Characteristics of Runtime Program Evolution

Mario Pukall and Martin Kuhlemann

School of Computer Science, University of Magdeburg, Germany
{pukall, kuhlemann}@iti.cs.uni-magdeburg.de

Abstract. Applying changes to a program results typically in stopping the program execution. This is not acceptable for highly available applications. Such applications should be evolved at runtime. Because runtime program evolution is nontrivial we give terms and definitions which characterize this process. We will specify two major dimensions of runtime program evolution – time of evolution and types of evolution. To sketch the state of the art we will pick out three different approaches which try to deal with runtime program evolution.

1 Introduction

Nowadays requirements for complex applications rapidly change due to frequently altering environmental conditions. Attending new requirements using well-known software re-engineering techniques results the recurring schedule: stopping the application, applying the changes, testing the application, and restarting the changed application. Erlikh [1] and Moad [2] calculate the costs to maintain and evolve software to be 90 percent of the overall engineering costs. This is unacceptable for applications that should be highly available, e.g. security applications, web applications or banking systems, because unavailability causes costs. Idioms like design patterns, component-based approaches, and configurable applications help to reduce the costs of maintenance, evolution, and unavailability of applications. Nevertheless, these approaches cannot avoid unavailability of applications at all.

In this paper we mark characteristics of runtime program evolution based on object-oriented programming language Java. Based on these characteristics we evaluate existing approaches of runtime evolution.

2 Terms and Definitions

In this section we define terms which characterize runtime program evolution. We start at the categories *time* and *types* of evolution because these are major features of runtime evolution.

2.1 Time of Evolution

We found 3 different stages of program evolution (Figure 1 – *Build time*, *Hire time* and *Deployment time*). Build time references the process of static software

evolution where a software engineer implements the required changes in the program's source code. Building the program (i.e., source code compilation) terminates the process. *Hire time* is the time after the compilation of a class but before loading this class, e.g. before creating an instance of the class at all. *Deployment time* is the time after loading the code for execution, e.g. after class loading. Hire time and deployment time belong to runtime program evolution, whereas build time belongs to static program evolution.

2.2 Types of Evolution

We identified three fundamental types of runtime evolution: *predictability*, *kind of code* and *program changes*.

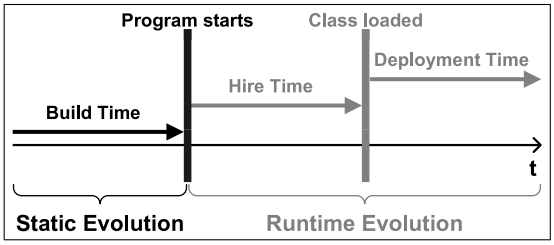

Fig. 1. Dimensions of Evolution Time.

Predictability. In some cases the application can be prepared for future program changes – so called *anticipated* program changes. We observed that the amount of program changes which cannot be foreseen (i.e. unanticipated program changes) is much bigger than the amount of predictable changes.

Kind of Code. Kind of code classifies the code which must be changed to achieve modifications into: *source code*, *byte code* (intermediate code of platform independent languages) and native *binary code* (directly executable by the host system).

Program Changes. Gustavsson and Assmann [3] identified two types of program changes: *source code* and *state* changes. In this work we concentrate on changes of program's source code because source code changes can also effect the program state. Additionally, program state changes can be prepared using interfaces and introducing the new state through, e.g. Java Remote Method Invocation (RMI) or Java Platform Debugger Architecture (JPDA).

In Table 1 we give our classification of program changes based on source code modifications. The classification respects major attributes of object-oriented paradigm and contains: changes that result in modified *structure*, *behavior*, *abstraction & encapsulation*, and *inheritance & polymorphism* of program's source code (which is organized in classes).

Due to space limitations we only depict a subset of possible source code changes. Each kind of source code change influences at least one category of our classification.[1] The benefit of our classification is a better highlighting of the approach's shortcomings concerning the feasibility of source code changes. We point out the applicability of our classification at byte code changes because byte code keeps the object-oriented paradigm.

[1] E.g., "modify method body" ↔ {behavior}, "add/remove class variable" ↔ {structure, encapsulation & abstraction}.

Examples	Categories of Source Code Changes			
	Structure	Behavior	Encapsulation & Abstraction	Inheritance & Polymorphism
add/remove class	yes	yes	yes	no
add/remove base/sub class	yes	yes	yes	yes
modify method body	no	yes	no	no
change method modifier	no	no	yes	optional
add/remove class variable	yes	no	yes	no
add/remove base/sub class variable	yes	no	yes	optional
add/remove class method	no	yes	yes	no
add/remove base/sub class method	no	yes	yes	optional

Table 1. Examples and Classification of Source Code Changes.

3 Evaluation

In this section we evaluate representative approaches which fit the topic of runtime evolution. Unless there is a huge amount of approaches we just have chosen these ones because they can be attached to different ideas of approximating runtime program evolution.

3.1 Javassist

Javassist (Java Programming Assistant) enables Java byte code changes [4, 5]. Using Javassist's source level API new or changed source code can be applied to existing class files without knowing the classes byte code. The byte code level API allows direct changes of class file contents.

Time of Evolution. Byte code changes can be executed until *hire time* (Figure 2). Byte code changes after class loading cannot be applied to the Java Virtual Machine (JVM).

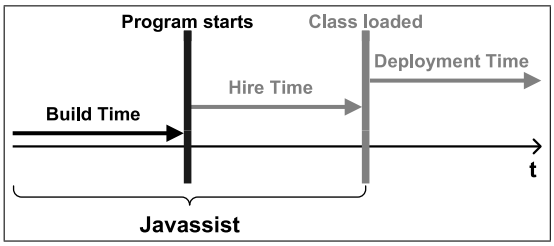

Fig. 2. Time of Evolution – Javassist.

Types of Evolution. Javassist enables all kinds of code changes[2], i.e. it covers all categories of our classification. This predestinates the tool for computing *unanticipated* program changes. Unfortunately Javassist lacks the complete bandwidth of time of evolution.

3.2 AspectWerkz

AspectWerkz [6, 7, 8] is an framework for aspect-oriented programming [9, 10] in Java.

[2] E.g., "add/remove class", "modify method body", "add/remove class method", etc.

Time of Evolution. AspectWerkz allows program changes until *deployment time* (Figure 3). The process of preparing the program for runtime evolution can only be performed until *hire time*.

Types of Evolution. Aspectwerkz offers two several concepts for modifying running applications – aspects and mixins – which enables unanticipated program changes.

Mixins introduce interfaces and there implementations to classes at hire time. To achieve this, methods (interface methods) and a field (instance of the interface implementing class) will be introduced into target class (processed at byte code level). These code changes cover all categories of our classification (Table 1).

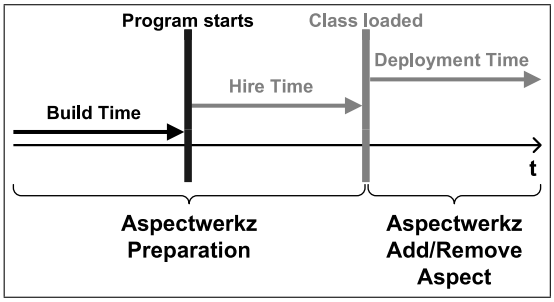

Fig. 3. Time of Evolution – Aspectwerkz.

The preparation for aspect deployment is processed at byte code level and results in modified method bodies. Aspects can be added and removed to (from) prepared program statements at deployment time. Such operations only change the program's state.

3.3 The Bean Scripting Framework – Java and JRuby in Concert

The *Bean Scripting Framework* (BSF) enables integration of scripting languages like JRuby[3] into Java programs [12]. The application is made up of the Java part (JP) and the JRuby part (SP). The interaction between the JP and the SP is controlled by a *BSFManager* which handles the scripting engine of JRuby. The BSFManager manages the execution of JRuby scripts and the interchange of object references[4] among the JP and the SP of the application.

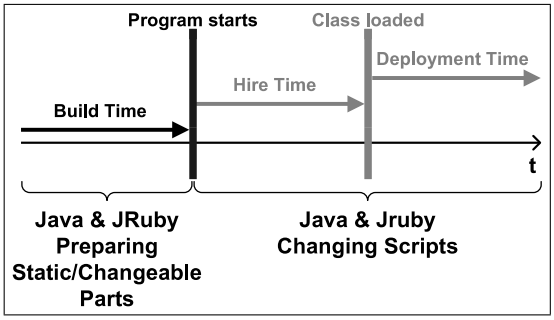

Fig. 4. Time of Evolution – Java and JRuby.

Time of Evolution. The combination of Java and JRuby enables program changes until deployment time (Figure 4). At build time it is terminated what is static (JP) and what is changeable (SP) in the application.

[3] Java implementation of scripting language Ruby[11]

[4] JRuby interpreter can execute scripts which contain Java source code.

Types of Evolution. Java classes defined within JRuby scripts can be redefined at deployment time. This influences all categories of source code changes (Table 1). Modifications can be applied to existing instances of the redefined class, i.e. unanticipated program changes are possible.

4 Conclusion

In this paper we characterized runtime evolution of programs. We identified two essential characteristics of runtime program evolution – time of evolution and types of evolution.

We reason that the combination of Java and scripting language JRuby using the Bean Scripting Framework offers the most suitable options for enabling runtime program evolution. JRuby scripts are interpreted and enable all kinds of program changes until deployment time. Nevertheless, because of the overhead of script interpretation at program's runtime this approach is not appropriate to time-critical use cases. New approaches are needed which enable runtime program evolution, as offered by interpreted languages, for compiled languages like Java and C++.

References

[1] L. Erlikh: Leveraging Legacy System Dollars for E-Business. IT Professional (2000)

[2] J. Moad: Maintaining the competitive edge. DATAMATION (1990)

[3] J. Gustavsson and U. Assmann: A Classification of Runtime Software Changes. In: Proceedings of the First International Workshop on Unanticipated Software Evolution (USE). (2002)

[4] S. Chiba and M. Nishizawa: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In: Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE). (2003)

[5] S. Chiba: Load-Time Structural Reflection in Java. Lecture Notes in Computer Science (2000)

[6] A. Vasseur: Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address It? In: DAW: Dynamic Aspects Workshop. (2004)

[7] J. Bonér: AspectWerkz – dynamic AOP for Java. Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)

[8] J. Bonér: What are the key issues for commercial AOP use: how does AspectWerkz address them? In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)

[9] G. Kiczales and J. Lamping and A. Mendhekar and C. Maeda, C.V. Lopes and J.-M. Loingtier and J. Irwin: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). (1997)

[10] K. Czarnecki and U. Eisenecker: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

[11] D. Thomas and C. Fowler and A. Hunt: Programming Ruby: The Pragmatic Programmers' Guide, Second Edition. Pragmatic Bookshelf (2004)

[12] V.J. Orlikowski: An Introduction to the Bean Scripting Framework. Presented at Conference ApacheCon US. (2002)