creates a new adapter around the iterator returned by its `ArrayList` adaptee.

III. Study Design

A. Methodology

Our study is designed to answer these research questions:

```
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) { this.adaptee = i; }
21     public Object nextElement() { return adaptee.next(); }
22     ...
23 }
```

Fig. 1. Vector and Enumeration adapters.

What are the design challenges faced by developers when implementing wrapping layers around OO APIs? What are the solutions employed in practice?

The study is primarily based on two major open-source wrapping projects implementing both directions between Java's GUI APIs Swing and SWT. Furthermore, the study takes into account our previous work on a wrapper in the XML domain [4].

The study was conducted in two steps. The first step consisted in uncovering important design challenges faced by developers; see §IV. We initially studied the APIs and detected some 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 wrappers' source code trying to understand the correspondences between types of the source and target APIs.

The second step comprised understanding the solutions for the identified challenges and eventually abstracting solutions as design patterns; see §V. To this end, we performed architectural code queries and manual inspection on source code. Our investigation resulted in 5 design patterns. Finally, we designed simple metrics to provide evidence that the patterns are indeed applied in practice; see §VI.[1]

B. Subjects

Table I presents the wrappers and APIs that are subjects of our study. For wrappers, we indicate the API they replace (source API) and the one they use instead (target API). For the APIs, we show the studied version, used by the wrappers. All sub-packages of top-level packages were included.

The central subjects are the GUI wrappers SwingWT and SWTSwing, but we also leveraged our XML wrapper

[1]Supporting material, including the source code for every example in this paper can be found at http://gsd.uwaterloo.ca/icsm2010/. The distribution also contains additional wrapper implementations that were built to further validate the discussion of §IV and §V.

Wrapper	**SwingWT**	**SWTSwing**	**XOM2JDOM**
	swingwt.sf.net	swtswing.sf.net	
Source API	Swing	SWT	XOM
Target API	SWT	Swing	JDOM
Types	617	660	42
Visible	506	361	40
Methods	6.115	5.894	587
Visible	5.480	3.246	432
Fields	2.296	2.613	69
Visible	1.814	1.068	16
Constants	1.188	572	16
NCLOC	30.818	64.564	7.257

API$_{ver.}$	**Swing**$_{1.4}$	**SWT**$_{3.2.2}$	**XOM**$_{1.2.1}$	**JDOM**$_{1.1}$
	Core Java API	www.eclipse.org/swt	www.xom.nu	www.jdom.org
Top-Level Packages	java.awt javax.swing	org.eclipse.swt	nu.xom	org.jdom
Types	1.986	600	110	73
Visible	1.212	401	50	52
Methods	18.627	8.639	884	908
Visible	12.833	5.424	358	644
Fields	7.438	5.016	279	256
Visible	2.965	2.522	17	52
Constants	1.943	1.606	17	29
NCLOC	234.164	89.901	19.395	8.755

TABLE I
THE SUBJECTS OF THE STUDY: THE WRAPPERS (LEFT) AND THE APIS (RIGHT).

XOM2JDOM available from a previous study. The GUI wrappers were chosen because they wrap large, complex APIs, are open-source and we could directly contact their main developers. The XML wrapper is mainly shown for comparison. By the nature of the involved APIs, this wrapper has a much simpler design than the GUI API wrappers, and it is less suited for an objective discussion because we have developed it.

The table presents size metrics for wrappers and APIs. We consider types, methods, fields, and constants (i.e., final fields in Java). *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. The table shows that all wrappers have fewer visible entities than their source APIs, i.e., they are not complete. A client application which references some of the missing elements would not compile if used with the wrappers. Even though the wrappers cover only parts of the APIs, the wrapper sizes, measured as number of non-comment lines of code (NCLOC), are substantial: 1/3 and 1/4 of the size of the respective target APIs.

IV. CHALLENGES OF GUI API WRAPPING

Now we examine the challenges that emerge when designing wrappers around Swing and SWT. To this end, we use two versions of a simple program with a GUI; one version for Swing; another version for SWT. First, we observe the differences between the two versions. Second, we discuss the issues that arise in defining wrapping layers that serve both directions. We use Swing2SWT and SWT2Swing to refer to these directions. For instance, Swing2SWT is the wrapping layer around SWT yet with Swing's interface so that application code written for Swing could execute with this wrapper and get SWT widgets instead; likewise for SWT2Swing.

A. Sample-based inquiry into API differences

Fig. 2 shows the Swing and SWT-based GUIs for a program that translates words from a list box—all words or only the selected ones. Fig. 3 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.

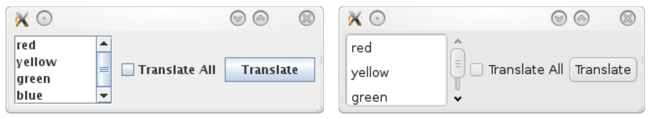

Fig. 2. Swing and SWT versions of a simple GUI.

The handler of the button calls the reusable `Translator` class which is also configured with the value of the checkbox (l.20-21). The called method (l.36-40) recursively searches for lists in a window, and depending on configuration, it translates all words in the list or just the selected ones.

The two source-code versions are relatively straightforward. In Swing (Fig. 3, left), `JFrame` (l.1) represents the main window and `JScrollPane` (l.4-5) creates the scrollable area in which the list resides. `JList` (l.11) implements the MVC view, while `DefaultListModel` (l.6-10) implements the MVC model expected by `JList`. `JCheckBox` (l.13) and `JButton` (l.16) are the button widgets. Lines 18-23 register an `ActionListener` to the `JButton`. The listener will receive a callback from the API when the button is pressed. A layout is created for the window (l.25) and the widgets are wired together (l.27-30). Swing maintains its own thread; hence, 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 (Fig. 3, right) is strikingly similar. `Shell` (l.2), `List` (l.4-5) and `Button` (l.13 and l.16) represent the window, list and button widgets, respectively. Notable differences include `Button` being configured to behave as a checkbox (l.13) or a push button (l.16), and the lack of a scroll pane widget since List can be configured to show a scroll bar (l.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; hence, lines 31-34 implement a loop to control the application's main thread.

B. Non-trivial mapping multiplicities

Our first task is to map source API types to corresponding target API types, thereby preparing the surrogates' implementation. The trivial collection wrapper of §II only involved one-to-one correspondences between adapter and adaptee. The GUI

```
 1      final JFrame frame = new JFrame();
 2      Container pane = frame.getContentPane();
 3
 4      JScrollPane scroll = new JScrollPane();
 5      scroll.setPreferredSize(new Dimension(100, 70));
 6      DefaultListModel model = new DefaultListModel();
 7      model.addElement("red");
 8      model.addElement("yellow");
 9      model.addElement("green");
10      model.addElement("blue");
11      JList list = new JList(model);
12
13      final JCheckBox check = new JCheckBox("Translate␣All");
14
15
16      JButton translate = new JButton("Translate");
17
18      translate.addActionListener(new ActionListener() {
19          public void actionPerformed(ActionEvent e) {
20              new Translator().translateLists(
21                frame, check.isSelected());
22          }
23      });
24
25      pane.setLayout(new FlowLayout());
26
27      scroll.getViewport().setView(list);
28      pane.add(scroll);
29      pane.add(check);
30      pane.add(translate);
31
32      frame.pack();
33      frame.setVisible(true);
34      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35  ...
36    public class Translator {
37      public void translateLists(Container container, boolean all) {
38          for (Component c : container.getComponents()) {
39              if (c instanceof JList) { ... }
40    }}}
```

```
 1      Display display = new Display();
 2      final Shell shell = new Shell(display);
 3
 4      List list = new List(shell,
 5          SWT.BORDER | SWT.MULTI | SWT.V_SCROLL);
 6      list.setLayoutData(new RowData(100, 70));
 7      list.add("red");
 8      list.add("yellow");
 9      list.add("green");
10      list.add("blue");
11
12
13      final Button check = new Button(shell, SWT.CHECK);
14      check.setText("Translate␣All");
15
16      Button translate = new Button(shell, SWT.PUSH);
17      translate.setText("Translate");
18      translate.addSelectionListener(new SelectionAdapter() {
19          public void widgetSelected(SelectionEvent e) {
20              new Translator().translateLists(
21                  shell, check.getSelection());
22          }
23      });
24
25      RowLayout layout = new RowLayout();
26      layout.center = true;
27      shell.setLayout(layout);
28
29      shell.pack();
30      shell.open();
31      while (!shell.isDisposed ()) {
32          if (!display.readAndDispatch()) display.sleep();
33      }
34      display.dispose();
35  ...
36    public class Translator {
37      public void translateLists(Composite composite, boolean all) {
38          for (Control c : composite.getChildren()) {
39              if (c instanceof List) { ... }
40    }}}
```

Fig. 3. Swing (left) and SWT (right) source code versions of the GUI application displayed in Fig. 2.

wrappers require additional mapping multiplicities, presented in Table II, which pose challenges to the wrapper design.

By analyzing the code of Fig. 3 we can see that SWT's `Display` (l.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 a source API type implements variants which are distributed over different target 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.

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 II
MAPPING MULTIPLICITIES WRT. A SINGLE SOURCE API OBJECT.

Fig. 3 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. SWT2Swing's `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 $List \rightarrow (JList, ListModel)|(JList, ListModel, JScrollPane)$.

Table II only shows multiplicities with respect to a single source API object. The opposite direction of a Composite Targets mapping, however, also represents an issue to wrappers because multiple 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 agree in the features they offer and data they carry. `Shell`'s `disposed` flag (l.31), for example, is missing

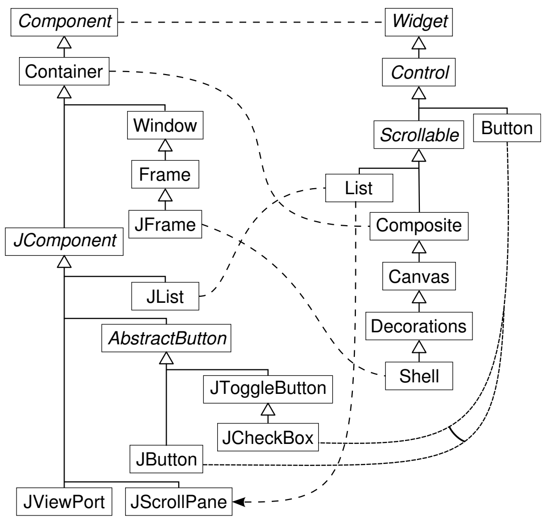

Fig. 4. One option for Swing and SWT Type Mappings.

in `JFrame`, and must be re-implemented by the surrogate.

C. Mapping varying type hierarchies

Fig. 4 shows an excerpt of Swing and SWT's class hierarchies, with the type mappings conjectured in the previous section. Further, `Component` is mapped to `Widget` since they are the super-types 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 [5]. For `List`'s case of a *Composite Target* we use a directed edge to `JScrollPane` to express the uni-directionality of this mapping. Finally, we note that the mappings of the open-source wrappers SwingWT and SWTSwing are slightly different from Fig. 4.

Wrappers should provide surrogates that completely reflect the inheritance hierarchy of the source API for two main reasons. First, client applications can reference and extend any visible type of the source API. Clients using surrogates that do not comply with the hierarchy may not even compile. For instance, if Swing2SWT exposed `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 source 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. This kind of reuse, however, can only be achieved if type mappings respect covariance. Fig. 4 shows that `Component`$\rightarrow$`Widget` and `Container`$\rightarrow$`Composite` are covariant mappings. Hence, 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, therefore, 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 (l. 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.

D. Varying creation and wiring protocols

Swing provides distinct methods for creating widgets and wiring them to parents. Thus, Swing clients can dynamically create and modify their compositions. In contrast, SWT enforces the parent-child relationship on constructors. Thus, clients are more constrained.

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. 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. The surrogate constructors must create the adaptee and wire it to its parent.

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).

E. Inversion of control

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

For instance, our Swing application registers an `ActionListener` to a `JButton` (l.18-23), which corresponds to adding a `SelectionAdapter` to an SWT's `Button`. In Swing2SWT, hence, 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`. SWT2Swing, naturally, will perform the opposite mapping.

In general, applications may override any visible instance method of the source API—giving rise to a major challenge. A relatively regular case is when the overridden method has a counterpart in the target API. In this case, wrappers must intercept calls to the method of the target API by some means, e.g., by using instrumented subclasses of the target

API, delegate to the overriding method of the subclass of the surrogate. In this paper, we only deal with the listener scenario.

F. Correspondence of object identities

Surrogates may need to wrap results that are returned by invocations of the target API. For instance, the surrogate method for `Container.getComponents()` (l. 38) calls the corresponding `Composite.getChildren()` on its adaptee and receives objects of target API types, which must be wrapped before returning to the application.

It can happen that the same target API object flows many times to the application. If a new surrogate is created each time, the application cannot determine that the apparently different objects in fact denote a single object [7]. Thus, it is necessary to manage the surrogate identities of objects of the target 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 just by the type of the object from the target API.

G. Additional design challenges

We have identified additional challenges which are specific to the studied APIs. First, Swing and SWT approach thread handling differently. In Swing the application thread is free after creating the GUI, and a Swing-specific daemon thread maintains the windows and sends events to registered handlers. 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 that in mind. SWT2Swing, on the other hand, 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` (l.5), in SWT the information is passed as `LayoutData` to the `List` (l.6). Because layouts are a well modularized concern of the APIs, the studied wrappers re-implemented the source API layouts completely, without trying to reuse target API layouts.

V. API Wrapping Design Patterns

We now present the patterns identified in the studied projects. Each pattern addresses at least one of the challenges described before. We abstracted the following patterns:

- *Layered Adapter* decomposes surrogates into layers, providing flexibility to implement mappings with Alternative and Composite Targets (c.f., Table II and §IV-B).
- *Stateful Adapter* keeps additional state in an adapter to re-implement features missing in the adaptee or to adjust differences in interaction protocols (c.f., the discussion of `Shell`'s `disposed` flag in §IV-B).

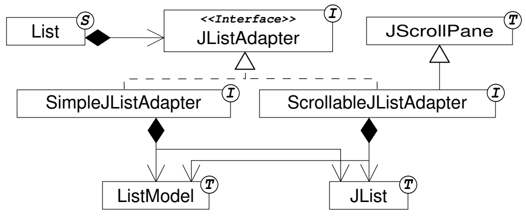

Fig. 5. *Layered Adapter* for Alternative and Composite targets.

- *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 §IV-D.
- *Inverse Delegation* provides adapters in the inverse direction to delegate target API events to the application—thereby addressing the challenge of §IV-E.
- *Wrapping Identity Map* maintains the correspondence between adaptees and surrogates in an identity map—thereby addressing the challenge of §IV-F.

A. Layered Adapter

The classic *ADAPTER* pattern can be directly applied to *Single Target* mappings, as defined in Table II. *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[2] in Fig. 5 shows the general structure of the pattern in the context of SWT2Swing's `List` surrogate implementation. Instead of directly referencing target 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, `SimpleJListAdapter` composes `JList` and `ListModel` whereas `ScrollableJListAdapter` additionally inherits `JScrollPane`. Note how this structure naturally represents the mapping definition for `List`, given as a regular expression in §IV-B.

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

Figure 6 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. The second scenario comes from XOM2JDOM and

[2]**Legend: class diagrams throughout the paper identify Ⓢurrogate, Ⓘnternal, Ⓣarget API and Ⓐpplication types.**

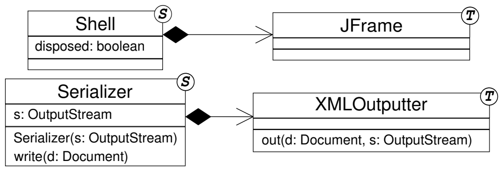

Fig. 6. *Stateful Wrapper* scenarios.

```
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 }
```

Fig. 7. Delayed Instantiation.

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.

C. Delayed Instantiation

Surrogates usually have their lifecycles bound to adaptees. When migrating to target APIs that enforce object relationships at construction time, surrogates may need to delay construction of adaptees until the relationship is established in terms of source API protocols. For example, in the Swing2SWT excerpt of Fig. 7, `JButton` cannot create its `Button` adaptee in a constructor because the parent object is yet unknown – the parent-child relationship will only be established when the `JButton` object is added to a `Container` with `Container.add` (l. 16).

The *FACTORY METHOD* pattern [5] provides infrastructure for delayed instantiation since it decouples surrogate and adaptee lifecycles. In the example, `Component` declares the `createAdaptee()` factory method (l. 3) to be implemented by sub-classes. While no adaptee is available, operations on surrogates must be deferred or re-implemented without delegation. For example, `JButton` stores interim data in a field `text` (l. 24) and re-implements its accessor methods (l. 30). The factory method then creates the adaptee (l. 25) and ensures interim data is delegated (l. 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 (l. 18), otherwise, the object is added to a list of child components `pending` adaptee instantiation (l. 17). Eventually a component will be added to a top-level widget, which always has an adaptee, which is when `wire` (l. 12) will be executed. This will trigger the factory method on the component, passing the parent widget's adaptee as parameter (l. 13). If the component is a container it may have pending children, so `createPending()` is called (l. 14) to iterate the list and cascade wiring (l. 9) of children.

Implementations of delayed instantiation must decide how to store interim data and which operations to re-implement or delay. In SwingWT, interim data is stored directly in surrogate fields, as in our example; getters and setters are re-implemented using interim data, whereas most of the remaining methods require proper adaptees.

In this solution, surrogate methods must always check if the adaptee has already been instantiated to decide if the operation should be delegated or simulated with the placeholder fields (e.g. `setText()` in l. 30). Another option would be to use the layered adapter pattern. An internal adapter variant would implement a placeholder and the surrogate would swap it to a final variant once the parent widget is known. This alternative has the advantage that surrogates delegate to the uniform internal adapter interface without having to check if the adaptee has already been instantiated.

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 l. 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. In contrast, factory methods can be properly overridden so that multiple instantiations are avoided.

Re-implementation of all operations of a surrogate with delayed instantiation ultimately violates the intention of reuse. Wrappers should re-implement or delay those operations that are often performed before the trigger in applications. Also, wrappers should provide documentation regarding the operations that are 'safe' in this sense.

D. Inverse Delegation

This pattern deals with control flowing from the target API to the application. In particular, *Inverse Delegation* allows application callbacks designed to work with the

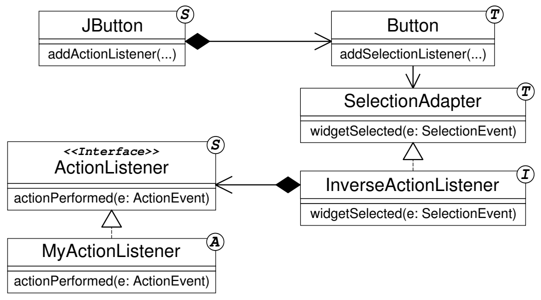
JButton
S
addActionListener(...)
Button
T
addSelectionListener(...)
SelectionAdapter
T
widgetSelected(e: SelectionEvent)
<<Interface>>
ActionListener
S
actionPerformed(e: ActionEvent)
InverseActionListener
I
widgetSelected(e: SelectionEvent)
MyActionListener
A
actionPerformed(e: ActionEvent)

Fig. 8. *Inverse Delegation* for callbacks.

source API to be executed when the corresponding events occur in the target 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 Fig. 8. When `MyActionListener` is registered to a surrogate (such as `JButton`), the surrogate creates an equivalent listener of the target API (`InverseActionListener`), which becomes an inverse adapter to the original listener. When the target API generates an event in the `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 Fig. 8, `JButton` would register itself to `Button` and 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`.

E. Wrapping Identity Map

The semantics of some source API operations is to return a new object at each call. For instance, the `elements()` method in Fig. 1 returns a new `Enumeration` object each time. In this case, a surrogate can simply wrap target API objects into new surrogates before returning to the application. For other operations it is necessary to maintain the correspondence between adaptees and surrogates. For example, in Fig. 9, `Container.getComponents()` (l. 12) delegates the call to its adaptee (l. 13) and gets an array of sub-widgets. It then iterates over the array wrapping the target API objects into corresponding surrogates (l. 16). `Component.getSurrogate()` (l. 3) consults a *Wrapping Identity Map* (l. 2) to get the `Component` surrogate mapped to the `Widget` adaptee. If the adaptee does not have a surrogate yet a new surrogate is created on the fly (l. 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, like having a default constructor and setters to adaptees) or implement a hard-coded type mapping. Furthermore, because the operation requests a surrogate of a certain type (`Component` in l. 16), it is possible to have an adaptee mapped to different surrogates depending on the context, which is necessary when many

```
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) getAdaptee()).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 }
```

Fig. 9. Identity Map.

Wrapper	**SwingWT**	**SWTSwing**	**XOM2JDOM**
Surrogates	471	248	38
Interfaces	120	46	0
Classic Adapters	9	22	30
Layered Adapters	64	54	1
Remakes	278	127	7
Internal Types	146	412	4
Internal Adapters	87	227	3
Anonymous Adapters	63	120	0
Stateful Adapter	98	181	13
Delayed Instantiation	✔	✘	✘
Inverse Delegation	20	59	1
Adapted Target API Types	20	26	1
Wrapping Identity Map	✔	✔	✔
Class Adapters	23	179	1
Object Adapters	103	124	31
Other Adapters	57	52	3
Single Adaptee	21	45	19
Multiple Adaptees	78	79	12

TABLE III
WRAPPER METRICS FOR TYPES, PATTERNS, AND ADAPTERS.

surrogates share a single adaptee (the map in l. 2 would need to be extended to account for multiple possible targets).

The identity map is only necessary 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, i.e., garbage collection should not be prevented from cleaning up objects only because they are in the map.

VI. WRAPPER METRICS

We designed straightforward metrics to provide evidence for the presence of the wrapper patterns in the studied projects. The metrics are described informally in the next sections, and the results of the measurements are presented in Table III. We also measure auxiliary properties of types and adapters.

A. *Classic vs. layered adapters*

A type present in a wrapper project can be either a *surrogate*, when it represents a type of the source API, or an *internal type*; see the breakdown of types at the top of the table. A surrogate can be one of four kinds: an *interface*, which is simply a copy of a source API interface; a *remake*, which is a class that re-implements a source API's type without any reference to the target API (neither directly nor indirectly through internal types); a *classic adapter*, which is a class that references the target API directly but not indirectly; a *layered adapter*, which references the target API indirectly through internal types.

Internal types can provide services to the wrapper, such as `ListTargetAdapter` in Fig. 5, 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 not surprising given that 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. Most of SWTSwing implements the layered adapter as described in Fig. 5 whereas SwingWT uses a simple 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 and adapters than SwingWT. Since XOM2JDOM maps structurally similar APIs, it relies mostly on classic adapters.

B. *Breakdown of other patterns*

The **Stateful Adapter** pattern is frequently used in the studied projects. We have counted all classic adapters, layered adapters, and internal types that carry additional data. We consider as additional data any non-constant field whose type is neither of a source API type (a relationship among surrogates) nor of a target API type (an adaptee). We exclude fields which are referenced in methods for delayed instantiation since they could represent interim data. Note that the defining characteristic of the stateful adapter pattern is that the target API cannot carry the data, whereas in delayed instantiation the target API can carry the data but is not ready to do so.

The **Delayed Instantiation** pattern is needed for one of the two GUI wrappers. That is, 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. XOM2JDOM does not need delayed instantiation because the lifecycles of surrogates and adaptees are bound. We should mention though that other XML wrappers may very well require delayed instantiation. For instance, a migration from XOM or JDOM to DOM would face the challenge that DOM enforces document-element relationships at construction time.

The **Inverse Delegation** pattern is exercised by all wrappers. 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 target API listeners (name ends with `Listener` or `Adapter`). XOM2JDOM implements a one-to-one inverse adapter approach for node factories used in de-serialization.

The **Wrapping Identity Map** pattern is exercised 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. XOM2JDOM uses a dedicated library that maintains a map per adaptee type and instantiates surrogates reflexively if necessary.

C. *Other adapter properties*

The table also summarizes how wrappers reference target API types: *class adapters* reference through inheritance; *object adapters* reference through object composition; *other adapters* reference in some other way, e.g., through calling target API static methods or having methods with target 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 listeners or part of a layered adapter pattern. This is also expected because Java does not allow multiple class inheritance and surrogates already implement the source API class hierarchy. Note that SWTSwing uses predominantly class adapters since the layered adapter pattern provides this flexibility.

Further, the table summarizes the multiplicity of field-based references for adaptees in object adapters: we speak of a *single adaptee* if an adapter type has a single field with a target API type; if there is more than one, then we speak of *multiple adaptees*. Here, we count both immediate and inherited fields. We can see that many types use multiple adaptees.

VII. Threats to Validity

The main threat to *internal validity* is related to the identified challenges and patterns. Although we have been systematic in analyzing the projects it can be that important challenges and patterns were missed. The data regarding the presence of patterns in wrappers were extracted by custom made tools and may present small deviations. Since 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 and involve our choice of subjects. We think that the challenges and patterns identified are not dependent on the specific APIs and could be generalized to other settings. However, since we have only studied GUI and XML APIs for Java our conclusions are restricted to these domains. Furthermore, the quality of the selected wrappers could influence the validity of the identified 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 source APIs, and the extensive user base. XOM2JDOM, which was developed by the authors, also achieves high compliance levels with its source API [4].

VIII. Related Work

Wrapping is a general re-engineering technique to provide access to existing functionality through a preferred interface. Wrapping can serve different purposes, e.g., the migration of old procedural code to OO abstractions [1], or the migration of non-service-oriented functionality, or systems, e.g., a GUI-based system, to services [8], [9], [10]. Our work contributes to 'the art of wrapping' with substantially advanced forms of the ADAPTER design pattern. We address layered adaptation, delayed adaptee instantiation, and other peculiarities.

API upgrade refers to the re-engineering objective of migrating applications across API versions. Both wrapping-based [2], [3] and transformation-based [11], [12] approaches have been studied. In such work, the involved APIs (in fact, versions) are related to each other through refactorings, which are then used to generate wrappers or to rewrite the application. In contrast, our work addresses the complexities of migration across different APIs, and it captures executable assumptions about their correspondence as advanced wrappers.

Transformation-based API migration has been studied both for the procedural paradigm [13], [14] and the OO paradigm [6], [15]. Previous (OO-based) work assumes relatively simple direct correspondences between types and methods of the API couple. In contrast, our work specifically studies the peculiarities of API source-to-target mappings that involve statefulness, delayed instantiation, or alternatives and composition (thereby leading to layered adaptation).

Language conversion involves API migration. Previous research mainly assumes a non-OO language (Cobol, C, etc.) on the source side [16], [17], [18]. Our work is the first to tackle current OO APIs with their type hierarchies and protocols. A notable but limited exception is the approach of [19]: simple API mappings, which can be useful for actual API migration, are inferred from an application before and after language conversion by matching code structure and names.

IX. Concluding Remarks

We have studied the design of two complex, practically relevant wrappers for the major GUI APIs of the Java platform. We have identified the design challenges that the developers of these wrappers and likely developers of other wrappers face. We have extracted solutions to these design challenges (problems) and abstracted them as design patterns.

Given the apparent complexity of wrappers in practice, the large number of APIs that play a role in current software development and the need to maintain and integrate applications, it appears to be a laudable objective to work towards generative support for wrapper development or, more generally, API migration. Compared to the transformation-based approaches of the related work discussion, we would like to specifically see that the identified patterns be addressed.

Another subject for future work concerns the semi-automatic identification of an API mapping by a combination of automated matching (as in [20]) and IDE-supported developer gestures. As far as matching is concerned, there is a considerable body of related research in the data management and meta-modeling communities.

References

[1] H. M. Sneed and R. Majnar, "A case study in software wrapping," in *Proc. of the International Conference on Software Maintenance (ICSM)*. IEEE, 1998, pp. 86–93.

[2] I. Şavga, M. Rudolf, S. Götz, and U. Aßmann, "Practical refactoring-based framework upgrade," in *Proc. of the Int. Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2008, pp. 171–180.

[3] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "Reba: refactoring-aware binary adaptation of evolving libraries," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*. ACM, 2008, pp. 441–450.

[4] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm, "Study of an API migration for two XML APIs," in *Proc. of the Int. Conf. on Software Language Engineering (SLE)*, 2009.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proc. of the Int. Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOSPLA)*. ACM, 2005, pp. 265–279.

[7] U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages," in *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1993, pp. 36–56.

[8] M. Li, B. Yu, M. Qi, and N. Antonopoulos, "Automatically wrapping legacy software into services: A grid case study," *Peer-to-Peer Networking and Applications*, vol. 1, no. 2, pp. 139–147, 2008.

[9] B. Zhang, L. Bao, R. Zhou, S. Hu, and P. Chen, "A Black-Box strategy to migrate GUI-Based legacy systems to web services," in *Int. Symp. on Service-Oriented System Engineering (SOSE)*. IEEE, 2008, pp. 25–31.

[10] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, "A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures," *Journal of Systems and Software*, vol. 81, no. 4, pp. 463–480, April 2008.

[11] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*. ACM, 2005, pp. 274–283.

[12] J. H. Perkins, "Automatically generating refactorings to support API evolution," in *Proc. of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2005, pp. 111–114.

[13] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *Proc. of the Int. Conf. on Software Maintenance (ICSM)*. IEEE, 1996, p. 359.

[14] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Proc. of the EuroSys Conference*. ACM, 2008, pp. 247–260.

[15] M. Nita and D. Notkin, "Using Twinning to Adapt Programs to Alternative APIs," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*, 2010.

[16] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos, "Code migration through transformations: an experience report," in *Proc. of the Conference of the Centre for Advanced Studies (CASCON)*. IBM, 1998, p. 13.

[17] A. J. Malton, "The Software Migration Barbell," in *ASERC Workshop on Software Architecture*, August 2001.

[18] J. Martin and H. Muller, "C to java migration experiences," in *European Conference on Software Maintenance and Reengineering*, 2002, pp. 143–153.

[19] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API Mapping for Language Migration," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*, 2010.

[20] D. Ratiu, M. Feilkas, F. Deissenboeck, J. Jürjens, and R. Marinescu, "Towards a Repository of Common Programming Technologies Knowledge," in *Proc. of the Int. Workshop on Semantic Technologies in System Maintenance (STSM)*, 2008.