A Case Study for Aspect-Based Updating

Susanne Cech Previtali and Thomas R. Gross

Department of Computer Science, ETH Zurich, Switzerland

Abstract. Rather than upgrading a software system to the next version by installing a new binary, software systems could be updated "on-the-fly" during their execution. We are developing a software evolution system that leverages aspect technology. As changes typically spread across several classes, we can handle updates like other crosscutting concerns: we encapsulate all changes, constituting a logical update, in one aspect. In this paper, we evaluate our approach. We report on a case study about the evolution of a Java application. The analysis provides details about how classes change between versions, and how these changes would be expressed in terms of updating aspects. Unfortunately, not all kinds of changes can be expressed using the aspect model. The results of our study, however, reveal that many changes fit our aspect-based approach.

1 Introduction

Dynamic software evolution represents an interesting technique to update software systems at run-time and is particularly helpful for systems that must be continuously available and up-to-date.

Our approach to the dynamic evolution of object-oriented software systems [1, 2] treats updates in a manner similar to crosscutting concerns in aspect-oriented programming: all changes that belong to a logical update are encapsulated in one aspect. We are developing a software evolution system that implements this idea. To compute the required updates, the system compares *statically* two complete versions of a Java program and deduces their structural differences. The structural differences constitute the individual changes. The system identifies the dependences between the changes and encapsulates these changes in an aspect. The dynamic aspect system PROSE [4–7] achieves dynamic software evolution by *dynamically* integrating the aspects.

In this paper, we report on a case study of an open-source Java program to determine if the evolution steps can be expressed as a sequence of updating aspects. The result of this case study reveals that—although not all evolution steps can be handled this way—most changes are limited to the implementation of methods and thus do not change the specification of classes. We show that many evolution steps can be decomposed and thus modularized, as indeed the actual changes concern "clusters" of few interacting classes.

The remainder of the paper is organized as follows: Sect. 2 explains the methodology of the case study. Sect. 3 presents the results of the evaluation and discusses the applicability of the updating model based on the results of the study. Sect. 4 concludes the paper.

2 Case Study

We have implemented a system to analyze compiled Java programs. All classes constituting the old and the new version of a program are compared to deduce the structural differences. First, classes, and recursively fields and methods, are matched as pairs to compute the sets of enduring, added, and removed entities. Second, the enduring entities are compared to compute the sets of unchanged and modified entities and the kinds of modifications. Based on the structural differences, the tool creates a method call-graph taking into account the static and dynamic target types and deduces the dependences between the changes. We refer to earlier work for a detailed description of the system architecture [1] and the corresponding algorithms [2]. Note that our current implementation matches two versions based on only the name and thus handles a rename as an removal and addition.

For the case study, we describe the evolution of a program from different points of view. First, we present the number of unchanged, modified, added, and removed classes. Then, we detail specific changes of the modified classes based on the actual modifications between two versions of an analyzed application. This data is based on the information available in the class file [3]: A **class** header stores the direct super-class and the interfaces a class implements, as well as the type parameters of generic classes. The header includes the access modifiers that determine whether a class is e.g., abstract, final, or synthetic. Furthermore, the header records the Java version. A **field** is characterized by its type, the access modifiers, the initial value, and generic parameters. A **method** is described by its body, the access modifiers, the return and argument types, exceptions, and generic parameters. Last, we discuss the updating aspects necessary to evolve the different versions.

3 Results

We have chosen Apache Tomcat 5.5, which implements version 2.4 of the Servlet and version 2.0 of the JSP specification, because it provides more than 20 releases. We downloaded the compiled releases of the "deployer" distribution and included all available Jar-files. **tomcat-5.5** initially consists of 399 classes, 1678 fields, and 3706 methods. In its latest release, **tomcat-5.5** includes 461 classes, 1902 fields, and 4348 methods.

The first part of Table 1 shows the coarse-grained evolution of **tomcat-5.5**. Mostly, classes are not changed between two versions except for release 5.5.1 when 60% of the existing 399 classes were modified. Only in six versions, classes were added. With three exceptions, classes are never removed. Fields are mostly stable, on average 99% are not changed. Only in nine versions, fields are modified; and only in eleven versions, fields are removed. In eleven versions, fields are added; in particular, in version 5.5.3, 177 fields are added. Similar to fields, methods are very stable. On average, only 1% of the methods changes, less than 1% are added or removed.

Table 1. Evolution of **tomcat-5.5**.

	Evolution												Modification													
	Classes				Fields				Methods				Classes					Fields			Methods					
	Unchanged	Modified	Added	Removed	Unchanged	Modified	Added	Removed	Unchanged	Modified	Added	Removed	Methods	Version	Fields	Super-class	Interfaces	Access	Type	Value	Body	Return type	Argument types	Access	Exceptions
5.5.0																									
→5.5.1	164	235	0	0	1676	1	3	1	3601	104	6	1	77	229	5	0	0	1	0	0	104	0	0	0	0
→5.5.2	384	15	0	0	1676	0	0	4	3668	41	1	2	15	0	3	0	0	0	0	0	41	0	0	0	0
→5.5.3	366	31	40	2	1640	0	177	36	3613	56	487	41	31	0	7	0	0	0	0	0	56	1	1	0	0
→5.5.4	422	15	0	0	1816	0	4	1	4114	38	5	4	15	0	2	0	0	0	0	0	38	0	0	0	0
→5.5.5	425	12	1	0	1819	1	1	0	4108	49	5	0	12	0	1	0	0	0	1	0	49	1	0	0	0
→5.5.6	430	8	0	0	1821	0	0	0	4133	29	0	0	8	0	0	0	0	0	0	0	29	0	0	0	0
→5.5.7	423	15	1	0	1820	0	10	1	4122	38	22	2	15	0	1	2	0	0	0	0	38	0	0	0	0
→5.5.8	420	19	0	0	1828	2	2	0	4138	44	2	0	18	0	3	0	0	0	2	0	44	1	0	0	0
→5.5.9	403	36	1	0	1816	13	19	3	4078	94	31	12	36	0	18	0	0	13	0	0	94	1	1	0	0
→5.5.10	413	25	1	2	1841	0	11	7	4136	61	15	6	25	0	5	0	0	0	0	0	60	1	0	0	0
→5.5.11	430	9	0	0	1852	0	0	0	4182	30	0	0	9	0	0	0	0	0	0	0	30	0	0	0	0
→5.5.12	417	17	17	5	1848	0	21	4	4159	41	61	12	17	0	2	0	0	0	0	0	41	0	0	1	0
→5.5.13	435	16	0	0	1868	1	0	0	4218	43	3	0	16	0	1	0	0	0	1	0	43	0	0	0	0
→5.5.14	442	9	0	0	1868	1	0	0	4230	34	0	0	9	0	1	0	0	0	1	0	34	0	0	0	0
→5.5.15	440	11	0	0	1868	1	1	0	4226	38	1	0	11	0	2	0	0	0	0	1	38	0	0	0	0
→5.5.16	440	11	0	0	1870	0	0	0	4232	33	0	0	11	0	0	0	0	0	0	0	33	0	0	0	0
→5.5.17	441	10	0	0	1870	0	0	0	4233	32	0	0	10	0	0	0	0	0	0	0	32	0	0	0	0
→5.5.20	439	12	0	0	1865	0	0	5	4227	38	0	0	12	0	1	0	0	0	0	0	38	0	0	0	0
→5.5.23	423	28	0	0	1857	4	3	4	4197	65	4	3	27	0	6	0	0	4	0	0	65	0	0	0	0
→5.5.25	431	20	0	0	1857	1	0	6	4214	50	0	2	20	0	4	0	0	1	0	0	50	0	1	0	0
→5.5.26	425	24	12	2	1846	0	56	12	4134	95	119	35	24	1	9	0	1	0	0	0	95	0	2	5	1

3.1 Modifications

In the following, we discuss the specific modifications shown in the second part of Table 1. We first present modified classes, then fields and methods.

Class modifications. The most frequent changes are method modifications. Only in version 5.5.1, the most frequent change regards the change of the Java version, when 60% of the classes were compiled from Java 1.2 to Java 1.4. The Java version (consisting of a major and minor version number in the class file header) defines the version of the class file format and consequently the minimal required Java virtual machine. A change in that version number may either reflect the migration of the Java development tools or the conscious usage of a new language feature. As we use the compiled application in bytecode form, we do not distinguish between the two cases. The inheritance structure of **tomcat-5.5** is very stable. Only in version 5.5.7, two classes extended different super-classes; and in version 5.5.26, one class removed an interface (i.e., java/lang/Serializable). Access modifiers and generic parameters are never changed (and are consequently omitted in the table).

Field modifications. In **tomcat-5.5**, fields are rarely changed. The access modifiers account for the most frequent change. This change consists of removing the modifier static (12 private fields and one protected in 5.5.9, two private fields in 5.5.23, one private field in 5.5.25). There is one change of the initial value in release 5.5.15. The few type changes refer to changes of a container type (i.e., java/util/Vector to java/util/List) or different representation (i.e., using a java/lang/ThreadLocal rather than a java/util/Hashtable keyed by thread-identifier for storing thread-local data).

Method modifications. The most prominent method change regards the change of the method body; in 99% only the body is changed. There are only a few changes of the return or argument types or access modifiers. These changes always imply an adaptation of the method body.

3.2 Updates

The updating approach cannot handle all kinds of changes. In such a case, the application cannot be updated at run-time and, as a consequence, must be restarted with the new version. For example, the updating model cannot update the super-class (one release in **tomcat-5.5**), as the aspect model does not provide a means to define such modifications. Furthermore, we cannot support changes to the Java version, as updated virtual machine is required to support new language features. As Java versions are released only every two years, this affects only one release in **tomcat-5.5**. Type changes require the adaptation of existing objects (four releases including a total of five changes). As a dynamic aspect system does not have the means to access each object, we again may choose to restart the application. As an alternative, we have described an extension to the aspect system using a copying garbage collector that could iterate over the object graph thereby transforming the objects [1].

Table 2 shows the aspects necessary to update the application. Column *Aspects* shows the number of aspects that contain the number of methods given in the first column. Column *Advised classes* shows the number of aspects that advise the number of classes given in the header, in relation to the total number of methods contained in the aspect. Column *Virtual methods* indicates the number of aspects that redefine the number of virtual methods given in the header, in relation to the total number of methods contained in the aspect. Overall, the table shows that most aspects encompass only a small number of classes and methods. Additionally, the number of virtual methods updated is small and explicit dispatching is therefore rarely required.

4 Concluding remarks

We analyzed more than 20 releases of **tomcat-5.5** that capture four years of its evolution. The results of this case study confirm our expectations: **tomcat-5.5** exposes fairly localized changes and thus allow the modular decomposition of

Table 2. Necessary updating aspects.

Methods	Aspects	Advised classes			Virtual methods			
		1	2	3	0	1	2	12
1	976	976	0	0	976	0	0	0
2	36	13	23	0	19	17	0	0
3	10	8	2	0	2	0	8	0
4	4	1	2	1	3	0	1	0
5	5	1	3	1	2	3	0	0
6	1	0	0	1	1	0	0	0
7	1	0	1	0	1	0	0	0
10	2	1	1	0	1	1	0	0
13	1	0	1	0	1	0	0	0
23	1	1	0	0	0	0	0	1
24	1	0	0	1	0	1	0	0

an update. There are various evolution steps the updating model can handle, and software developers may consider dynamic aspect-based updating as an alternative approach to achieve dynamic software evolution.

Acknowledgments. The work presented in this paper was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

1. S. Cech Previtali and T. R. Gross. Dynamic Updating of Software Systems Based on Aspects. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 83–92, 2006.
2. S. Cech Previtali and T. R. Gross. Extracting Updating Aspects from Version Differences. In *4th International Linking Aspect Technology and Evolution Workshop (LATE'08)*, 2008. *As the proceedings are not yet published, the paper is accessible at http://www.lst.inf.ethz.ch/research/publications/LATE_2008.html.*
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, 1999.
4. A. Nicoară and G. Alonso. Dynamic AOP with PROSE. In *International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05)*, pages 125–138, 2005.
5. A. Nicoară, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys'08)*, 2008.
6. A. Popovici, G. Alonso, and T. Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, 2003.
7. A. Popovici, T. R. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, 2002.