AJaTS: AspectJ Transformation System

Roberta Arcoverde
Informatics Center - UFPE
rla4@cin.ufpe.br

Sérgio Soares
Computing Systems Department - UPE
sergio@dsc.upe.br

Patrícia Lustosa
Informatics Center - UFPE
plvr@cin.ufpe.br

Paulo Borba
Informatics Center - UFPE
phmb@cin.ufpe.br

Abstract

The interest in aspect-oriented software development naturally demands tool support for both implementation and evolution of aspect-oriented software, as well as refactoring current object-oriented software to aspect-oriented. In this paper, we present AJaTS – a general purpose AspectJ Transformation System, that supports AspectJ code generation and transformation. AJaTS allows the definition of specific transformations, providing a simple template-based language, as well as pre-defined aspect-oriented refactorings.

Keywords Refactoring, Aspect-Oriented Programming, Code Generation

1. Introduction

Aspect-Oriented programming intends to increase software modularity, by separating the implementation of concerns which generally crosscut the system. Therefore, AOP addresses some object-oriented programming issues, like tangled and spread code, usually related to the implementation of transversal requirements. AspectJ [3], an aspect-oriented extension to Java [2], allows the definition of separated entities called aspects, which implement crosscutting concerns. This separation improves software quality, since it increases its modularity and reuse.

Due to its power and simplicity, the implementation of aspect-oriented systems with AspectJ is becoming each day more common. Tool support for AspectJ transformations has therefore become very important. However, there are still few tools that provide AspectJ programs generation and transformation, as well as refactoring's definition support.

In this paper, we present AJaTS – a general purpose AspectJ Transformation System, that supports AspectJ code generation and transformation. The main contribution of this paper is to present AJaTS's application value, describing its functionalities, use scenarios and examples of aspect-oriented refactorings supported.

Section 2 presents an introduction to the AJaTS engine, including its functionalities, template's language and application examples. Next, we discuss the AJaTS's architecture and technical points and Section 5 offers our concluding remarks.

2. AJaTS

AJaTS – AspectJ Transformation System – was conceived as a general purpose AspectJ Transformation System that supports AspectJ code generation and transformation. The main concept in AJaTS transformations is the capability of enable the user to specify templates for matching and code generation. Such templates are defined in a simple transformation language, similar to the target language. Such similarity makes AJaTS transformations easier to define and to understand. This feature allows the implementation of refactorings in a declarative way using a language, rather than hard coding refactorings in programs that manipulate AST or source code. This makes easier to write, to understand, and to evolve refactorings with AJaTS.

We show examples of both matching and generation templates below:

```
//matching template
public aspect #ASPECT_NAME { }
//generation template
public aspect #ASPECT_NAME {
    private String newField;
}
```

The matching template will match the source code, defining which classes/aspects will be transformed, as well as which structures will be saved in AJaTS variables. The generation template defines the transformation itself.

The basic constructs of the template's language are the AJaTS variables (i.e.: `#ASPECT_NAME`), used as information placeholders in a transformation. These variables have well defined types that can vary since a simple identifier until a whole set of methods of a class or aspect. The AJaTS variables are preceded by a '`#`' character. AJaTS template's language also offers more complex constructs, like conditional control (`#if, #else`) and loops (`forall`).

The AJaTS engine allows the user to define general transformation templates and applying them to any aspect-oriented project. Likewise, it also allows the generation of specific aspects, refactoring object-oriented software to aspect-oriented.

Besides allowing any developer to write their own transformation templates, AJaTS also brings some pre-defined useful transformations, which can be automatically applied to any Java/AspectJ project. One of these transformations is the Distribution Concern implementation [5]. It generates aspects that provide distribution, by modifying the system's façade, business entity classes and adding some auxiliary classes to the specified project. The details of this implementation are extensively explained elsewhere [5]. An example of how this transformation affects the system's code is shown above.

```
public class Facade {
  fds
  cds
  mds
}
//generated aspect
public aspect FacadeServerSideAspect {
  declare parents: Facade implements IFacade;
  declare parents : entities implements
    java.io.Serializable;
  ...
}
```

In this example, *entities* represents a list of business entity classes, automatically filled through user's input. This transformation example provide distribution through RMI, but it would be possible to use another distribution technology.

To make the facade instance remote, AJaTS generates an aspect called Server-side Aspect. It modifies the facade class (Facade) to implement the following remote interface (IFacade), also generated by AJaTS, which is demanded by the RMI API [7].

```
//generated interface
public interface IFacade implements
    java.rmi.Remote { mds' }
```

AJaTS also applies some pre-defined recommended refactorings to AspectJ code. The *Extract Pointcut* refactoring [4], for example, is demonstrated below.

```
//source code
aspect A {
    before() : exp { ... }
    after() : exp { ... }
}
//transformed code
aspect A {
    pointcut pc() : exp;
    before() : pc() { ... }
    after() : pc() { ... }
}
```

In this example, the pointcut pc is derived from the replicated expressions exp. All these transformations are implemented through templates, using the AJaTS template's language. The templates that perform these transformations are available at the project homepage (http://www.cin.ufpe.br/~jats/ajats).

Next section describes the architecture and implementation issues of AJaTS engine. It also presents the AJaTS plug-in, designed as an Eclipse IDE extension.

3. Architecture

The AJaTS Transformation Engine was conceived as an extension to a previously developed Java Transformation System, i.e., JaTS [1]. Whereas it reuses JaTS mechanisms to perform code generation and transformation, we still had to extend JaTS language and engine in order to support the manipulation of AspectJ code.

In this way, the JaTS parser had to be extended, including AspectJ syntax support. There were also included nodes to represent AspectJ constructs, and their respective meta-variables. These modifications allowed JaTS to create, identify and modify AspectJ syntax trees, performing transformations also in AspectJ programs.

In order to increase modularity and abstract JaTS's code modifications, AJaTS was designed as an aspect-oriented system itself. The visitors responsible for manipulating the AST, performing the engine operations, for example, were extended with methods inter-type declarations (an aspect-oriented construct), defined in separated aspects. Thus, we use AspectJ aspects to integrate AJaTS's code to the JaTS engine – making it easier to maintain. Figure 1 summarizes the AJaTS's extensions over JaTS's architecture: the addition of AspectJ nodes, and extension of the visitors and the parser.

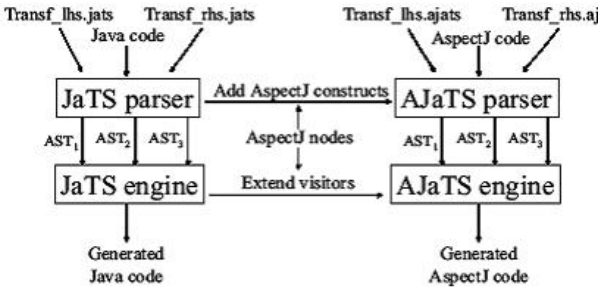

Figure 1 – JaTS x AJaTS architecture

We are currently improving an AJaTS Eclipse IDE plug-in. It integrates AJaTS main functionalities, such as refactorings definitions support, to Eclipse editor. This AJaTS implementation allows the application of its refactorings by code selection directly, using the Eclipse project explorer and the AspectJ editor provided by AJDT plug-in [6].

4. Conclusions

The elaboration of this work has shown some of AJaTS's limitations. Whereas it is clearly possible to define complex refactorings, they might require some extra processing, still not supported by the transformation engine itself. The *Extract Method Calls* [3], for example, is a well-known refactoring that involves Java code removal after its application. In order to realize it, several code comparisons are needed, which cannot be achieved with current's AJaTS version.

As a future work possibility, we propose an AJaTS improvement, which allows code analysis in a lower granularity level, to support the definition of such comparisons within the transformation templates. Another valuable contribution to this work is the implementation of a context-sensitive approach that allows the definition of much richer refactorings. Such approach is currently being developed.

Acknowledgments

We would like to thank the members of Software Productivity Group (www.cin.ufpe.br/spg) for all their technical contributions and support, in particular to Adeline Sousa. This research was partially sponsored by CNPq.

References

[1] F. Castor and P. Borba. A language for specifying Java transformations. In *V SBLP* , Brazil. May, 2001.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Java Series. Addison-Wesley, 2th Edition, 1996.

[3] G. Kiczales et al. Getting started with AspectJ. Communications of the ACM, 44(10):59-65, October 2001.

[4] R. Laddad. Aspect-Oriented Refactoring Series. TheServerSide.com, December 2003.

[5] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In Proceedings of OOPSLA 2002. ACM Press, 2002.

[6] AspectJ Development Tools. http://www.eclipse.org/ajdt/, 2007.

[7] Sun Microsystems. Java Remote Method Invocation (RMI).