Bridging Java Annotations and UML Profiles with JUMP*

Alexander Bergmayr[1], Michael Grossniklaus[2], Manuel Wimmer[1], and Gerti Kappel[1]

[1] Vienna University of Technology, Austria
[lastname]@big.tuwien.ac.at
[2] University of Konstanz, Germany
michael.grossniklaus@uni-konstanz.de

Abstract. UML profiles support annotations at the modeling level. However, current modeling tools lack the capabilities to generate such annotations required for the programming level, which is desirable for reverse engineering and forward engineering scenarios. To overcome this shortcoming, we defined an effective conceptual mapping between Java annotations and UML profiles as a basis for implementing the *JUMP* tool. It automates the generation of profiles from annotation-based libraries and their application to generate profiled UML models. In this demonstration, we (i) compare our mapping with the different representational capabilities of current UML modeling tools, (ii) apply our tool to a model-based software modernization scenario, and (iii) evaluate its scalability with real-world libraries and applications.

1 Introduction

The value of UML profiles is a major ingredient for model-based software engineering [3] as they provide features supplementary to the UML metamodel in terms of lightweight extensions. This powerful capability of profiles can be employed as annotation mechanism [9], where stereotypes show similar capabilities as annotations in Java. In the ARTIST project [1], we exploit these capabilities, as we work towards a model-based engineering approach for modernizing applications by novel cloud offerings, which involves representing platform-specific models (PSM) that refer to the platform of existing applications, e.g., the Java Persistence API (JPA)[3], when considering persistence, and the platform of "cloudified" applications, e.g., Objectify[4], when considering cloud datastores. Clearly, the modernization process relies on the availability of the profiles that correspond to the used Java libraries.

Manually developing a rich set of profiles demands a huge effort when considering the large number of possible annotations in Java. To automate the generation of UML profiles requires an effective conceptual mapping of the two languages. In recent work, we defined such a mapping [2] based on which we implemented the *JUMP* tool. Hence, we continue with the long tradition of investigating mappings between Java and

* This work is co-funded by the European Commission, grant no. 317859.
[3] http://oracle.com/technetwork/java/javaee/
[4] https://code.google.com/p/objectify-appengine/

UML [7, 8], though in this work we also consider Java annotations in the mapping.

The *JUMP* tool is intended to be used by (i) developers that produce platform-specific profiles to support transformations for reverse engineering and forward engineering scenarios or to enable platform-independent profiles abstracted from such platform-specific profiles and (ii) modelers that directly use the produced profiles to document important design decisions at the modeling level or to easier understand Java libraries by visualizing provided annotations in terms of UML profile diagrams.

In this demonstration, we discuss the benefits of *JUMP* compared to existing solutions of modeling tools that support annotations. Furthermore, we emphasize the unique capabilities of the *JUMP* tool to automatically generate profiles from annotation-based libraries, which are collected in the *UML-Profile-Store*. It leverages the generation of profiled models from applications. To report on the scalability, we measured the performance of *JUMP* tool by applying it to large Java code bases.

2 Bridging Java Annotations and UML: Profiles to the Rescue

UML profiles enable systematically introducing new language elements [5] without the need to adapt the underlying modeling environment, such as editors, model transformations, and model APIs [6]. UML provides a dedicated language to precisely define profiles and how stereotypes are applied on models. Similarly, Java provides an annotation language to declare annotation types that can be applied on the targeted code elements. Figure 1 demonstrates the relationship between the two languages based on the Objectify framework. On the left side, the *application* of annotation types, among them `Cache`, to the `Customer` class and the respective *declaration* of the `Cache` annotation type is shown. The corresponding UML-based representation shown on the right side demonstrates the stereotype application to the `Customer` class and the declaration of `Cache` by a `Stereotype`, which is part of the Objectify profile. To ensure that the `Cache` stereotype provides at least similar capabilities as the corresponding annotation type, the extension relationship references the UML meta-class `Type`. The Objectify profile generated by the *JUMP* tool enables modelers to refine UML class di-

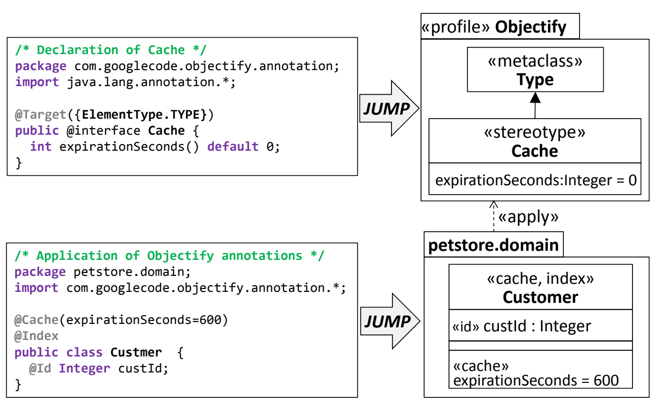

Fig. 1: JUMP in Action

Modeling Tool		Mapping (Java -> UML)		UML Profile Generation
Name	Ver	Annotation Application	Annotation Declaration	
Altova UML http://altova.com/umodel.html	2013	Generic Java Profile	Interface	-
ArgoUML http://argouml.tigris.org	0.34	Generic Java Profile	Interface	-
Enterprise Architect http://sparxsystems.com	9.3	Built-in Tool Feature	Interface	-
Magic Draw http://nomagic.com	17.0.4	Generic Java Profile	Interface	-
Rational Software Architect http://ibm.com/developerworks/rational	8.5.1	Specific Profiles	Stereotype	-
Visual Paradigm http://visual-paradigm.com	10.2	Built-in Tool Feature	Class	-
JUMP Tool	1.0.0	Specific Profiles	Stereotype	+

Table 1: Comparison of Modeling Tools

agrams towards the Google App Engine (GAE) and supports developers to realize code generators that produce richer program code.

Mapping Java Annotations to UML. Currently, three significantly different solutions exist to support Java annotations for UML models: (i) *built-in* annotation feature of modeling tools, (ii) *generic* profile for Java, which enables capturing annotations and their type declarations, and (iii), profiles which are *specific* to a Java library or even an application with custom annotation type declarations. The first solution is certainly the most tool specific one as it goes beyond Java and UML. It facilitates to capture Java annotations, though the type declaration of an annotation and its applications are not connected. A generic profile for Java emulates the representational capabilities of Java's annotation language. Although with this approach the connection of annotation type declarations and their applications can be ensured, the native support of UML for annotating elements with stereotypes is still neglected. However, stereotypes specifically defined for annotation types would facilitate their application in a controlled UML standard-compliant way as they extend only the required UML metaclasses. From a language engineering perspective, such stereotypes facilitate defining constraints and model operations, such as model analysis or transformations, because they can directly be used in terms of explicit types similar to a metaclass in UML. Therefore, the *JUMP* tool is based on a mapping between Java's annotation language and UML's profile language [2], which enables the generation of specific stereotypes for corresponding annotation types that in turn leverage platform-specific profiles.

Existing Modeling Tools. Several commercial and open-source modeling tools support Java annotations at the modeling level as summarized in Table 1. While all evaluated modeling tools support the generation of annotated UML class diagrams from Java applications, none of them is capable of generating profiles for Java libraries, and so exploiting the powerful capabilities of stereotypes and profiles.

3 JUMP Tool

The *JUMP* tool envisages two main scenarios: *UML Profile Generation* and *Profiled UML Model Generation*. The first scenario is executed on a Java-based Eclipse project that covers the library from which a profile with the corresponding stereotypes is generated. Optionally, the generated profile is added to a local copy of the *UML-Profile-*

Store, which exposes frequently used profiles as plug-ins to facilitate their reuse, thereby avoiding to regenerate them again and again. The practical application of profiles is employed in the second scenario, which is integrated into the generation of UML class diagrams from Java applications. Annotation applications are replaced by the corresponding stereotypes applied to the reverse-engineered UML elements. The applied stereotypes are imported from the *UML-Profile-Store*. If user-defined annotation types are declared in the application, the respective profile is generated in a pre-processing step as they need to be defined prior their application. Such application-specific profiles are provided together with the generated UML class diagram rather than added by default to the *UML-Profile-Store*. Similarly to the first scenario, the second scenario is also executed on Java-based Eclipse projects.

Prototypical Implementation. To realize *JUMP*, we developed three transformation chains, i.e., *JavaCode2UMLProfile*, *JavaCode2ProfiledUML*, and *ProfiledUML2-JavaCode*. For injecting Java code to our chains, we reuse MoDisco [4], which generates a Java model that is considered as input for the *JUMP* tool. Hence, it can be considered as a specific model discoverer to extract annotation types from Java libraries in terms of profiles. To generate Java code from such models we extended the code generator provided by Obeo Network[5]. The prototype and the collected profiles from 20 Java libraries with over 700 stereotypes are available at our project web site[6].

Scalability Evaluation.To report on the scalability of the *JUMP* tool, we measured the execution time of applying the *JavaCode2UMLProfile* and *JavaCode2ProfiledUML* transformation chain to real-world libraries and applications. For obtaining the measures, we executed them in Eclipse Kepler SR2 with Java 1.7 on commodity hardware: Intel Core i5-2520M CPU, 2.50 GHz, 8,00 GB RAM, Windows 7 Professional 64 Bit.

Table 2 summarizes our obtained results by emphasizing (i) the number of *code elements* in the intermediate Java model, (ii) the number of *declared* and *applied stereotypes* and (iii) the measured *execution times*. The rationale behind our selection of libraries (JPA, Objectify, Spring[7], and EclipseLink[8]) and applications (Petstore[9], DEWS-Core[10], Findbugs[11], and once more EclipseLink) is to consider small-sized to large-sized libraries and applications with varying number of declared and applied stereotypes. Clearly, the size of the input models passed to the transformation

Library	**Code Elements**	**Declared Stereotypes**	**Execution Time in Sec**
JPA[1]	20K	84	2.362
Objectify[2]	40K	18	1.842
Spring[6]	500K	63	10.292
EclipseLink[7]	700K	127	29.614
Application		**Applied Stereotypes**	
Petstore[8]	10K	287 (12 Profiles)	4.581
DEWS-Core[9]	30K	253 (2 Profiles)	3.116
Findbugs[10]	100K	1808 (3 Profiles)	26.620
EclipseLink	700K	7117 (3 Profiles)	199.028

Table 2: Performance Measures

[5] http://marketplace.eclipse.org/content/uml-java-generator
[6] http://code.google.com/a/eclipselabs.org/p/uml-profile-store
[7] http://projects.spring.io/spring-framework
[8] https://www.eclipse.org/eclipselink
[9] http://oracle.com/technetwork/java/index-136650.html
[10] Distant Early Warning System (DEWS), a use-case of the ARTIST project [1]
[11] http://findbugs.sourceforge.net

chains has a strong impact on the execution time of the *JUMP* tool as they are traversed throughout the generation of profiles and profiled models. Regarding the profile generation, the number of generated stereotypes is another main factor that impacts on the execution time. The more stereotypes are generated the more extensions to UML metaclasses need to be created. For instance, even though the JPA is compared to Objectify smaller in size the execution time is higher because a lot more transformation rules are applied when considering the number of declared stereotypes. Similarly, the number of applied stereotypes and their respective profiles impacts on the execution time. For instance, in the Petstore application, stereotypes are applied from 12 different profiles, which explains the higher execution time compared to DEWS-Core, even though the latter is larger in size. Finally, the execution time of generating profiled models is generally higher compared to profiles because the class structure of the former is much larger in size compared to the latter. For instance, considering EclipseLink and the number of generated stereotypes compared to classes the factor is almost 30.

4 Future Work

The *JUMP* tool aims at closing the gap between programming and modeling concerning annotation mechanisms. Still, open challenges remain to further integrate the two areas. For instance, with the latest Java version (1.8) we have repeating annotations that enable the same annotation to be repeated multiple times in one place which is currently not supported by stereotypes in UML. Furthermore, we plan to incorporate the production of UML activity diagrams for Java method implementations in JUMP in order to represent also annotation applications on the statement level in UML. Another line of research we plan to investigate is to further evaluate the JUMP tool in the context of the ARTIST project by empirical studies with our use case providers to determine the practical benefits for understanding and migrating legacy applications.

References

1. A. Bergmayr et al. Migrating Legacy Software to the Cloud with ARTIST. In *Proc. of CSMR*, pages 465–468, 2013.
2. A. Bergmayr, M. Grossniklaus, M. Wimmer, and G. Kappel. JUMP—From Java Annotations to UML Profiles. In *Proc. of MODELS*, 2014.
3. M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan&Claypool, 2012.
4. H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proc. of ASE*, pages 173–174, 2010.
5. L. Fuentes-Fernández and A. Vallecillo. An Introduction to UML Profiles. *European Journal for the Informatics Professional*, 5(2):5–13, 2004.
6. P. Langer, K. Wieland, M. Wimmer, and J. Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *JOT*, 11(1):1–29, 2012.
7. U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. In *Proc. of ICSE*, pages 742–745, 2000.
8. R. F. Paige and L. M. Rose. Lies, Damned Lies and UML2Java. *JOT*, 12(1), 2013.
9. B. Selic. The Less Well Known UML: A Short User Guide. In *Proc. of SFM*, pages 1–20, 2012.