Migrating a Static Analysis Tool to AspectJ™

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver BC Canada V6T 1Z4
{mrobilla,murphy}@cs.ubc.ca

November, 1999

OOPSLA '99 Workshop on Multi-dimensional Separation of Concerns in Object-oriented Systems.

1 Introduction

Software design and programming techniques that provide explicit support for separating concerns are intended to help developers more easily express and evolve software systems. The degree to which these techniques can aid a software developer is dependent on the difficulty of determining what the separable concerns are within a system.

To gain a sense of the kinds of concerns which might be useful to separate in a system, we examined the source for the Jex static analysis tool, which extracts exception information from Java files [5]. To gain a sense of the process involved in actually separating a concern from an existing code base, we migrated Jex's Java code base to AspectJ™ [7]. AspectJ provides aspect-oriented programming [3] support for Java.

This position paper describes this migration case study. Section 2 briefly describes the relevant elements of the architecture of Jex, and lists the aspects that have been used in the redesign. Section 3 describes the porting of the initial code to a version using aspects. Section 4 discusses the use of aspects in the design of object-oriented code.

2 Jex and Aspects

2.1 The Architecture of Jex

Jex is a tool for extracting exception information from Java source files [5]. It comprises 23 000 lines of commented Java source code spread over 138 classes.

The architecture of Jex consists of five components: the application controller, the parser, the abstract syntax tree (AST), the type system, and the Jex loader (Figure 1).

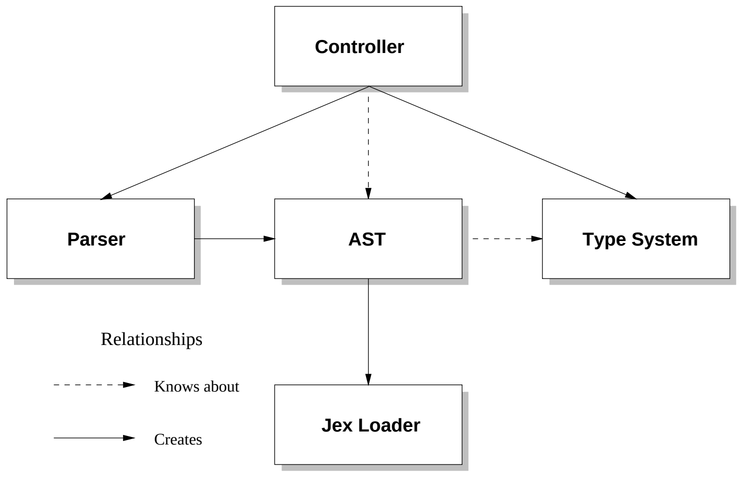

Figure 1: The Simplified Architecture of Jex.

The **application controller** is the entry point to Jex. It processes the command-line arguments, loads the type system, and invokes the **parser** on the input file, which returns a reference to an AST representing the input file. The controller then requests the AST to perform various functions, including exception analysis.

The **AST** contains functionality to analyze the type of expressions, to determine a call order within the methods of a class, to extract exception information about the program, and to generate an output file containing exception-related information. The **AST** component comprises 94 classes representing mostly the different node types.

The AST relies on the **type system** and on the **Jex loader** components to perform the task of exception information generation. It uses the type system to obtain a list of all implementations of a particular method. The Jex loader is used to determine the exceptions that can be raised by a Java method call.

All the modifications described in this document apply to the AST component.

2.2 Aspects Identified

In an attempt to make a cleaner design and to promote reuse, three main concerns were identified and introduced as aspects: `JexPathA`, `JexA` and `ControlFlowA`.[1]

[1] Although other aspects, such as debugging and error handling, were created as a part of this experiment, they are not described in this document because they are archetypal.

JexPathA is a global variable aspect. It introduces a variable, the Jex path, that is global to the application. JexA and ControlFlowA are decomposition aspects. They allow functionality to be factored out of the AST code.

3 Implementing Aspects

Modifications to the code of Jex release 1.0 were carried out using AspectJ version 0.3beta3. The following subsections describe the details of the changes to the code necessary to implement each aspect.

3.1 The JexPathA Aspect

During the generation of exception information, some nodes in the AST need to determine where to save newly created Jex information files. The directory in which to store these files is obtained as a command-line argument. Since only a very small subset of the AST nodes need to access this information, it was not deemed convenient to pass it along the call chain of the AST methods. Instead, in the initial design, the Jex directory was stored as a system property. Another alternative not using aspects would have been to store the value as a globally-available, class-scoped attribute.

In an attempt to provide a more robust solution for this problem, the aspect JexPathA was designed to store the Jex directory. The aspect contains a String attribute holding the value of the directory, and introduce weaves[2] for get and set operations on the attribute (Figure 2).

```
aspect JexA
{
   private static aJexPath = "";

   introduce public void Analyzer.setJexPath( String pJexPath )
   {
      aJexPath = pJexPath;
   }

   introduce public String ASTAllocationExpression.getJexPath(),
                           ASTConstructorDeclaration.getJexPath(),
                           ASTExplicitConstructorDeclaration.getJexPath(),
                           ASTMethodInvocationExpression.getJexPath()
   {
      return aJexPath;
   }
}
```

Figure 2: The JexPathA Aspect

To install this aspect in the Jex code, it was necessary to identify all statements that accessed the system property with the key identifying the Jex di-

[2]In AspectJ 0.4, weaves are called *crosscuts*. In this paper, we retain the original terminology and syntax of AspectJ 0.3beta3.

rectory. Occurrences of the `setProperty` and `getProperty` methods were then replaced by the equivalent method names used in the aspect, and a descriptor was added for the classes in the `JexPathA introduce` declarations.

The advantage of this approach compared to the system property or static class approaches is that it allows the programmer to statically restrict access to the globally available Jex directory value. In other words, the aspect explicitly names the classes that can write or read the directory value. This prevents accidental accesses to the value by other classes.

The disadvantage of using this approach is that it is not possible to compile the classes which use the `JexPathA` aspect without weaving them with the aspect. Of the three approaches implemented (system property, static class, aspect), the static class was the one retained, for both its simplicity and its non-intrusiveness.

3.2 The `JexA` Aspect

In the initial design of Jex, the various operations on the AST were coded directly in the AST nodes. General behavior was coded in the class `SimpleNode`, and specialized behavior was coded in subclasses of `SimpleNode` implementing specific node types, such as `Expression` or `Statement`. The operations defined in AST nodes include support for parser actions, type analysis, and exception information generation.

To enable the use of the AST as a stand-along framework for the analysis of Java programs, we wanted to factor out the exception generation functionality.

The exception generation functionality is distributed amongst three types of methods: `generateExceptions`, `buildExceptionStructure`, and `getExceptions`. The method `generateExceptions` is used to carry out the command to generate exception information from the root of the AST, the `CompilationUnit` node, to the individual class body declarations containing methods and constructors to analyze. The `buildExceptionStructure` method is used by every statement to add exception information to a dynamically-allocated structure describing the exception flow for every method. Finally, the `getExceptions` method is used to obtain the set of exceptions that can be raised by a particular expression.

These three methods form a phase of operations that is completely independent from other processing performed on the AST, such as resolving the type of expressions. These methods also access data in the AST nodes in a sequence that does not interfere with previous or future operations on the AST. It was thus relatively easy to create an aspect, `JexA`, containing all of the exception generation methods. Forming this aspect meant the methods left in the AST nodes supported generic analysis operations such as parser actions and type analysis.

The modifications to the code necessary to install this aspect were limited to moving exception generation methods from the AST node classes to the `JexA` aspect file, as `introduce` weaves.

There are many advantages to this decomposition. First, it is now easier to reuse the AST component for tasks other than exception analysis because it is not cluttered with exception generation operations. Second, the aspect provides a nice modular structure for adding features to the AST component. For example, to add the exception generation, it is now only necessary to "weave in" the exception aspect `JexA`.

One could argue that similar benefits can be obtained using the Visitor design pattern [1]. However, aspects are a more elegant and more flexible solution. More elegant because it is not necessary to have a huge Visitor interface including an `accept` method for all of the different types of nodes, and because the nodes do not have to implement the `accept` callback. More flexible because it can accommodate a call chain that does not necessarily follow the tree structure, and because the methods can return values, like in the case of the `getExceptions` methods. Furthermore, other advantages of the Visitor pattern, such as the accumulation of state, can also be implemented using aspects. Tarr et al. [6] describe the addition of feature to an AST using aspects in a similar manner.

3.3 The `ControlFlowA` Aspect

The exception generation operations, factored out as described in the previous section, rely on the knowledge of the total call order of the methods of a class being analyzed (this information was perhaps inappropriately named "Control Flow" in the design). In the original version of Jex, since such control flow information is only necessary when exception analysis is to be performed, the corresponding functionality was embedded in the exception generation code.

A goal of the redesign was to factor out the control flow functionality, in order to provide it as a separate AST service. In comparison to the exception generation aspect, factoring out the control flow required substantial redesign.

The redesign required modifying the class hierarchy of certain types of nodes, introducing a new class, and moving functionality between different classes (see Figures 3 and 4).

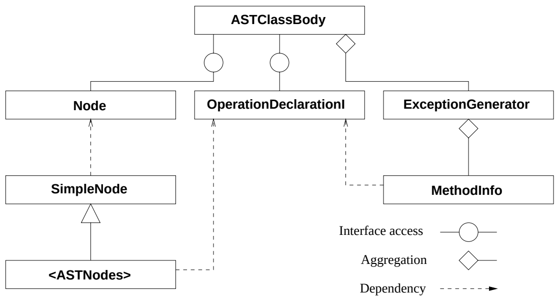

Figure 3: Initial Control-Flow Design

In the original design, once a `ClassBody` node received a `generateExceptions` message, it created an `ExceptionGenerator` object, which, in turn, created and initialized a data structure to hold control-flow information. This data structure

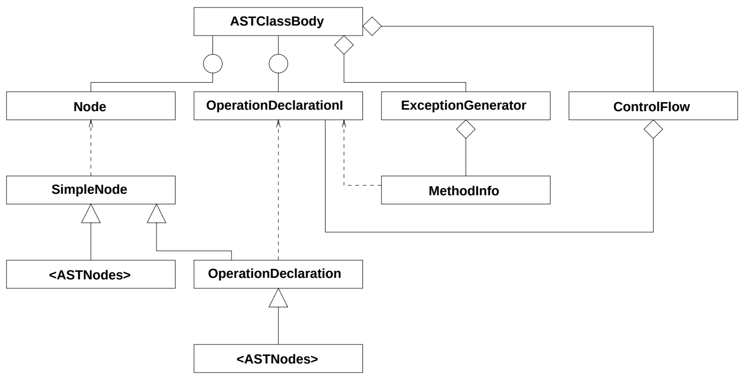

Figure 4: Modified Control-Flow Design

was composed of `MethodInfo` objects initialized with `OperationDeclarationI` objects. `MethodInfo` object contain attributes and methods related to both control-flow and exception information generation activities. This caused a problem of coupling between the exception information generation and control-flow extraction activities.

In the new design, the class `ControlFlow` was created to hold the structures that were previously held in the `ExceptionGeneration` object. Operations to generate control-flow were then moved to the `ControlFlow` class. To decouple the control-flow operations from the exception-related operations in `MethodInfo`, the control-flow data structure was made to hold `OperationDeclarationI` objects instead of `MethodInfo` objects, and the services necessary for control-flow analysis were moved from the `MethodInfo` class to a new class, `OperationDeclaration`, inserted in the AST node hierarchy.

This redesign successfully factored out the control flow from the exception generation aspect. It is now possible to provide an AST with only type analysis operations, or with additional control-flow and exception generation operations.

The installation of the `ControlFlowA` aspect in the design of Jex introduced some interesting issues, like linking classes to aspects and dependent aspects. Linking classes to aspect refers to the fact that in the new design, the `ControlFlow` class is only necessary in conjunction with the `ControlFlowA` aspect and can be removed from distributions not including the control-flow feature. The insertion of the `ControlFlowA` aspects also introduced a dependency between aspects, because, since it relies on control-flow operations, the `JexA` aspect can now only be weaved in together with the `ControlFlowA` aspect.

4 Discussion

While designing with aspects, and especially with the `JexA` and `ControlFlowA` aspects, it was useful to think about aspects as architectural layers with weaving serving as an architectural connector.

This approach was useful because it supported a decomposition of components that was not possible using Java alone. Using aspects as layers also made it possible to avoid potentially "dangerous" uses of aspects, such as aspects with bidirectional coupling [2].

Using a "layered" approach to the aspect design meant that different autonomous entities could be generated depending upon which aspects were layered together. For example, in the Jex redesign, each of the three compilable combinations of `JexA` and `ControlFlowA` (AST, AST + `ControlFlowA`, AST + `JexA` + `ControlFlowA`) form different autonomous components.

Since the initial architecture of Jex heavily relied on components having their interactions defined at compile time, this case study focused on AspectJ's static composition capabilities. AspectJ also supports *dynamic* aspects, which could have been used to add more dynamically-configurable features.

5 Summary

Two useful aspects were identified for the Jex static analysis tool: an aspect containing analysis specific code (e.g., for exception analysis), and an aspect containing optional analysis functionality (e.g., coarse-granularity control-flow). Both of these aspects are decomposition aspects which allow functionality to be factored out of an existing code component, a collection of classes implementing an AST component. Factoring the code into aspects rendered the AST component more reusable.

One aspect factored very cleanly out of the existing code base. In essence, the code comprising this aspect was already separated within the existing code base: Java had simply not provided any language facilities to make the separation explicit. It was more difficult to factor out the second aspects, requiring introduction of new classes, etc. This aspect represented code which had not been recognized as separable when the system was first coded. Standard refactoring techniques [4] could have aided the migration process to the aspect form.

Acknowledgments

This research was funded by the Xerox Corporation.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and Jonh Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[2] Mik A. Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In

Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999. To appear.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, June 1997.

[4] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[5] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java™ programs. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 322–337. Springer-Verlag, September 1999.

[6] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, May 1999.

[7] Xerox PARC. *AspectJ HomePage*. http://aspectj.org.