AJaTS – AspectJ Transformation System: Tool Support for Aspect-Oriented Development and Refactoring

Roberta Arcoverde[1], Patrícia Lustosa[1], Adeline Sousa[1], Sérgio Soares[2], Paulo Borba[1]

[1]Informatics Center – Federal University of Pernambuco (CIn-UFPE)
Caixa Postal 7851– CEP 50732-970 – Recife, PE – Brazil
[2]Computing Systems Department – Pernambuco State University (DSC-UPE)
Rua Benfica, N 455, Madalena, CEP 51720-001 – Recife, PE – Brazil
{rla4,plvr,adss,phmb}@cin.ufpe.br, sergio@dsc.upe.br

Abstract. *The interest in aspect-oriented software development naturally demands tool support for both implementing and evolution of aspect-oriented applications, as well as refactoring current object-oriented software to aspect-oriented. In this paper, we present AJaTS – a general purpose AspectJ Transformation System for AspectJ code generation and transformation. AJaTS allows the specification of aspect-oriented refactorings in a templates-based language, syntatically similar to the AspectJ language. It also applies pre-defined recommended aspect-oriented refactorings and concerns implementation with aspects, such as distribution and persistence, increasing software development productivity.*

1. Introduction

Aspect-Oriented programming (AOP) intends to increase software modularity, by separating the implementation of concerns which generally crosscut the system. Therefore, AOP addresses some problems of object-oriented programming, 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 software modularity and reuse.

In order to improve software modularity and maintainability, the use of aspect-oriented programming has increased progressively. Tool support for many activities in aspect-oriented software life-cycle has been proposed, to facilitate their development. The AJDT Plug-in [6], for example, provides support for defining aspects in Eclipse IDE [4], as well as many integrated features, like cross references trees and crosscutting maps editor. Those features are very useful to implement, execute and debug AspectJ programs. Other systems, like MAJ – Meta AspectJ [9], focus on AspectJ programs generation, also combining the benefits of generative programming and AOP. However, their approach does not handle programs transformations, focusing on structured generation of AspectJ programs. The demand for tool support for code generation and transformation has motivated the creation of an AspectJ Transformation System called AJaTS.

AJaTS supports code transformation in a simple template-based language, similar to the AspectJ language. In this way, specific refactorings and transformation

can be defined, stored and reused anywhere. Due to its simplicity, AJaTS is being successfully used by real projects, like FLIP, a research project for extraction of product variations in the context of mobile games product lines [10].

The contributions of this tool are:

- Combining the benefits of aspect-oriented and generative programming, resulting in more productivity, modularity, and reuse;
- Automatize both concerns implementation and refactorings with AspectJ;
- Defining transformations in a simple template-based language, rather than hard-coding, through a declarative approach;
- An Eclipse plug-in allowing its widespread use for Java and AspectJ developers.

2. Functionalities

AJaTS transformations are written in a language that consists of AspectJ extended by AJaTS constructs, such as meta-variables and iterative and conditional structures. These constructs make program matching possible and allow the specification of new types that are to be generated (class, aspect or interface).

An AJaTS transformation consists of two parts: a left-hand side and a right-hand side. Both sides consist of one or more type declarations written in AJaTS language. The declarations in the left-hand side are matched with AspectJ source program that we wish to transform. The right-hand side is the skeleton of the program that will be produced by the transformation.

2.1. Features

The basic constructions in AJaTS are the AJaTS variables, used as information placeholders in a transformation. AJaTS variables consist of an AspectJ identifier preceded by the '#' character. These variables have well defined types that can vary from a simple identifier to a whole set of pointcuts of an aspect. For example, consider the following aspect declaration in the right-hand side of a transformation:

```
aspect #A {}
```

The variable *#A* is used as a placeholder for an aspect name, in such a way that the left-hand side of the transformation matches with any empty aspect.

There are some cases where it is not possible to determine the intended matching for a certain variable. In these cases, the user must declare the type of this variable in order to correctly specify the intended semantics of the transformation. For example, if we want a certain variable *#pcuts* to be matched with a set of pointcut declarations, *#pcuts* must be declared as being of type *PointcutDeclarationSet*.

The left-hand side of a transformation is matched with the source AspectJ type. A construction in the source AspectJ type matches the one in the left-hand side of a transformation if they are identical or if the second one corresponds to an AJaTS variable. This process derives a mapping from variables to the matched values.

The right-hand side of transformations can contain declarations that appear in the left-hand side and might have some additional fixed declarations. However, it is also

useful to have, on the right-hand side, declarations that use information from the original declarations, but are not necessarily identical to them. In order to support the extraction and modification of original declarations, AJaTS provides the so called executable declarations. They can appear anywhere a variable can, but only on the right-side of transformations. Executable declarations appear in AJaTS transformations enclosed by the "#<" and ">#" symbols. For example, in the transformation:

Left-Hand Side	**Right-Hand Side**
`public aspect #A {}`	`public aspect #< #A.addSuffix("Aspect") ># {}`

The construction `#< #A.addSuffix("Aspect") >#` is an executable declaration. This transformation applied to the aspect:

```
public aspect Visitor {}
```

results in the following AspectJ type:

```
public aspect VisitorAspect {}
```

AJaTS language also provides other complex constructs, like conditional and iterative declarations. As executable declarations, those can only appear in the right-hand side of a transformation.

A complete list of all available commands performed by AJaTS's executable declarations – such as the command `addSuffix` demonstrated above – can be found on AJaTS's website.

2.2. The AJaTS Plug-in

The AJaTS Plug-in has been designed to specify, to visualize, and to apply AJaTS's transformations in an intuitive and simple way. Moreover, the plugin was developed aiming at a major integration with Eclipse Platform [4], so that users that already use this IDE can easily work with AJaTS Plug-in.

In AJaTS's Plug-in, the transformations that are available for the user are shown in an Eclipse view. In this view, transformations are presented in a tree, with their respectives left and right-hand sides. These AJaTS templates can be written using an editor similar to the one used to write Java files.

AJaTS transformations are applied in the following steps: first, the user selects the 'Apply Transformation' option in the 'AJaTS' menu. It shows the dialog described in Figure 1. The user must specify where the generated/transformed resources must be saved (1), indicating a destination project and a destination package. After that, the user chooses which transformation he wants to apply (2). The left templates related to such transformation will appear on the 'Left Templates' list. The next step is to choose which resources will match each left template (3). If the matching suceeds, the 'Matched Templates' table will be filled with it (4). When all the left templates are correctly matched, the user will be able to execute the transformation, pushing the 'OK' button. The generated/transformed resources will then be stored in the specified location.

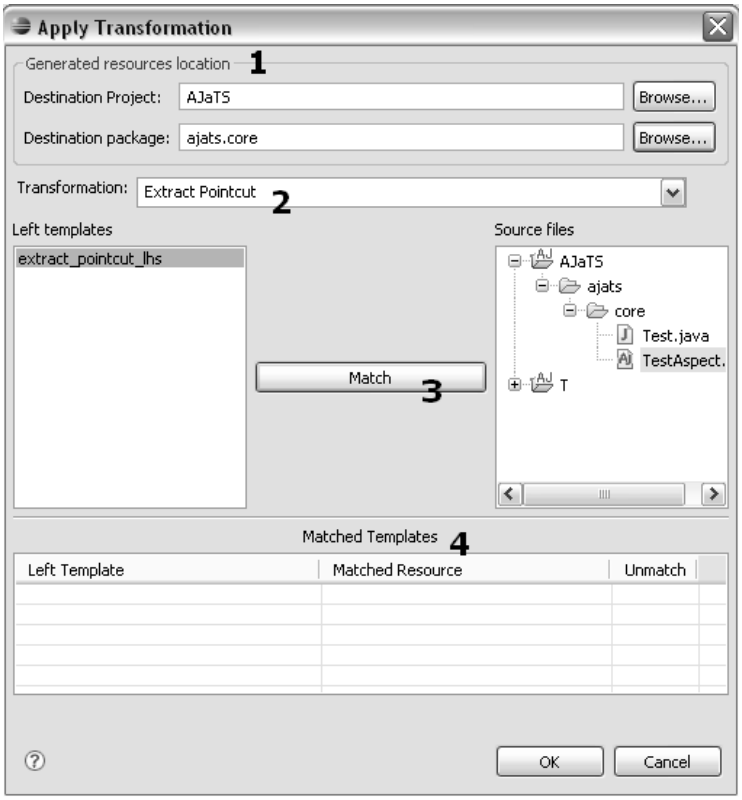

Figure 1. Apply Transformation Dialog

AJaTS's Plug-in offers some pre-defined recommended aspect-oriented refactorings, which can be applied to general project. Moving fields from classes to aspects inter-type declarations and replacing interface implementation with AspectJ's declare parents construct [7] are examples of such refactorings. Likewise, AJaTS also offers code generation for concerns implementation. The Distribution Concern transformation [8], for instance, generates aspects that provide distribution, by modifying the system's façade, business entity classes and adding some auxiliary classes to the specified project. This transformation currently uses RMI to provide distribution, but it would be possible to implement it with another technology, like EJB.

The AJaTS Plugin also allows the definition of new transformations, that can be stored and reused in any Eclipse Project. The templates can be edited in a customized Eclipse editor, with syntax highlighting, providing a friendly environment.

3. Implementation

AJaTS was conceived as an extension of a previously developed Java Transformation System – JaTS [1]. Whereas it reuses most of JaTS mechanisms to process code transformation, it still had to extend its language and engine in order to support the manipulation of AspectJ code.

The Eclipse IDE Plug-in presented in this paper uses the AJaTS transformation engine to perform code generation and transformation. Thus, we will explain both the engine and the plugin implementation issues in the following sections.

3.1. AJaTS Transformation Engine

The AJaTS engine has been implemented using Java Development Kit v1.4 and AspectJ v1.5. It has been constructed as an extension to JaTS, providing support to AspectJ transformations. In this way, the JaTS parser had to be extended, including AspectJ syntax support. There were also included nodes to represent AspectJ constructs. Furthermore, visitors responsible for manipulating the ASTs, performing engine operations (like replacement, evaluation and pretty-printing), were modified to include behaviour for AspectJ nodes.

In order to improve adherence to newly implemented versions of JaTS, the AJaTS engine was implemented as an aspect-oriented system itself. The visitors, for example, were extended using aspects with method inter-type declarations (an AspectJ construct), to avoid code insertion directly in JaTS's classes. Thus, we use aspects to integrate AJaTS's code to JaTS engine, making it easier to maintain. Likewise, every improvement made on JaTS is easily incorporated to AJaTS, since they can be glued without effort. Figure 2 summarizes how such integration work.

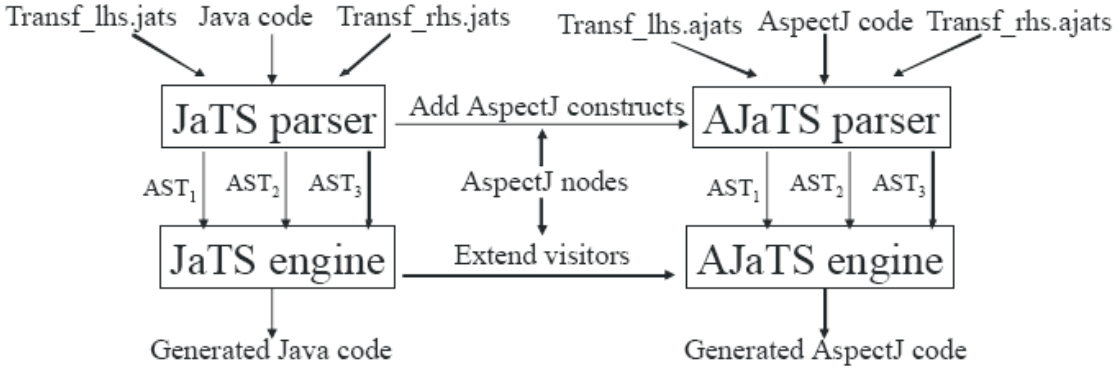

Figure 2. AJaTS extensions over JaTS

AJaTS implementation also involved using JaTS's generation templates. Thus, some AJaTS modules could be totally generated by JaTS, in a bootstrapping process. Most of nodes representing AspectJ constructs, for example, were automatically generated using generation templates. Since AspectJ is a superset of Java, AJaTS is a superset of JaTS. Any transformation supported by JaTS is therefore supported by AJaTS.

3.2. AJaTS Plug-in

The AJaTS plug-in integrates AJaTS features and functionalities to Eclipse IDE. It allows the user to create, store, apply and reuse transformation in any Eclipse project, whether object or aspect-oriented. In this section, we show how the plug-in was implemented, its architecture and technical decisions.

The AJaTS Plug-in is composed by two main parts: EclipseAdapter and AJaTSController. EclipseAdapter is the plug-in itself. It is responsible for preparing and organizing a transformation, since gathering user input data and accessing AJaTS engine, until storing the transformed code in the workbench. FileServices is the module responsible for file manipulation within Eclipse. In order to obtain better integration with Eclipse IDE, views and menus were extended.

AJaTS Controller represents the engine itself; it is responsible for interacting with both AJaTS parser and transformation mechanisms. AJaTSIO performs the parsing and converts syntax-trees in source code. The AJaTSEngine module applies the transformation. It can be divided in three subsystems: the Matcher tries to match the left-hand template's parse-tree with Java/AspectJ source code's parse-tree. It also generates a results map. The Replacer receives as input the parse-tree of the right-hand template and the result map previously generated. It is responsible for replacing occurences of variables in the template by the values mapped to them in result map. Finally, the Processor process the executable and iterative declarations.

4. Conclusion

In this paper we presented AJaTS – an AspectJ Transformation System. We discussed its main features and functionalities, as well as its architecture and application value. The current plug-in implementation allows defining, editing, storing and applying transformations to an AspectJ project, integrated to Eclipse IDE.

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. Context-sensitive refactorings, like Extract Method Calls[5], for instance, are still not supported under the declarative approach of AJaTS. 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 refactorings within the transformation templates.

More information, examples and AJaTS's download are available at http://www.cin.ufpe.br/~jats/ajats/.

References

[1]Castor, F., Oliveira, K., Sousa, A. and Santos, G. (2001) "JaTS: A Java Transformation System". In: *SBES 2001*, pages 374-379. Rio de Janeiro, Brazil.

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

[3]Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. (2001) "Getting started with AspectJ". Communications of the ACM, 44(10):59-65.

[4]Eclipse Foundation Inc., http://www.eclipse.org. Eclipse, 2007.

[5]Laddad, R. (2003) Aspect-Oriented Refactoring Series. TheServerSide.com.

[6]AJDT: AspectJ Development Tools. (2007) http://www.eclipse.org/ajdt/

[7]Monteiro, M. and Fernandes, J. (2005) "Towards a Catalog of Aspect-Oriented Refactorings". In: *AOSD 2005*, Chicago, USA. ACM Press.

[8]Soares, S., Laureano, E. and Borba, P. "Implementing Distribution and Persistence Aspects with AspectJ". (2002) In: *OOPSLA 2002*, pages 174-190. ACM Press.

[9]Zook, D., Huang, S. and Smaragdakis, Y. (2004) "Generating AspectJ Programs with Meta-AspectJ". In *GPCE 2004.* Volume 3286 of LNCS, pages 1-19.

[10]Alves, V., Matos, P., Cole, L., Borba, P. and Ramalho, G. (2005) "Extracting and Evolving Mobile Games Product Lines". In: *SPLC 2005*. Springer-Verlag.