Combining Multiple Dimensions of Knowledge in API Migration

Thiago Tonelli Bartolomei[1], Mahdi Derakhshanmanesh[2], Andreas Fuhr[2], Peter Koch[2],
Mathias Konrath[2], Ralf Lämmel[2], and Heiko Winnebeck[2]
[1] *University of Waterloo, Canada*
[2] *University of Koblenz-Landau, Germany*

***Abstract*—We combine multiple dimensions of knowledge about APIs so that we can support API migration by wrapping or transformation in new ways. That is, we assess wrapper-based API re-implementations and provide guidance for migrating API methods. We demonstrate our approach with two major GUI APIs for the Java platform and two wrapper-based re-implementations for migrating between the GUI APIs.**

***Keywords*-Software migration, API migration, API analysis, Wrapping, Mining software repositories**

I. Introduction

API migration is a kind of software migration; it may be necessary to meet requirements for software modernization, application integration, and others. API migration is realized by wrapping or transformation. We refer to [1], [2], [3], [4], [5], [6], [7], [8] for recent work on the subject.

For instance, consider the following re-engineering scenario. Two Java applications need to be integrated, but they use different GUI APIs, say SWING and SWT. Based on the exercised features and possibly other considerations, one of the two APIs is favored for the integrated application. The disfavored API (the "source API") can be re-implemented in terms of the favored API (the "target API") as a *wrapper* so that the migration requires little, if any, rewriting of the application's code. Incidentally, there are two advanced open-source wrappers that serve both directions of migration: SWINGWT[1] and SWTSWING[2].

In previous work [6], [8], we substantiated that migration between independently developed source and target APIs may be complex because of significantly different generalization hierarchies, contracts, and protocols.

Contribution: In the present paper, we describe an approach for the combination of multiple dimensions of knowledge about APIs so that API migration can be supported in new ways. That is, we assess wrapper-based API re-implementations and provide guidance for migrating API methods. To this end, we leverage a model-based approach to the integration of knowledge about APIs into a repository for convenient use in declarative queries. Throughout the paper, we use the SWING/SWT APIs and the above-mentioned wrappers as subjects under study.

Road-map: Sec. II describes the integrated repository. Sec. III and Sec. IV cover different forms of supporting API migration. Related work is discussed in Sec. V, and the paper is concluded in Sec. VI. The paper and accompanying material are available online.[3]

[1] http://swingwt.sourceforge.net/: re-implements SWING in terms of SWT
[2] http://swtswing.sourceforge.net/: re-implements SWT in terms of SWING
[3] http://softlang.uni-koblenz.de/apirep/

Acknowledgement We are grateful to Daniel Ratiu for providing us with data related to the programming ontology of [9], [10]. We are also grateful to four anonymous MDSM 2011 reviewers for their excellent advice.

II. The integrated repository

We integrate three data sources with API knowledge into a repository. Let us describe those data sources, the metamodel of the integrated repository, and the repository technology as such.

A. Data sources

- **APIMODEL** (developed by the present authors)—a model of API implementations (including SWING, SWT, SWINGWT, SWTSWING) with an underlying metamodel that is a (very) limited Java metamodel for structural properties and calling relationships;
- **APIUSAGE** (developed by Lämmel et al. [11])—a fact base (say, database) with usage properties of 1476 open-source Java projects at SourceForge, in particular with facts for API method calls within the projects' code;
- **APILINKS** (developed by Ratiu et al. [9], [10])—an ontology for programming concepts that were extracted semi-automatically from APIs in different programming domains, complete with trace links between concepts and the API source-code elements from which they were derived.

The APIMODEL source contributes basic knowledge about types and methods of genuine API implementations, and their coverage by the typically incomplete wrapper-based re-implementations. The APIUSAGE source helps to assess, for example, the relevance of genuine methods that are not implemented in a wrapper. The APILINKS source helps to derive candidate classes and methods that could be used in a wrapper-based API re-implementation.

B. Metamodel of the repository

Fig. 1 shows the metamodel (a UML class diagram) of our integrated repository where metaclasses are tagged by data sources APIMODEL, APIUSAGE, and APILINKS. We must note that the metamodel does not cover all elements of the sources, but is streamlined to fit our objectives.

The metaclass *NamedElement* represents package-qualified names of packages, classes, and methods. Because of the composition relationships in the metamodel, *NamedElement*s are also qualified by the name of an API, in fact, by a particular implementation, which could be a genuine implementation or a wrapper-based re-implementation.

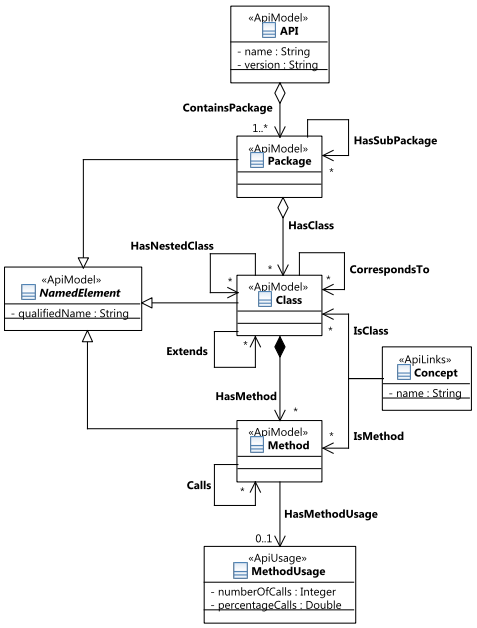

Figure 1. Metamodel of the integrated repository with API knowledge

The metaclasses *Package*, *Class*, and *Method* represent the package hierarchy with the Java classes and their methods, further with extension relationships between classes (see association *Extends*) and calling relationships between methods (see association *Calls*). As a means of prioritization, we leave out interfaces; they are trivially copied by wrappers.

Classes of genuine API implementations are linked with the corresponding classes of wrappers (see association *CorrespondsTo*). Here we note that wrappers may use different package prefixes. Also, these links improve convenience for those queries that need to navigate between the different API implementations. The metaclass *Concept* models concepts in the sense of APILINKS' ontology. Classes and methods can be linked with concepts; see associations *IsClass* and *IsMethod*. Hence, classes and methods of different APIs may be linked transitively.

The metaclass *MethodUsage* represents the usage data that was integrated from APIUSAGE. That is, for each API method, we maintain the *number of calls* to the method (if any) within the SourceForge projects covered by APIUSAGE [11]. We translated this number also into a relative measure in the sense of the *percentage of the calls* to the given method relative to the number of all calls to methods of the API.

C. Repository technology

The repository leverages the model-based TGraph approach [12]. The metamodel of Fig. 1 is represented as a TGraph schema; converters instantiate the schema from the different data sources. All analysis is performed by means of queries on TGraphs using the language *GReQL* (Graph Repository Query Language) [13]. For brevity, we describe all queries ("measurements") only informally in this paper, but here is a simple, illustrative GReQL example for retrieving all classes c of an API a that are not implemented by a wrapper:

```
using a:
from c: V{Class}
with c.qualifiedName =~ a and count(c-->{CorrespondsTo}) = 0
reportSet c
end
```

That is, a is an argument of the query for the name of the API; the query selects ("reports") all classes c such that the qualified name of c matches with a and there are no outgoing edges of the type *CorrespondsTo* (see `-->{CorrespondsTo}`) from c.

III. WRAPPER ASSESSMENT

Consider again our introductory scenario for API migration. Which wrapper, SWINGWT or SWTSWING, should we favor? Such decision making should take into account wrapper qualities, e.g., its completeness or compliance—both relative to the genuine API implementation. In case we want to improve a given wrapper, we should also track progress by simple metrics. Accordingly, we propose some concepts for wrapper assessment.

A. Coverage of source API

We can trivially compare the APIMODEL data between genuine API implementation and wrapper to get a basic sense of completeness in terms of (the percentage of) genuine packages, classes, and methods that are covered (say, re-implemented) by the wrapper. Table I collects such metrics for the SWING/SWT wrappers. The numbers show that the wrappers are highly incomplete.

	SWINGWT	SWTSWING
Packages	25 (78.12 %)	16 (51.61 %)
Classes	533 (18.61 %)	372 (56.97 %)
Methods	4533 (26.60 %)	3426 (42.59 %)

Table I
COVERAGE OF SOURCE API

B. Wrapper compliance issues

Some forms of non-compliance of a wrapper with the genuine API implementation can be determined by simple queries on our repository, e.g., differences regarding generalization hierarchies or the declaring classes for methods. Consider the following extension chain for SWING's *AbstractButton*:

```
java.lang.Object
 |_ java.awt.Component
     |_ java.awt.Container
         |_ javax.swing.JComponent
             |_ javax.swing.AbstractButton
```

The chain itself is preserved by SWINGWT. However, SWING declares the method *addActionListener* on the class *AbstractButton* whereas SWINGWT declares the method already on the class *Component*.

	SWINGWT	SWTSWING
• Declarations on supertypes	516	161
• Empty implementations	1006	230
• Missing methods	12506	4618
◦ Class missing	9604	3698
◦ Class present	2902	920

Table II
WRAPPER COMPLIANCE ISSUES

Table II shows numbers for some metrics for (lack of) wrapper compliance. In reference to the above example of the method *addActionListener*, we measure the number of methods that are declared "earlier" on a supertype in the wrapper. Further, we measure methods with empty implementations, i.e., implementations without any outgoing method calls, while the corresponding genuine implementations had outgoing method calls. (The substantial number of empty implementations may be surprising, but these wrappers are nevertheless reportedly useful in practice.) Finally, we also subdivide missing methods into those that are implied by missing classes vs. those that are missing from existing classes.

C. Relevance in terms of usage

Let us qualify wrapper (in-) completeness with APIUSAGE data. If the developers of the wrappers applied the right judgement call for leaving out classes and methods, then the missing methods should be less relevant in practice than the implemented ones. Table III lists usage metrics for the SWING/SWT wrappers.

	SWINGWT	SWTSWING
Unimplemented methods		
• Any usage	9,01 %	2,90 %
• Cumulative usage	2,88 %	2,35 %
Empty methods		
• Any usage	42,53 %	25,71 %
• Cumulative usage	11,41 %	1,49 %
Non-empty methods		
• Any usage	48,46 %	71,39 %
• Cumulative usage	85,72 %	96,17 %

Table III
USAGE OF API METHODS IN SOURCEFORGE

In the table, we break down SWING's and SWT's methods into categories according to the wrappers as follows: unimplemented, empty, and non-empty implemented methods. For each category, we show the percentage of methods with "any usage" (say, any calls) in the SourceForge projects in the scope of the APIUSAGE source. We also show "cumulative usage" for each category, i.e., the contribution of the category to all API method calls. These are contrasting numbers which show, for example, that the many unimplemented and empty methods (see again Table II) are exercised much less frequently than the fewer non-empty methods.

IV. GUIDANCE FOR MIGRATION

A given wrapper may be effectively incomplete in that a missing method is actually exercised by the application under API migration. In this case, we seek guidance for migrating the API method in question. Such guidance is universally useful for API migration—even when transformation is used instead of wrapping. A practical approach to guidance would need to combine elements of API type matching, IDE support (such as autocompletion and stub generation), and others. We focus here on the aspect of proposing method candidates to be called in methods of wrapper-based API re-implementations.

A. Concept-based method candidates

We can use APILINKS' trace links between API methods and concepts to propose method candidates. The idea is that if methods of the source and target APIs are related to the same concept, then the latter may be useful in re-implementing the former. Further, let us sort all such candidates by their cumulative usage, say, by their relevance as far as APIUSAGE is concerned.

Qualified candidate name	Cumulative usage (%)
swing.javax.swing.ImageIcon.ImageIcon	**0,4816**
swing.java.awt.image.BufferedImage.BufferedImage	0,1063
swing.java.awt.Frame.getIconImage	0,0059
swing.java.awt.....MemoryImageSource	0,0046
swing.java.awt.Frame.setIconImage	0,0042
swing.javax.swing.text.html.ImageView.ImageView	0,0005
swing.java.awt.....ImageGraphicAttribute	N/A

Table IV
CANDIDATES FOR RE-IMPLEMENTING SWT'S *Button.setImage*

Suppose you need to migrate SWT's *Button.setImage* to SWING. Table IV shows the method candidates that were automatically determined by a GReQL query. Consider the first line with the constructor of *ImageIcon*. We show the line in bold face to convey the fact that there is an existing wrapper, SWTSWING, whose method implementation of *setImage* readily involves the constructor of *ImageIcon*.

Further inspection reveals that SWING's *JButton*, which is a counterpart to SWT's *Button*, does not provide an *Image* property and, hence, we cannot simply migrate SWT's *Button.setImage* to a corresponding setter of SWING. Extra state and a more complex idiom (indeed involving *ImageIcon*) is needed.

B. Assessment of the ontology

The above example shows that APILINKS may suggest reasonable candidates—in principle. We would like to assess APILINKS's relevance more generally. In particular, we could compare APILINKS-based links with actual calling relationships in existing wrapper implementations, as they are available through APIMODEL's data. Table V lists corresponding metrics for the SWING/SWT wrappers.

	SWINGWT	SWTSWING
Unimplemented methods with links	10.83 %	0.35 %
Implemented methods with links	28.06 %	24.98 %
Correct links	42.75 %	37.20 %

Table V
API LINKS BETWEEN SWING AND SWT

The coverage of API parts by APILINKS' trace links is an artifact of the underlying semi-automatic ontology extraction approach [9], [10], which involves elements of name matching and thresholds for the inclusion of concepts. We cannot expect to retrieve links for arbitrary methods from APILINKS.

In the table, we break down SWING's and SWT's methods into the categories of unimplemented and implemented methods according to the wrappers. For both categories, we show the percentage of methods that are linked (transitively) with one or more methods of the

corresponding target API. The numbers are such that implemented methods happen to be much better linked than unimplemented ones.

At the bottom of the table, we also list the percentage of correct APILINKS' trace links. We say that a link from the method m of the source API s to a method m' of the target API t is correct, if a given wrapper-based re-implementation of s in terms of t implements m in a way that it directly calls m'. When we specify the percentage, we consider as the baseline (100%) only those methods m that both have associated trace links to t and actually call some method of t. It turns out that APILINKS predicts a correct link in more than 1/3 of the cases. We have to note though that APILINKS typically proposes multiple candidates—with a median of 8.

V. RELATED WORK

Work on API migration has previously focused on transformation and wrapper-generation techniques for API upgrades [2], [3], [4], [5] and, to a lesser extent, on migration between independently developed APIs [1], [6], [7], [8]. The present work is the first to integrate diverse data sources to assess wrappers and to guide their development. Typically, wrappers are assessed by *testing* (i.e., testing whether the application under migration continues to function, or recovers from any test failures that had to be addressed by improving a pre-existing wrapper) [6]. There is no previous work on guiding API-wrapper development for independently developed APIs.

Most of the techniques that we integrate are inspired by program comprehension research. For instance, our comparison of different API implementations is a simple form of object-model matching [14]. Also, our exploitation of API-usage data is straightforward, when compared to other scenarios of exploiting such data in the context of API usability [15] and understanding API usage (patterns) [16], [17]. Our proposal for guided migration can be viewed as one specific approach to advanced ("intelligent") code completion systems [18], [19].

VI. CONCLUDING REMARKS

The complexity of API migration requires many skills and techniques. Of course, one must understand the API's domain, and the application under migration. Basic software engineering skills such as testing, design by contract, effective use of documentation are critical as well. Still API migrations are largely unstructured today, and they come with unpredictable costs. We submit that techniques for assessment and guidance, such as those discussed in this short paper, are needed to tackle non-trivial API migrations in the future.

Clearly, our work is at an early state, and makes only a limited contribution to the larger API migration theme. There is a need for a comprehensive approach for guided API migration, which should combine diverse elements of assessment, mapping, matching, code completion, code generation, and testing.

REFERENCES

[1] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proc. of OOSPLA 2005*. ACM, 2005, pp. 265–279.

[2] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *Proc. of ICSE 2005*. ACM, 2005, pp. 274–283.

[3] 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.

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

[5] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: refactoring-aware binary adaptation of evolving libraries," in *Proc. of ICSE 2008*. ACM, 2008, pp. 441–450.

[6] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm, "Study of an API Migration for Two XML APIs," in *Proc. of Conference on Software Language Engineering (SLE 2009)*, ser. LNCS, vol. 5969. Springer, 2010, pp. 42–61.

[7] M. Nita and D. Notkin, "Using Twinning to Adapt Programs to Alternative APIs," in *Proc. of ICSE 2010*, 2010.

[8] T. T. Bartolomei, K. Czarnecki, and R. Lämmel, "Swing to SWT and Back: Patterns for API Migration by Wrapping," in *Proc. of ICSM 2010*. IEEE, 2010, 10 pages.

[9] D. Ratiu, M. Feilkas, and J. Jürjens, "Extracting Domain Ontologies from Domain Specific APIs," in *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, Proceedings*. IEEE, 2008, pp. 203–212.

[10] 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.

[11] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *SAC'11 - ACM 2011 SYMPOSIUM ON APPLIED COMPUTING, Technical Track on "Programming Languages"*, 2011, to appear.

[12] J. Ebert, V. Riediger, and A. Winter, "Graph Technology in Reverse Engineering: The TGraph Approach," in *WSR 2008*, ser. GI-EditionProceedings, vol. 126. Gesellschaft für Informatik, 2008, pp. 67–81.

[13] D. Bildhauer and J. Ebert, "Querying Software Abstraction Graphs," in *Query Technologies and Applications for Program Comprehension (QTAPC 2008), Workshop at ICPC 2008*, 2008.

[14] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Proceedings*. ACM, 2005, pp. 54–65.

[15] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving API documentation using usage information," in *Proc. of the 27th Intern. Conf. on Human Factors in Computing Systems, CHI 2009*. ACM, 2009, pp. 4429–4434.

[16] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," in *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), Proceedings*. IEEE, 2006, pp. 195–202.

[17] T. Xie and J. Pei, "MAPO: mining API usages from open source repositories," in *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 54–57.

[18] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005)*. ACM, 2005, pp. 48–61.

[19] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of ESEC/SIGSOFT FSE 2009*. ACM, 2009, pp. 213–222.