EXTRACT: Extensible Transformation and Compiler Technology

by

Paul W. Calnan, III

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

Paul W. Calnan, III

April 30, 2003

APPROVED:

____

Dr. George T. Heineman, Major Advisor

____

Dr. Lee Becker, Department Reader

____

Dr. Micha Hofri, Head of Department

Abstract

Code transformation is widely used in programming. Most developers are familiar with using a preprocessor to perform syntactic transformations (symbol substitution and macro expansion). However, it is often necessary to perform more complex transformations using semantic information contained in the source code.

In this thesis, we developed EXTRACT; a general-purpose code transformation language. Using EXTRACT, it is possible to specify, in a modular and extensible manner, a variety of transformations on Java code such as insertion, removal, and restructuring. In support of this, we also developed JPath, a path language for identifying portions of Java source code. Combined, these two technologies make it possible to identify source code that is to be transformed and then specify how that code is to be transformed.

We evaluate our technology using three case studies: a type name qualifier which transforms Java class names into fully-qualified class names; a contract checker which enforces pre- and post-conditions across behavioral subtypes; and a code obfuscator which mangles the names of a class's methods and fields such that they cannot be understood by a human, without breaking the semantic content of the class.

Acknowledgements

I would like to thank my advisor, Dr. George Heineman, for his invaluable support and guidance throughout the course of this project. I would also like to thank Dr. Lee Becker for being the reader of this thesis.

During the course of my five years in Worcester, I have met many wonderful people who have made my time here that much more enjoyable, and for that I am grateful. In particular, I would like to thank Sumeet; you've helped me in more ways than you can imagine and I couldn't have done this without you.

Finally, this thesis is dedicated to my parents, who gave me this opportunity. Thank you.

This work is sponsored in part by the Defense Advanced Research Projects Agency under DARPA Order K503 monitored by the Air Force Research Laboratory F30602-00-2-0611.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Worcester Polytechnic Institute, the Defense Advanced Research Projects Agency, or the U.S. Government.

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

Table of Figures vi

Conventions vii

Chapter 1 Introduction 1

Chapter 2 Related Work 6

AIDE Compiler 6

XSL Transformations 6

Byte-Code Instrumentation 8

Visitor Design Pattern 9

Chapter 3 Design Considerations 12

Component Adapters 12

The AIDE Prototype Compiler 13

Approaches to Code Transformation 15

Requirements for EXTRACT 16

Chapter 4 JPath – A Path Language for Java Source Code 18

Introduction 18

Location Paths 21

Location Predicates 22

Evaluating JPath Expressions 23

Implementation of the JPath Evaluation Engine 25
JPath Visitors 25
OpenJava Visitors 27
Chapter 5 The EXTRACT Language 29
Introduction 29
EXTRACT Modules 29
Execution Blocks 31
Transformation Declarations 33
Module Properties 36
Implementation Classes 38
Extensibility 39
Main Modules 40
Module Compilation 42
Module Execution 46
Chapter 6 Evaluation 47
Chapter 7 Case Study: Type Qualifier 48
Chapter 8 Case Study: Behavioral Contract Checking 55
The Behavioral Subtyping Condition 56
Chapter 9 Case Study: Code Obfuscator 60
Chapter 10 Conclusion 62
References 65

Table of Figures

Figure 1: Formatting Conventions vii

Figure 2: Embedded Code Sensors 2

Figure 3: Visitor Design Pattern Object Structure (from [14]) 10

Figure 4: Visitor Design Pattern Sequence Diagram (from [14]) 11

Figure 5 - Sample AIDE Transformations 14

Figure 6: AST for `HelloWorld.java` 20

Figure 7: JPath Selection Pseudo-code. 26

Figure 8: EXTRACT Module Compilation 30

Figure 9: The Behavioral Subtyping Condition 56

Figure 10: The Behavioral Subtyping Condition, Generalized to Multiple Inheritance .. 56

Conventions

The following formatting conventions are used:

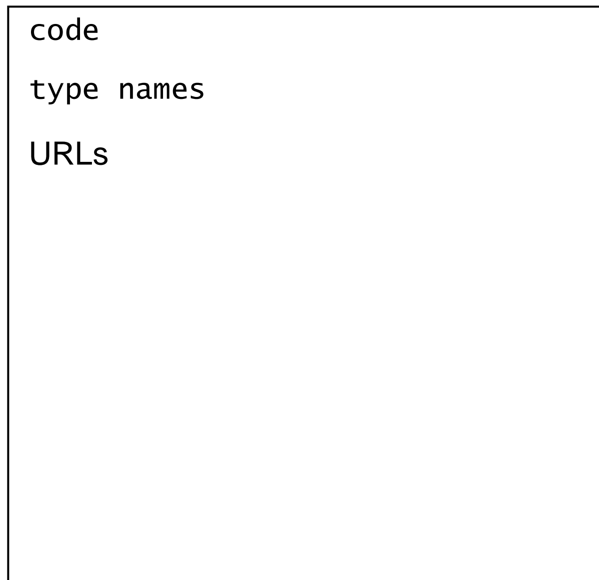

Figure 1: Formatting Conventions

Chapter 1
Introduction

Code transformation is widely used in programming. The most common example is using a preprocessor that performs symbol substitution and macro expansion. This form of code transformation is often based on syntactic transformations; for example, C code is transformed into more C code through string substitution. There are domains, however, where code transformations are complex and the resulting target language differs from the source language. An example of this is the Java Remote Method Invocation (RMI) compiler that generates stub and skeleton classes from Interface Definition Language (IDL) specifications [27].

In our application domain, we need to instrument Java classes by adding callbacks at the beginning and end of each method. The resulting classes have an active interface [17]. These classes form the basis for component adaptation, monitoring and validation, and it allows users to create Embedded Code Sensors (ECS). An ECS is a *probe* that can emit events when an object is instantiated, a class attribute is accessed, a user-specified assertion fails, a reflective method invocation occurs, or an exception is thrown.

The Active Interface Development Environment (AIDE) compiler was developed as a prototype for the DARPA DASADA (Dynamic Assembly for Systems Adaptability, Dependability, and Assurance) project [CITE – Peter Gill]. AIDE takes a Java source file and instruments each class in the file. Each class is made to implement the `Adaptable` interface (as well as the `StaticAdaptable` interface, if the class contains static methods) by adding the necessary methods and appropriate fields. These interfaces allow adapters to be attached to instrumented objects. Then, callback code is added at the

beginning and end of each method. When each method is called, if an adapter is attached to the object, the adapter's `invokeCallback` method is called. The arguments to this method provide the signature of the instrumented method, the values of the instrumented method's parameters, and a reference to the instrumented object containing the method. Similar callback code is added before each return statement as an after hook. The callback code (i.e., `invokeCallback`) in the attached adapter can be used to monitor the execution of the instrumented class's methods. The callback code also allows for the adapter to modify the parameters sent to the method as well as the value being returned.

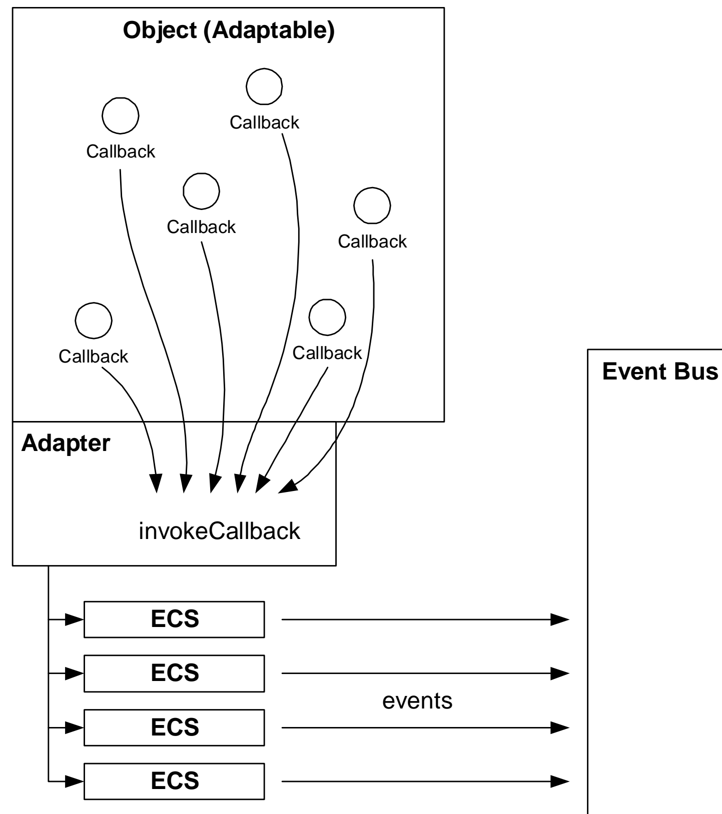

Figure 2: Embedded Code Sensors

For the DASADA project, an infrastructure was developed for instrumenting running systems with probes and passing the data gathered by the probes to gauges [15]. The gauges can then collect, collate, filter, and aggregate that data to provide system level measurements of the system's operation. The goal of DASADA is to improve a system's responsiveness and robustness by dynamically analyzing system level measurements to determine any modifications, adaptations, or reconfigurations [1]. The callbacks inserted by the AIDE compiler are used to generate events. These events are broadcast on an Internet-scale bus (SIENA – Scalable Internet Event Notification Architectures [28]). External gauges can then subscribe to the SIENA event stream and monitor the information sent out by the probes. This information is then used to decide whether and how the monitored application must be reconfigured, and then coordinate lower-level effectors to implement the actual reconfiguration [1].

Applications can subscribe to the event streams for a variety of monitoring purposes. SoftViz [24] provides program-flow visualization by displaying, either in real-time or from a captured event stream, which methods are called in a probed application. Other applications might monitor the event streams for patterns of behavior. Certain patterns signify the failure of a given component, high load on a given component, or a variety of other conditions. This enables monitoring applications to dynamically reconfigure the target application.

There are a variety of ways in which probes can be embedded into an application. Method calls can be intercepted by the runtime environment or by a DLL, executable code or Java byte-code can be instrumented, or semantic information can be used to insert probes directly into the source code. The AIDE compiler was designed to perform

the latter method of embedding probes. In its prototype form, AIDE was a Java grammar (for JavaCC [21]) that emitted the probe code when the appropriate points were reached during the parsing of a source file. This proved to be difficult to modify and maintain. The technology also required multiple passes to process the code.

AIDE represents a specific type of code transformation; namely, the insertion of probe code at the beginning and end of certain methods. However, other types of code transformations can be useful in the development and testing of code. For example, code transformations can be used to perform refactoring, coverage checking, and code obfuscation.

This thesis presents the EXTRACT (*Ex*tensible *Tr*ansformation *a*nd *C*ompiler *T*echnology) language and its supporting libraries. Using EXTRACT it is possible to specify, in a modular and extensible manner, a variety of transformations on Java code. While the version of EXTRACT developed in this project is limited to Java code, it is possible to extend it to be used with other languages. This thesis also presents JPath, a path language for Java source code. JPath provides a mechanism for EXTRACT modules to identify code to be transformed. We also present the supporting JPath interpreter libraries and EXTRACT compiler.

The remainder of this paper is organized as follows: In Chapter 2, we examine related work. In Chapter 3, we discuss the design considerations for EXTRACT. In Chapter 4, we introduce JPath and show how it can be used to locate portions of Java code for the purpose of transformation. In Chapter 5, we introduce the EXTRACT language and describe how it is used to specify code transformations. In Chapter 6 and

beyond, we evaluate the EXTRACT technology and present case studies on how it performs.

Chapter 2
Related Work

AIDE Compiler

As discussed in Chapter 1, the Active Interface Development Environment (AIDE) was developed as a prototype Java preprocessor for the DARPA DASADA project. It served a limited purpose in that it only instrumented classes so that they would implement a given interface and added code at the beginning and end of certain methods. The code transformations that it performed were limited in scope, and proved to be difficult to modify and maintain. This is due to the fact that the transformations were embedded in a JavaCC grammar.

The transformations that AIDE performs can be viewed as specific instances of a more general set of transformations. We envisioned a technology that was capable of not only inserting code, but also inserting, deleting, and restructuring code. To this end, we developed EXTRACT.

XSL Transformations

The Extensible Markup Language (XML) allows users to define their own markup tags for storing structured data. Using either a Document Type Definition (DTD) or an XML Schema, users can define how markup in a document conforming to such a specification must appear. However, different users will inevitably define different tags and structures for similar data. For instance, the following two XML documents represent the same information but are defined using different schemas:

```
<?xml version="1.0"?>
```

```
<product title="Component Based Software Engineering: Putting the Pieces
Together" id="0201794854">
<author name="George T. Heineman">
<author name="William T. Councill">
</product>
```

```
<?xml version="1.0"?>
<book>
<title>Component Based Software Engineering: Putting the Pieces Together
</title>
<author>George T. Heineman </author>
<author>William T. Councill </author>
<isbn>0201794854</isbn>
</book>
```

To a human reader, it is easy to see the mapping between the two document types. However, a mechanism is needed to programmatically transform documents from one type to another. XSL Transformations (XSLT) perform this task.

The XSLT specification [7] defines an XML-based language for expressing transformation rules from one class of XML document to another. An XSLT stylesheet can describe transformations from XML to arbitrary text-based formats. The source document is parsed and a parse tree conforming to the Document Object Model (DOM) [10] is produced. The XSLT stylesheet then uses the XML Path Language (XPath) [9] to identify portions of the DOM tree which correspond to portions of the original document. The stylesheet then contains rules for emitting formatted text based on the portion of the DOM tree. Schema transformations are described in XSLT by implementing an exemplar of the target schema in terms of its deltas from the source [5]. Thus, interoperability between disparate document types can be achieved.

EXTRACT uses a similar approach to transforming Java code. A Java source file is parsed into an Abstract Syntax Tree (AST). An EXTRACT module then uses JPath (a Java path language similar to XPath; see Chapter 4 for details) to identify portions of the Java code. The module then specifies how to transform that portion of code.

Byte-Code Instrumentation

As an alternative to transforming Java source code, it is possible to transform the compiled byte-code. This section will examine a number of approaches to byte-code instrumentation.

ProbeMeister [26], also developed under the DASADA project, supports the dynamic insertion of probe code into Java byte-code. Probe code is self-contained and can be inserted into code dynamically at any point during the application's execution. Probes can be inserted while the application is running, and can be inserted into byte-code that is executing remotely. The probes then emit data back to ProbeMeister for viewing. This sort of byte-code manipulation requires sophisticated support from the virtual machine, as found in the Java Development Kit (JDK) 1.4.

Javassist [6][20] is a load-time reflective system for Java. It provides a class library for editing byte-code from within a Java application, enabling applications to define a new class at runtime and to modify a class file when it is loaded by the JVM (if used with a customized class loader).

Addistant [31] transforms byte-code at load time. It enables the distributed execution of Java software that was originally developed to run on a single virtual machine. No source code modifications are necessary. It can modify Swing applications so the GUI executes on the local machine while the rest of the application executes remotely.

Two other technologies, Java-MaC [23] and Java PathExplorer [16] perform byte-code instrumentation for runtime verification of Java applications. Both technologies insert probes into Java byte-code which emit events that are interpreted by

an outside application that performs the verification. A high level, state-based specification of the behavior of the monitored application is given to the verifying application. The verifying application then infers state transitions from the events that are emitted and compares them to the corresponding specification. When the application behaves differently than the specification, a warning or errors are reported back to the user.

Visitor Design Pattern

Portions of this paper make reference to the Visitor design pattern [14]. This section reviews the Visitor pattern and how it is used. Readers familiar with this design pattern can skip this section.

As described in [14], the Visitor design pattern represents an operation to be performed on the elements of an object structure. It allows the developer to define a new operation without changing the classes of the elements on which it operates. Given a complex structure of objects (elements) whose interfaces vary, it is possible to define operations on that structure that depend on the concrete class being operated on. Rather than define these operations in the element classes, which would scatter the implementation of an operation across a number of different classes, the Visitor design pattern allows the developer to define an operation in a single Visitor object which visits each element and performs the operation. To add a new operation, one simply defines a new Visitor class. The visitor implements each operation declared by the visitor, and each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure.

The figures below show UML diagrams for the class structure of the Visitor design pattern and the sequence of calls made between the elements and the visitors.

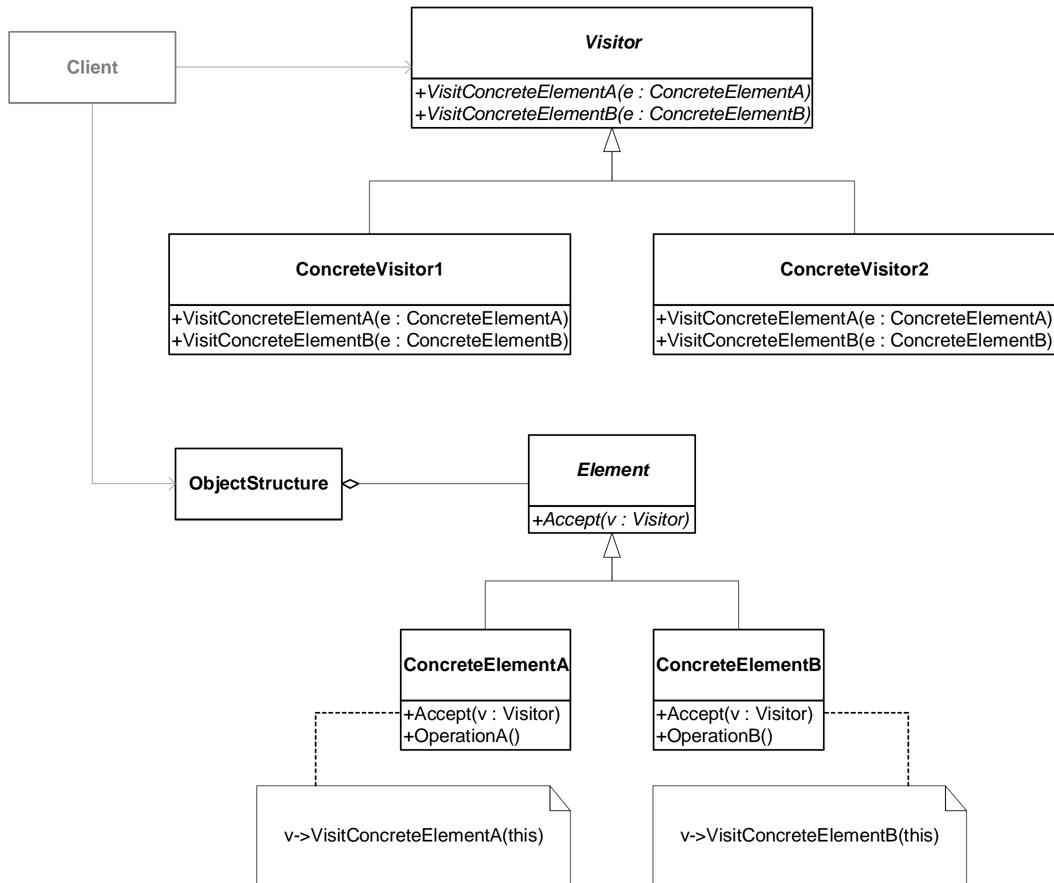

Figure 3: Visitor Design Pattern Object Structure (from [14])

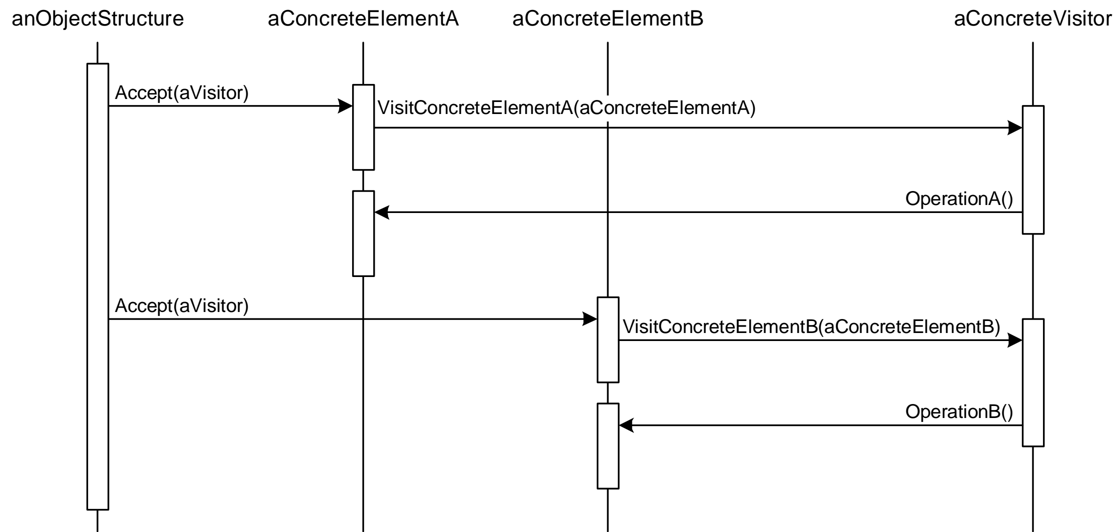

Figure 4: Visitor Design Pattern Sequence Diagram (from [14])

Chapter 3
Design Considerations

Component Adapters

In [17], a mechanism for adapting software components is described. When assembling a system from reusable software components, the developer needs to locate a component that matches the functionality and interfaces that are needed. Often, a component does not exactly match a particular need. Therefore it is necessary for developers to be able to adapt the behavior of a component.

Consequently, software components can have two interfaces; one for the behavior of the component, and one to adapt that behavior as needed. A component's interface defines more than a syntactic description of the method invocations accepted by a component; it can define methods to invoke, events to send and receive, or complex access protocols [17]. An active interface, introduced in [17], decides whether to take action when a method is called, an event is announced, or a protocol executes. Interface requests occur in two phases: the "before-phase" which occurs prior to any execution of the request, and the "after-phase" which occurs once the component has completed the execution of the request. Active interfaces allow for the specification of callback functions to be invoked during these phases, allowing a developer to augment, replace, or deny a client request [17].

The ADAPT Project [1] defined a set of Java interfaces (namely `edu.wpi.cs.adapt.Adaptable` and `edu.wpi.cs.adapt.StaticAdaptable`) that provide a component or class with an active interface. These interfaces allow for callback methods to be installed that augment, replace, or deny a client request on a

component or object. The callbacks are invoked in the "before-phase" when the method is first called and in the "after-phase" when the method is about to return.

In order for an object to be adaptable in the manner, code needs to be present at the beginning and end of each method that calls the callback method and takes the appropriate action as signaled by the callback method. While it is possible to have the developer insert this code when writing the class, it is preferable that this process be automated. Code that achieves this functionality fits a template, and can be inserted programmatically. The Active Interface Development Environment (AIDE) instruments a class and its methods in this manner. The class is made to implement the `Adaptable` and `StaticAdaptable` interfaces and the necessary callback code is inserted into each method.

The AIDE Prototype Compiler

The original AIDE compiler was developed using a JavaCC [21] grammar that parses a Java source file and reproduces the code as a string. Predefined lines of Java code (i.e., the callback invocation code and the interface implementation code) are inserted when the compiler reaches specific points in the grammar (e.g., when a method is entered). Code is inserted at the beginning of each method and before each return statement (or at the end of a void method). Class variables are added to the end of the class definition to store the component adapter. The class is declared to implement the `Adaptable` interface, and accessor methods for the component adapter are added at the end of the class definition.

The technology also requires multiple passes to process the code. Specifically, after callbacks often result in unreachable statements. The figure below shows how various methods would be instrumented using AIDE. Note how instrumenting method m2 results in dead code. Thus, the second stage of the AIDE compiler uses `javac` to compile the source code produced by the first stage, examines the output for dead code or unreachable statement errors, and then removes the offending code.

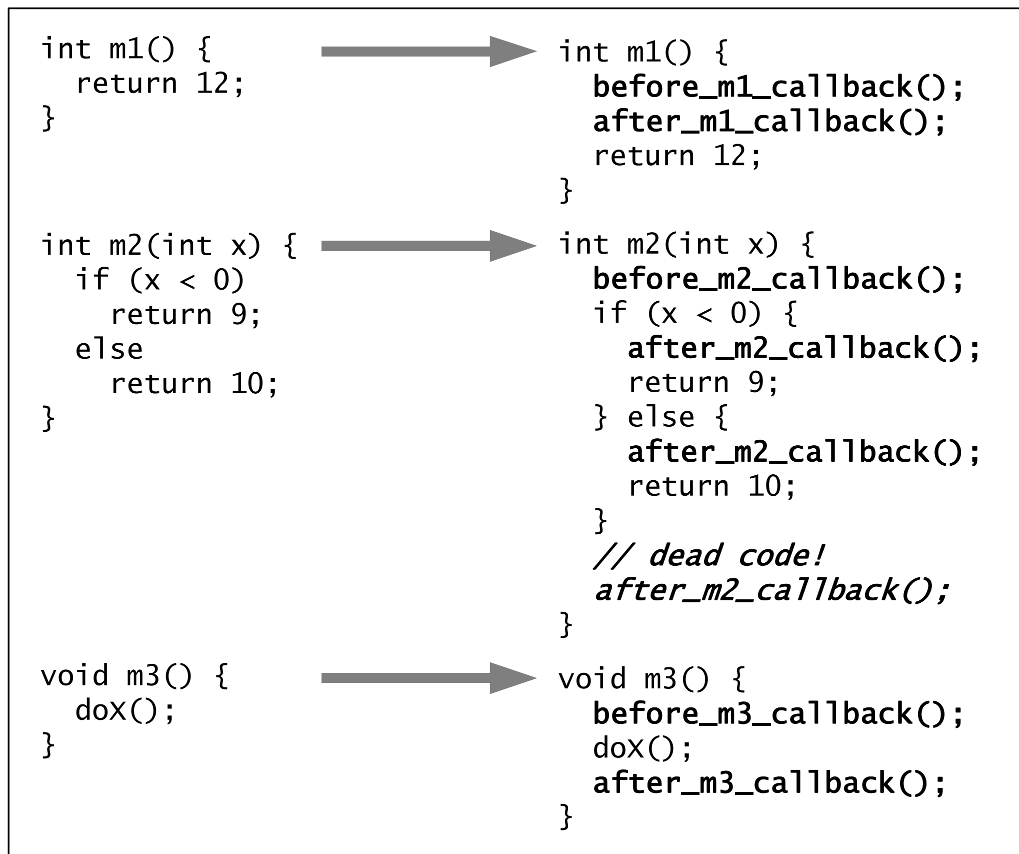

Figure 5 - Sample AIDE Transformations

There are a number of problems with this approach. First, all code transformations (here, only insertions) are explicitly written in the parser code. This makes maintenance of, and modifications to, the inserted code difficult as it is scattered throughout the grammar. Second, general-purpose code restructuring is difficult if not

impossible considering just the syntax. Code insertions can be performed, but modifications to existing code would require significant effort. Third, since all transformations are embedded in the parser, it is impossible to detect whether multiple transformations are incompatible. Fourth, the second compilation stage is required because only the syntax was considered during transformation. It must be possible to transform correct code in a single transformation phase.

Approaches to Code Transformation

Various approaches can be taken to programmatically instrument Java classes. If the source code is available, the code itself can be modified to accomplish the instrumentation. If the source code is unavailable, the compiled byte-code must be manipulated to instrument the code.

Byte-code instrumentation can be performed either prior to run-time by an external application or at run-time by the class loader. The latter requires sophisticated virtual machine support, as found in J2SE 1.4 [18]. The byte offset of the beginning of a method can be retrieved from the class file. From there, the appropriate byte-code implementing the probe can be inserted. While tools exist for programmatically manipulating byte-code (see the previous chapter), some instrumentation applications may require that the byte-code be disassembled and analyzed to determine the location of the probe. For instance, to insert a probe at the end of a method, program flow analysis is needed to determine all exit points of the method [26].

Source-code instrumentation resolves some problems found in byte-code instrumentation. Java source code can be parsed into an abstract syntax tree (AST) and

then manipulated using graph rewriting techniques [12]. If the AST data model is mutable, transformation can be performed on the tree itself and exported into a file. Searching for the appropriate location to insert probes is simplified by the additional syntactic information contained in the AST.

Requirements for EXTRACT

The limitations of the prototype AIDE compiler, described above, lead us to a number of requirements for EXTRACT, a general-purpose code transformation system. First, an AST grammar and library are required for the target language. This project focused only on the Java language, but we expect that EXTRACT could be extended easily for use with a variety of other languages. To implement arbitrary transformation on an AST, an appropriate AST class library was required. After evaluating a number of AST libraries [4][19][20][22][25][30], we decided to implement EXTRACT using the OpenJava [25] metaobject protocol (MOP). OpenJava was chosen because it creates a mutable AST on which transformations can be performed. Furthermore, we can simplify the process of inserting code by using their library to parse code fragments into their appropriate AST subtrees. Rather than having to explicitly build AST subtrees for inserted portions of code, OpenJava provides a way to convert arbitrary strings into the correct AST subtrees that are then ready to insert into the main tree.

The OpenJava MOP was originally used as an extension system for the Java language. It allows developers to extend Java, write code using the extended language, and then define ways to transform that code back into standard Java. Consequently, it provides a parser which creates a mutable AST. The AST allows a variety of

transformation, most importantly insertion and deletion. Using these facilities, it is possible to implement our entire range of transformations. The AST provided by OpenJava has built-in support for the Visitor design pattern [14], which simplifies traversal code.

Chapter 4
JPath – A Path Language for Java Source Code

Introduction

As discussed in previous sections, the OpenJava library provides a mechanism for parsing Java source code into an abstract syntax tree (AST). EXTRACT processes all transformations over the AST. For EXTRACT scripts to be compact and provide a basis for analysis, a mechanism is needed to identify portions of the AST. To accomplish this we designed JPath, a path language for navigating through and identifying portions of Java source code.

JPath was inspired by XPath, a language designed to address parts of XML documents [9]. XML documents are parsed into an AST conforming to the Document Object Model (DOM) [10]. XPath provides a mechanism for addressing nodes in the DOM tree and provides functionality for a variety of XML technologies, including XSL Transformations [7].

Paths in JPath are stated as hierarchical path expressions, for example:

```
/a/b/*/c
```

Paths are given as a sequence of steps from either the root (i.e., an absolute path) or from some location in the tree (i.e., a relative path). Paths can also contain wildcards which may appear at any point in the JPath expression except at the rightmost end. This ensures that JPath expressions evaluate to a homogeneously typed set of nodes.

A JPath expression is evaluated relative to a context node to yield a node set. This node set, which can contain zero or more nodes, represents the portions of the AST that match the JPath expression. Each matched node in the node set is associated with a

concrete path from the root of the AST to that node. This path can be retrieved in an EXTRACT script and used to derive further context for the selected node. Additionally, individual steps in a JPath expression can be enclosed in parentheses. When a match is found, the nodes along the concrete path that correspond to the steps in parentheses are accessible as sub-matches. This is similar to Perl's use of parentheses in regular expressions.

To illustrate how JPath works, we present a simple example. Consider the following trivial Java class:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world.");
  }
}
```

A source file (`HelloWorld.java`), containing the class declaration above, is then parsed into an OpenJava AST. The resulting AST, used for our JPath examples, is shown in Figure 6: AST for `HelloWorld.java`.

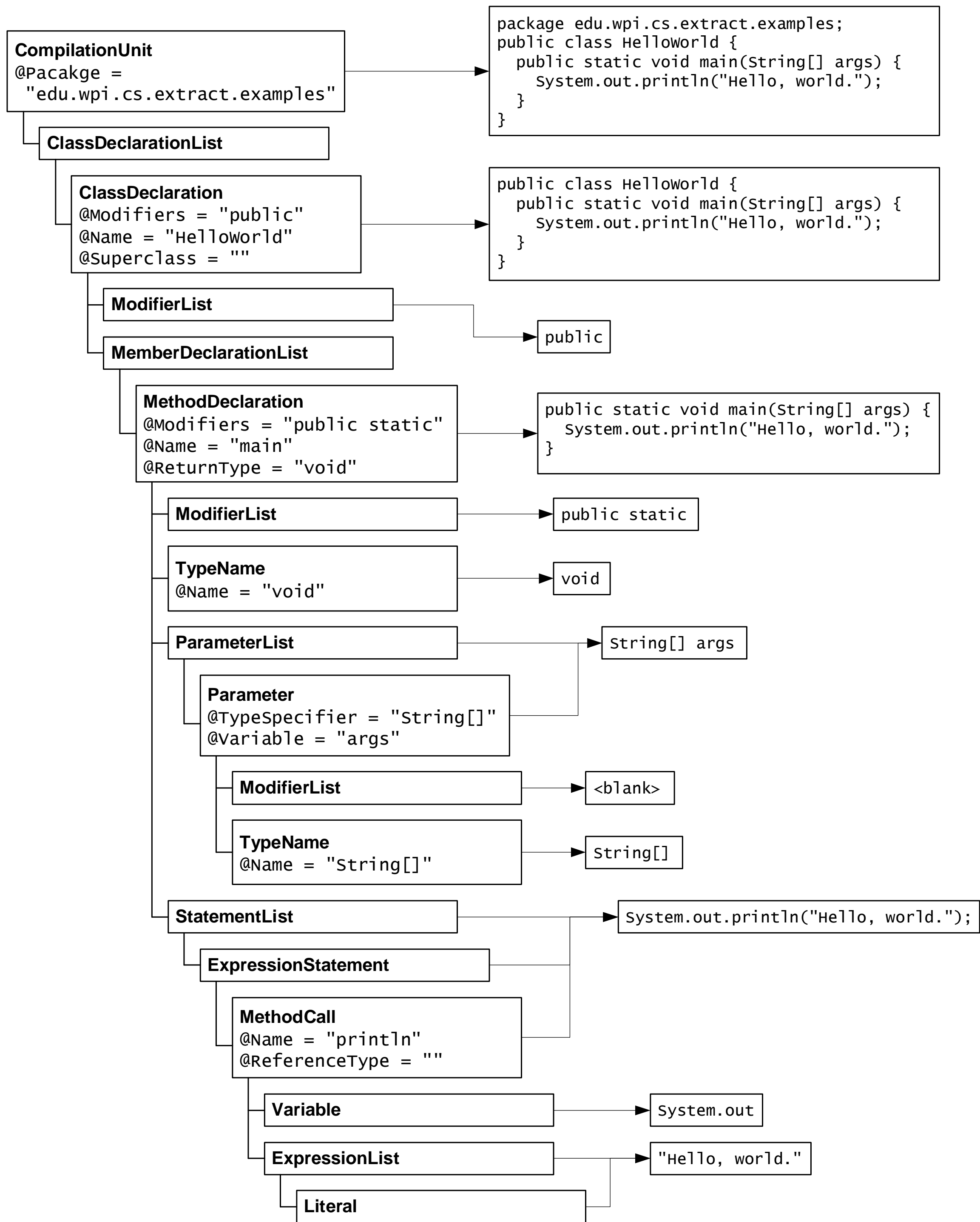

Figure 6: AST for `HelloWorld.java`

Location Paths

A JPath location path is expressed as a sequence of one or more location steps, separated by a '/' character. A '/' character alone denotes the root node of the Java source code (i.e., the `CompilationUnit`). Thus, any JPath beginning with a '/' character is an absolute path; all other path expressions are considered to be relative. Each location step corresponds to a class in the AST hierarchy (e.g., `CompilationUnit` or `MethodDeclaration`). JPath also allows wildcards ('*' and '.' characters) to be used in place of any location step (except the last). The '.' wildcard matches all children of the context node. The '*' wildcard matches the context node and all of its descendants (i.e., zero or more steps from the context node).

Each location step selects a set of nodes relative to a context node. An initial sequence of steps is composed with a succeeding step in the following manner. The initial sequence of steps selects a set of nodes, in document order, relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes selected from this step are then unioned together and used as context nodes for the next step.

This is best illustrated by an example. Consider the following JPath expression (to be evaluated on `HelloWorld.java`):

```
/*/ClassDeclaration/MemberDeclarationList/MethodDeclaration
```

The first character in the JPath expression is a '/', so the context node is the root of the AST (i.e., the `CompilationUnit`). The first step is a '*'; we select the context node and its entire sub-tree. Each node in the selection is used in turn as the context node for the following step. Because this is an ordinary location step, we examine the children of

each of the context nodes and select those whose type matches the step (ClassDeclaration). This will select the only ClassDeclaration in the tree. That ClassDeclaration is then used as the context node for the next step, selecting its only child of type MemberDeclarationList. Finally, the selected MemberDeclarationList is used as the context node for the last step which selects the MethodDeclaration child. Note that the JPath expression: /*/MethodDeclaration would select the same set of nodes.

Location Predicates

The location steps described above are limited in that only the node type can be used in selection. JPath provides the option of using location predicates that allow for further refining of node selections for a given step. Any location step can have one or more location predicates, contained in square brackets. Location predicates can refer to a set of attributes specific to a given node type. For example, the JPath expression:

```
MethodDeclaration
```

selects all method declarations that are children of the context node. We can select all public method declarations using the following expression with a location predicate:

```
MethodDeclaration[@Modifiers contains “public”].
```

Note that attributes are identified by the ‘@’ character preceding their name. Attributes evaluate to either scalar values (i.e., integers or strings) or list values (i.e., arrays or lists of scalar values). Consequently, equality (==, <, <=, >, >=) and list containment (“contains”) operators are provided. Regular expression matching is provided using the “matches” operator. Location predicates can be combined using “AND” and “OR”.

Thus, we can further refine our selection to public and protected methods with a boolean or:

```
MethodDeclaration[@Modifiers contains "public" AND
                                @Name == "main"]
```

Figure 6: AST for `HelloWorld.java` gives the attribute values for the node types shown. Consult the documentation for a complete list of attributes and their descriptions.

When a location predicate follows a list type (e.g., `StatementList`), the predicate can be used to select an element from the list. For example, `StatementList[3]` selects the 3rd element in the statement list.

Evaluating JPath Expressions

Each step in a JPath expression is checked to ensure that it corresponds to a valid OpenJava class name, but no effort is made to ensure a logical ordering. In other words, JPath expressions must be syntactically correct but not semantically correct. Thus, a JPath expression such as:

```
IfStatement/CompilationUnit
```

is considered syntactically correct, even though it is impossible for an if-statement to have a compilation unit as its child. In this case, the expression would evaluate to an empty node set.

Steps in the path expression are evaluated in a left-to-right order. Given a context node, we consider only the sub-tree of the AST rooted at that node. For a location step containing a type name, each child of the context node is examined; if its type matches the location step's type name, this node is considered a match. For a location step containing a wildcard, the appropriate descendant nodes are selected. The resulting node

set is then filtered by the location predicate to yield the final set of matches for that step. Each node in the set of matches is used as the context node for the next step in the expression. These resulting node sets are unioned together and used as the context nodes for the next step, and so on until the end of the expression is reached. The resulting node set corresponds to all nodes matching the entire JPath expression. (Note: Pseudo-code for the selection algorithm is shown in the next section.)

To illustrate this, consider the following JPath selection statement:

```
select(contextNode,
        "typeA/*/typeB[predB]/typeC[predC]/*/typeD[predD]")
```

This is equivalent to:

```
select(select(select(select(contextNode,
                            "typeA"),
                      "*/typeB[predB]"),
               "typeC[predC]"),
        "*/typeD[predD]")
```

Note that the `select()` function, as presented here, returns a node set. Thus, the context node (i.e., the first argument) to the function can be either a node or a node set. If a node set is provided as the context node, the JPath expression is evaluated relative to each node in the set. The union of the nodes selected from each evaluation of the JPath expression across the node set is returned.

The resulting node set can be iterated through or accessed using an index. Each node in the resulting node set has a corresponding concrete path that from the root of the AST to the selected node. The concrete path can be examined to extract context. Furthermore, any nodes matching sub-expressions (contained in parentheses in the JPath expression) can be accessed as sub-matches. This also allows for the retrieval of

contextual data. The type of the nodes in the resulting node set will be the same as the type name specified in the last (rightmost) location step in the JPath expression.

Implementation of the JPath Evaluation Engine

JPath Visitors

The `edu.wpi.cs.jpath` package provides a parser and parse-tree classes for the JPath language. Given the JPath grammar, a JavaCC grammar was produced. Then, Java TreeBuilder [18] was used to produce AST classes for the JPath language (one for each grammatical production) and a JavaCC parser that builds a JPath AST from a string.

As with the OpenJava AST library, the JPath parse tree library supports the visitor pattern. Consequently, we can use visitors to interpret JPath expressions. A JPath visitor object (`LocationPathEval` in Figure 7: JPath Selection Pseudo-code.) traverses the expression tree in a depth-first manner, sending messages to an object implementing `PathEvaluator` when the various selection elements are encountered. The `PathEvaluator` modifies the selection when these messages are received using OpenJava visitor objects (described in the next section).

Figure 7: JPath Selection Pseudo-code.

If the `LocationPathEval` visitor encounters an `AbsoluteLocationPath`, it notifies the `PathEvaluator`. The `PathEvaluator` then discards the current context node set and selects the root of the AST. When the `LocationPathEval` visitor

encounters a LocationStep, it notifies the PathEvaluator that a step has been encountered and whether it is a back-reference (beginStep(boolean)). The LocationPathEval then visits the NodeType, and notifies the PathEvaluator to select children of the context nodes, depending on the node type or wildcard specified (selectAllChildren(), selectSubtree(), selectChildren(String)). If a Predicate is present, it is evaluated using a separate predicate visitor. This visitor translates attribute names into methods to be invoked on a source-tree node to retrieve the attribute's value. This filters the source-tree nodes selected by the LocationStep.

OpenJava Visitors

As discussed in previous sections, the OpenJava library contains support for the visitor design pattern. The OJPathEvaluator controls a set of OpenJava visitors. Since expressions are evaluated relative to a context node, we can search from a context node using that node's childrenAccept() method (calls accept() from the visitor pattern on each child). By default, the visitor does not recurse through the entire tree. Rather, only the children of the context node are considered. However, if the visitor is searching for the '*' wildcard, it will recurse through the entire sub-tree.

In the visitor design pattern, the Visitor class has visit() methods for each type in the object structure. Thus, an OpenJava visitor would define visit() methods for each AST class. For the purposes of searching, however, it suffices to have a generic visitAll() method that takes as an argument the base class for the AST (i.e., ParseTree) that is called by the individual visit() methods. A candidate for a match is found when its class name (as retrieved by the getClass() method in java.lang.Object) matches the name provided in the location step.

The set of candidate matches are then filtered using the location predicate, if one exists. A mapping exists between attribute names accessible in the location filter and methods used to retrieve those attributes from the OpenJava AST objects. If the location predicate evaluates to a true value, the candidate match is considered a match. Otherwise, it is discarded.

Pseudo code for the selection algorithm is presented in Figure 7: JPath Selection Pseudo-code.

Chapter 5
The EXTRACT Language

Introduction

The EXTRACT (Extensible Transformation and Compiler Technology) language allows developers to define transformations in a modular and extensible manner. This chapter introduces the EXTRACT language and describes how it is used. Throughout this chapter, we will examine a simple example to illustrate how EXTRACT can be used. This example, called `ExceptionAdder`, adds an exception to the throws clause of methods in a Java source file.

EXTRACT Modules

Transformations are expressed as EXTRACT modules. Modules are named and are defined in a file corresponding to the module's name. A module is comprised of an execution block, a set of transformations, and a set of properties. The execution block traverses the AST, identifies contextual information contained in the AST, and calls transforms. Transformations define modifications to the AST. Properties represent data that is collected or generated by the module and exposed for external use. Modules can also have Java implementation files that provide Java code to support the transformations. Each of these parts is discussed in turn later on in this section.

An EXTRACT module is compiled into Java code by the `ExtractC` compiler. The generated Java source, along with the Java implementation file, is then compiled

using `javac`. The module can then be executed to transform a set of Java files using `Extract`. This process is outlined in Figure 8: EXTRACT Module Compilation.

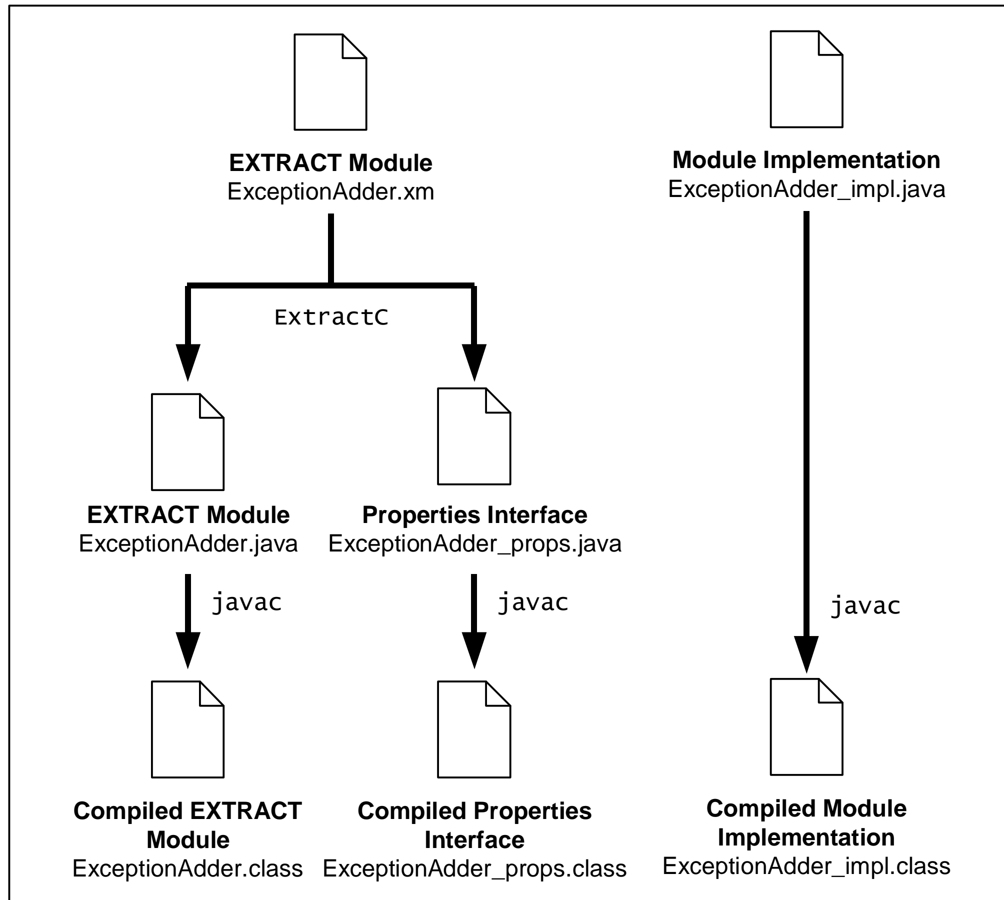

Figure 8: EXTRACT Module Compilation

Now, we will examine the different parts of an EXTRACT module. As concepts are introduced, we will build up the `ExceptionAdder` example until we have a fully-functional version. We begin by defining a module, named `ExceptionAdder`, in a file named `ExceptionAdder.xm`:

```
module edu.wpi.cs.extract.examples.ExceptionAdder {
    // the module body (execution block, transforms, and properties)
    // will be defined here
}
```

In the code fragment above, we define the name of the module (edu.wpi.cs.extract.examples.ExceptionAdder). Note the naming convention; we use a fully-qualified class name for the module. The file (ExceptionAdder.xm) is placed in the same subdirectory (e.g., edu/wpi/cs/extract/examples/) as a Java source file would be placed.

EXTRACT provides package and class importing facilities similar to Java. For example, if a module uses java.util.Vector and java.util.Enumeration, we can add an import statement at the beginning of the module file (prior to the module declaration). Thus, we can add the following import statements:

```
import java.util.Vector;
import java.util.Enumeration;
```

Or, we can import the entire package:

```
import java.util.*;
```

Execution Blocks

Execution blocks traverse a Java source file, identify contextual information necessary to perform transformations, and apply transformations to modify the Java source file. Execution blocks also define parameters for the module. These parameters must be passed to the module prior to execution. In this example, we need to pass the module the name of the exception type that is to be added. Each module can define at most one execution block. If a module extends another module (as will be seen in later examples), the execution block may be inherited from the base module and therefore omitted.

As described above, the `ExceptionAdder` adds an exception to the throws clause of methods in a Java file. In order to accomplish this, we need to locate the method declarations in the Java source code. In this case, we do not need to identify any additional contextual information. We simply need to select each of the method declarations in the source code and apply a transform to each of them. This is shown in the code fragment (which appears in the module body) below:

```
ExceptionAdder (String exceptionType) {
    :: MethodDeclaration method = Select("/*/MethodDeclaration") {
        ApplyTransform(AddException(exceptionType), method);
    }
    { }
}
```

The execution block must have the same as the module (without the package name prefix). Required arguments for the module are specified in the parenthesis following the module name (here, `String exceptionType`).

An execution block specifies how portions of the AST are selected and when transformation are to be applied. In the example above, we use a Select statement to select all of the method declarations in the AST. Select statements in EXTRACT use the following syntax:

```
:: SelectionType elementName = Select([contextNode,] JPath_expression)
{
    // code here is executed for each elementName in the selection
}
{
    // code here is executed if the Select statement returns an empty result
}
```

Thus, the set of all `MethodDeclarations` are selected (using the JPath expression "`/*/MethodDeclaration`"). Since no context node is specified in the select statement, the root of the AST (the `CompilationUnit`) is used. Then, for each element in the selection (in our example, `method`), the following statement is executed:

```
ApplyTransform(AddException(exceptionType), method);
```

The `ApplyTransform` function applies the transform specified in the first argument on the remaining arguments. Here, we apply the `AddException` transform (defined below) on each selected `method`.

In our example, the `ApplyTransform` function is called for each method selected by the selection statement. If the selection statement were to return an empty result (i.e., if there are no `MethodDeclarations` specified in the source file) the code in the second set of braces would be executed.

The execution block defined in the `ExceptionAdder` example is quite simple. We perform a selection and apply a transform to each selected element. Execution blocks can be more complex, however, as will be seen in later examples. Multiple selection statements can be present in an execution block; they can be executed sequentially or in a nested manner (i.e., perform one selection and then for each selected element, perform another selection).

Execution blocks do not necessarily need to call transformations, either. In later examples, we present modules that simply collect information about a Java source file. In this case, the execution block performs a series of selections and saves the appropriate data in the module's properties (discussed below).

Transformation Declarations

Transformation declarations define how a transformation is performed and on what data the transformation can be performed on. Transformation declarations also define a set of parameters for the transform. These parameters are passed to the transformation when it is applied. In this example, we need to pass the transformation the name of the exception type that is to be added.

As we saw above, the ExceptionAdder execution block calls the ApplyTransform function as follows:

ApplyTransform(AddException(exceptionType), method);

where exceptionType is a String and method is a MethodDeclaration. Consequently, we would define the AddException transformation (placed in the module body) as follows:

```
transform AddException (String exceptionType) {
    (MethodDeclaration md) {
        AddToList(GetAttrValue(md, "Throws"), exceptionType, -1);
    }
}
```

Thus, when we call the ApplyTransform function as shown above, the following actions occur:

- The AddException transform is instantiated, using the exceptionType String as a parameter.
- The transform is then applied to the MethodDeclaration method.

The AddException transform defines one transform block (i.e., for a MethodDeclaration named md). It is possible to define multiple transform blocks within a single transform. For example, if we wanted to allow AddException to work on ConstructorDeclarations as well as MethodDeclarations, we would do the following:

```
transform AddException (String exceptionType) {
    (MethodDeclaration md) {
       // ...
    }
    (ConstructorDeclaration cd) {
       // ...
    }
}
```

Note that it is also possible to define transform blocks that take more than one parameter. Thus, it is possible to state our AddException transform as follows:

```
transform AddException2 ( ) {
    (MethodDeclaration md, String exceptionType) {
       // this is the same as in AddException
    }
}
```

To apply this transform, we would use the following code:

```
ApplyTransform(AddException2( ), method, exceptionType);
```

While these two approaches achieve the same result, we will use the first approach in our example.

The `AddException` transform uses two API calls: `GetAttrValue` and `AddToList`. `GetAttrValue` retrieves an attribute from a given AST node. Here, we call `GetAttrValue(md, “Throws”)` (where `md` is a `MethodDeclaration`). This retrieves the throws clause for the method declaration. The attribute value that is returned is a list. Consequently, the `AddToList` function adds the specified value (here, `exceptionType`) to the list at the specified position (here, the index of -1 represents the end of the list).

At this point, we have a fully functional `ExceptionAdder` module. The complete code for this module is shown below:

```
module edu.wpi.cs.extract.examples.ExceptionAdder {
   ExceptionAdder (String exceptionType) {
      :: MethodDeclaration method = Select(“/*/MethodDeclaration”) {
         ApplyTransform(AddException(exceptionType), method);
      }
      { }
   }

   transform AddException (String exceptionType) {
      (MethodDeclaration md) {
         AddToList(GetAttrValue(md, “Throws”), exceptionType, -1);
      }
   }
}
```

Module Properties

Modules can collect data during the course of execution and expose that data to external use. This is accomplished using properties, which are defined within the module body. The syntax for defining properties is as follows:

```
module Name {
   properties {
      [ import module [, module ]* ; ]*
      [ Type Name; ]*
   }

   // ...
}
```

Thus, we can define import the set of properties from another module (using the import statement) and we can define our own set of properties. Modules support a `getProperty` and a `setProperty` operation for retrieving and storing property values.

For example, consider a module which collects the names of classes within a Java source file. We would define a property called `classes` that would be stored as a `Vector`. Then, in the execution block, we would simply record the name of each class that is selected. The code for this example is shown below:

```
module edu.wpi.cs.examples.ClassNameCollector {
   properties {
      Vector classes;
   }

   ClassNameCollector ( ) {
      // get the property
      Vector classes = (Vector) getProperty("classes");

      // make sure it is initialized
      if (classes == null) {
         classes = new Vector();
      }

      // select all class declarations
      :: ClassDeclaration cd = Select("/*/ClassDeclaration") {
         // get the class name
         String name = GetAttrValue(cd, "Name").scalarValue().toString();
         // and add it to the classes vector
         classes.addElement(name);
      } { }

      // save the property
      setProperty("classes", classes);
   }
}
```

Here, we define a `Vector` property called `classes`. The execution block first ensures that the property is initialized. Then, it saves a reference to the property, modifies the property within the selection, and finally saves the property. This is done via the property methods defined in the Module base class (which all Modules extend):

- `getProperty(String propertyName) : Object`
- `setProperty(String propertyName, Object propertyValue) : void`

EXTRACT also provides a set of convenience methods for accessing the properties. These methods are named "`get_propertyName`" and "`set_propertyName`". Thus, the `ClassNameCollector` module would have the following methods generated for it:

- `get_classes() : Vector`
- `set_classes(Vector value) : void`

These methods simply call the `getProperty` and `setProperty` methods; however, they cast the return value and parameter respectively to the correct type.

We could then define another module that has the same properties as the `ClassNameCollector` using an import statement, as shown below:

```
module AnotherModule {
   properties {
      import edu.wpi.cs.extract.examples.ClassNameCollector;
   }
}
```

The resulting module would have the same properties as `ClassNameCollector`, as well as the same accessor methods for those properties.

Implementation Classes

Modules can define Java implementation classes that provide Java code to support the module. This is done to simplify EXTRACT modules, allowing developers to call arbitrary Java code from a module. Every module has a default implementation that is generated by `ExtractC`. Developers can then modify the default implementation, adding whatever functionality is deemed necessary. The implementation class, whose name is the module's name followed by "`_impl`", must define an `init` method and a `dispose` method. These methods are called automatically by the module when it loads the implementation and when the module is finalized, respectively.

The module can access any public fields and call any public methods provided by the implementation class. The implementation class is instantiated within the module's constructor (which is generated by `ExtractC`), and retain a reference to the implementation object named "`impl`". Thus, an method named `foo` contained in the implementation can be accessed via `impl.foo()` from anywhere in the module's execution block or transforms.

The `ExceptionAdder` example does not require any extra functionality in the module's implementation. Consequently, it is sufficient to use the default implementation (`ExceptionAdder_impl`) that is provided by the `ExtractC` compiler. Later examples will show how to provide extra functionality in the implementation.

Extensibility

EXTRACT modules are extensible via inheritance. Thus, we can define a module that extends another module. In the specialized module, we can choose to override the execution block and/or any portion of a transform. To illustrate how to extend EXTRACT modules, we will extend our `ExceptionAdder` example. In the example thus far, we add the specified exception type to the throws clause of each method in a Java source file. In this section, we will create a `TypedExceptionAdder` module that extends the `ExceptionAdder` module. In the `TypedExceptionAdder`, we examine the exception type already being thrown by each method and add our new exception type only if its superclass exception is not already being thrown.

To extend the functionality of the `ExceptionAdder`, it is only necessary to override the transform. The execution block, which selects all of the method declarations in the source file can remain as it is in the superclass. If we wanted to modify how the method declarations were selected (e.g., to select only public methods), we would need to override the execution block as well.

The source code for the `TypedExceptionAdder` is listed below:

```
module edu.wpi.cs.extract.examples.TypedExceptionAdder
extends edu.wpi.cs.extract.examples.ExceptionAdder
{
   transform AddException (String exceptionType) {
      (MethodDeclaration md) {
         if (!impl.containsSuperclass(GetAttrValue(md, “Throws”),
                                      exceptionType))
         {
            super.transform(md);
         }
      }
   }
}
```

Note the use of the `extends` clause in the module declaration. This specifies which module this module extends. We override the `AddException` transform to add the if-statement shown above. If the `containsSuperclass` method (which must be provided by the implementation) returns false, we perform the transform specified in the superclass. Note that the implementation (`TypedExceptionAdder_impl`) must provide a method with the following signature: `public boolean containsSuperclass(AttrValue throwsList, String exceptionType)`. This method, omitted for space, can then use Java reflection to see if the exception specified by `exceptionType` has a superclass that is specified in the `throwsList`.

Main Modules

The modules shown in the examples thus far are simple: one module performs the entire transformation. As transformations become more complex, as seen in the case studies discussed below, it becomes necessary to have multiple modules act together. For instance, we may want to take multiple passes over a set of source files. The first pass can collect information from the set of source files, and then the second pass can use that information to perform the transform. A mechanism is needed to chain multiple modules together like this. This is accomplished using a main module.

When a single module is executed, it is instantiated and then run on each of a set of source files. The transformed source files are then written back out to disk. However, it is often desirable to have more fine-grained control over how the module is executed.

A main module uses the same syntax as an ordinary module. However, there is no execution block, no properties, and no transforms defined. A single main execution block is specified instead. The syntax of this is as follows:

```
module MainModuleName {
   main(String[] args, SourceFile[] files) {
      // ...
   }
}
```

Consider our ExceptionAdder example. It would be sufficient to execute the module simply using Extract (the execution process is described in depth in a later section). This would be accomplished using the following command line:

```
java –jar Extract.jar edu.wpi.cs.extract.examples.ExceptionAdder ExceptionName
-- SourceFiles
```

where `ExceptionName` is the name of the exception to add, and `SourceFiles` is a list of source files to transform. However, we could define a main module to accomplish this:

```
module edu.wpi.cs.extract.examples.ExceptionAdderMain {
   main(String[] args, SourceFile[] files) {
      ExceptionAdder module = new ExceptionAdder();
      module.setParams(args);

      for (int i = 0; i < files.length; i++) {
         module.execute(files[i]);
      }
   }
}
```

This is a simple example that mimics how Extract would execute the ExceptionAdder module. However, this allows for more complex processing by transformations. For examples of this, see the case studies below.

Module Compilation

This section explains how EXTRACT modules are compiled into Java code. All modules (except main modules) extend the `edu.wpi.cs.extract.Module` class, summarized below. The `setParam` method saves the arguments passed to the execution block of the module in a protected object variable. Values for the arguments are passed in as an array of `Objects`. If the array of `Objects` is not of the correct size, or is any of the objects are not of the expected type, an `InvalidParamException` is thrown. The execution block is translated and placed in the body of the `execute` method.

```
package edu.wpi.cs.extract;

public abstract class Module {
   public void setParam(Object[] args) throws InvalidParamException;
   public void execute(SourceFile src) throws Exception;
   public void setProperty(String name, Object value);
   public Object getProperty(String name);
}
```

Main modules implement the `edu.wpi.cs.extract.MainModule` interface. This only defines the `main` method:

```
package edu.wpi.cs.extract;

public interface MainModule {
   void main(String[] args, SourceFile[] files) throws Exception;
}
```

To illustrate the compilation process, we will examine the code that is generated by compiling our `ExceptionAdder` example. The code for the ExceptionAdder module (ExceptionAdder.xm) is shown below:

```
module edu.wpi.cs.extract.examples.ExceptionAdder {
   ExceptionAdder (String exceptionType) {
      :: MethodDeclaration method = Select("/*/MethodDeclaration") {
         ApplyTransform(AddException(exceptionType), method);
      }
      { }
   }

   transform AddException (String exceptionType) {
      (MethodDeclaration md) {
         AddToList(GetAttrValue(md, "Throws"), exceptionType, -1);
      }
   }
}
```

This translates into the following Java code:

```
package edu.wpi.cs.extract.examples;

import edu.wpi.cs.jpath.*;
import edu.wpi.cs.extract.*;
import openjava.ptree.*;
import java.util.Iterator;

public class ExceptionAdder extends Module implements ExceptionAdder_props {
    protected ExceptionAdder_impl impl;

    public ExceptionAdder() {
        impl = new ExceptionAdder_impl();
        impl.init();
    }

    protected void finalize() throws Throwable {
        impl.dispose();
        super.finalize();
    }

    protected String exceptionType;

    protected boolean initialized = false;

    public void setParams( Object[] args ) throws InvalidParamException {
        if (args.length != 1) {
            throw new InvalidParamException();
        }
        if (!(args[0] instanceof String)) {
            throw new InvalidParamException();
        } else {
            exceptionType = (String) args[0];
        }
        initialized = true;
    }

    public void execute( SourceFile src ) throws Exception {
        if (!initialized) {
            throw new InvalidStateException( "Module not initialized" );
        }
        Selection sel1 = JPathAPI.select( src.getCompilationUnit(),
                                          "/*/MethodDeclaration" );
        Iterator it1 = sel1.iterator();
        if (!it1.hasNext()) {
        }
        while (it1.hasNext()) {
            MethodDeclaration method = (MethodDeclaration) it1.next();
            {
                AddException transform =
                   new AddException( exceptionType );
                transform.transform( method );
            }
        }
    }

    public static class AddException implements Transform {

        protected String exceptionType;

        public AddException( String exceptionType ) {
            this.exceptionType = exceptionType;
        }

        public void transform( MethodDeclaration md ) throws ModuleException
        {
            ExtractAPI.addToList( JPathAPI.getAttrValue( md, "Throws" ),
                                  exceptionType, -1 );
        }
    }
}
```

The constructor for the module class creates and saves the implementation instance, as well as calls the implementation's `init` method. Similarly, the module's `finalize` method calls the implementation's `dispose` method. The module's sole parameter (`String exceptionType`) is a protected field. It is set in the `setParams` method. Note how `setParams` determines whether the parameters are correct. The execution block is translated into proper Java code and inserted as the body of the `execute` method. It begins by ensuring that the module was initialized with the correct parameters. If the module was not properly initialized, an `InvalidStateException` is thrown.

Transforms are translated into static inner classes belonging to the module. This allows for the transform to be extensible. Transforms implement the `edu.wpi.cs.extract.Transform` interface. The `AddException` transform takes a `String` argument called `exceptionType`. This is passed to the constructor of the transform object and is saved in a protected field.

Each transform block is translated into a method named transform. The parameters to this method correspond to the types given at the beginning of the transform block. As stated above, a transform declaration can define multiple transform blocks. Each transform block corresponds to a transform method whose parameter types match the transform block's type.

Note that the `ApplyTransform`, `AddToList`, and `GetAttrValue` functions that are called in the EXTRACT module are translated into the appropriate Java method calls. `ApplyTransform` expands into a series of statements which instantiate the transform class and call the transform method. `AddToList` translates to the

`ExtractAPI.addToList` method and `GetAttrValue` translates to `JPathAPI.getAttrValue`.

Module Execution

Modules are executed using the edu.wpi.cs.extract.Extract class. This class can be run from the command-line using the following syntax:

```
java edu.wpi.cs.extract.Extract [-o output_dir] module [module_args] -- files
```

When executing, the following steps are performed. First, the module is loaded (using `java.lang.Class.forName`). Next, the source files (specified by `files`) are parsed and ASTs are constructed.

If the module being executed is a main module, the `main` method is called with `module_args` and the parsed source files as arguments. If the module being executed is not a main module, `setParams` is called with `module_args` as arguments, then `execute` is called for each of the parsed source files.

After the module has executed, the transformed source files are saved out to disk. If an output directory is specified (using the `-o` option), the source files are written to that directory. Otherwise, the source files are written to the `transformed` sub-directory of the current directory (which is created if it does not exist). If the transformed files are part of a package, the appropriate package directories are created.

In order to keep EXTRACT modules simple, they do not have any exception handling capabilities. If an exception occurs during the execution of an EXTRACT module, the module is considered to have failed and processing stops for the given source file. Execution continues on the next file.

Chapter 6
Evaluation

To evaluate EXTRACT, we chose three case studies. The following three chapters examine these case studies. In our first case study, we developed a type qualifier. In Java code, developers often import classes and packages, allowing them to use short names instead of fully-qualified class names (e.g., `Vector` instead of `java.util.Vector`). It is necessary for certain transformations and certain types of source code analysis to have fully-qualified names. Therefore, we developed the type qualifier module.

In our second case study, we developed a behavioral contract checker, based on the work of Findler, et al [13]. In [13], the authors present a mechanism for performing run-time checks on pre- and post-condition contracts. The authors discuss constructing a special compiler that would add the appropriate contract-checking byte code to already-compiled Java code. However, it is not clear that this implementation was completed, and the publication is two years old. We were able to develop a contract checker using EXTRACT that inserts the contract-checking code into Java source code.

In our third case study, we developed a code obfuscator. It is not difficult to decompile Java byte code back into source code. There are many situations, however, where this would not be desirable. Code obfuscation allows a developer to take a set of Java source files and mangle different symbols (e.g., method or variable names) so they are meaningless to someone who would decompile the code.

Chapter 7
Case Study: Type Qualifier

Our first case study is a type qualifier. When writing Java code, developers often use import statements to avoid having to use fully-qualified type names in the code. For example, importing `java.util.Vector` or `java.util.*` allows the developer to use the Vector class without the "`java.util.`" prefix. While this is convenient for the developer, it somewhat complicates the analysis of source code. It is often useful to translate type names into fully-qualified names for the purpose of transformation.

In order to accomplish this, we need to take a two-pass approach. First, we process all of the source files to register all of the classes defined therein. Given each source file, we can retrieve the package name of that file and the names of all of the classes defined in that file. In the second pass, we examine each file, record its import statements, and then resolve each type name to a fully qualified type name.

We begin by defining a main module that controls the execution of the two passes:

```
Main.xm

module edu.wpi.cs.extract.examples.qualifier.Main {
    main(String[] args, SourceFile[] files) {
       // first, register all types contained in the sources being
       // processed

       // create a local class registrator
       LocalClassRegistrator lcr = new LocalClassRegistrator();

       // initialize it
       lcr.setParams(new Object[0]);

       // run the registrator on each of the files
       for (int i = 0; i < files.length; i++) {
           lcr.execute(files[i]);
       }

       System.out.println("LocalClassRegistrator done.");

       // second, resolve type names using the local names found
       // above and the import statements at the beginning of each
       // file

       // create a type resolver
       TypeResolver tr = new TypeResolver();

       // requires an IClassRegistry_props as a parameter
       Object[] params = { lcr };

       // set the parameters
       tr.setParams(params);

       // run the resolver on each of the files
       for (int i = 0; i < files.length; i++) {
           tr.execute(files[i]);
       }

       System.out.println("TypeResolver done.");
       System.out.println("Main done.");
    }
}
```

On the first pass, we execute the `LocalClassRegistrator` module over all of the source files. This registers all classes defined in the source files being processed with their fully-qualified names. On the second pass, we execute the `TypeResolver` module over all of the source files. This uses the information from the `LocalClassRegistrator` and import statements to resolve type names.

We define a set of properties, in a module called IClassRegistry:

```
IClassRegistry.xm

module edu.wpi.cs.extract.examples.qualifier.IClassRegistry {
    properties {
       /**
        * Provides a mapping from a class name (e.g., String) to a
        * fully-qualified class name (e.g., java.lang.String).
        */
       java.util.Hashtable nameResolution;
    }
}
```

This provides a hashtable which maps from a short class name (e.g., `String`) to a fully-qualified class name (e.g., `java.lang.String`). Our `LocalClassRegistrator` imports this set of properties:

```
LocalClassRegistrator.xm

import java.util.Hashtable;

module edu.wpi.cs.extract.examples.qualifier.LocalClassRegistrator {
    properties {
       // we provide this interface
       // Hashtable nameResolution;
       import IClassRegistry;
    }

    LocalClassRegistrator () {
       // make sure the nameResolution property is initialized
       if (getProperty("nameResolution") == null) {
           setProperty("nameResolution", new Hashtable());
       }

       // load the property
       Hashtable table = get_nameResolution();

       // primitive types
       table.put("boolean", "boolean");
       table.put("byte", "byte");
       table.put("char", "char");
       table.put("short","short");
       table.put("int", "int");
       table.put("long", "long");
       table.put("float", "float");
       table.put("double", "double");
       table.put("void", "void");

       // this file's package name
       String packageName = null;

       // select the compilation unit
       :: CompilationUnit cu = Select("/") {
           // get the package name
           ScalarValue pkg = GetAttrValue(cu, "Package").scalarValue();

           // if the package name exists, save it
           if (!pkg.isNull()) {
              packageName = pkg.toString();
           }
       } { }

       // select all class declarations
       :: ClassDeclaration cd = Select("/*/ClassDeclaration") {
           // save the class name
           String className = GetAttrValue(cd,
                                            "Name").scalarValue().toString();

           // map the class name to it's fully-qualified name
           if (packageName != null) {
              table.put(className, packageName + "." + className);
           }

       } { }

       // save the property
       set_nameResolution(table);
    }
}
```

This module is relatively simple. First, we ensure that our property is properly initialized. Next, we add the Java primitive types to the name resolution hashtable.

Then, we select the compilation unit and retrieve the package name. If one exists, it is saved; otherwise, we assume the default package. Finally, we select each of the class declarations in this file and register them in the name resolution hashtable using the package name for the file.

On the second pass, we execute the `TypeResolver` module. Code for the `TypeResolver` is found below:

```
TypeResolver.xm

import java.util.*;

module edu.wpi.cs.extract.examples.qualifier.TypeResolver {
    TypeResolver (IClassRegistry_props registry) {
      // on-demand imports
      Vector odi = new Vector();
      // class imports - maps name -> fq name
      Hashtable ci = new Hashtable();

      // always import java.lang
      odi.addElement("java.lang");

      // process the imports
      :: CompilationUnit cu = Select("/") {
          // get the list of imports
          ListValue imports = GetAttrValue(cu,
                                           "DeclaredImports").listValue();

          // loop through the imports
          for (int i = 0; i < imports.length(); i++) {
            String impt = imports.getAt(i).toString();

            // if the import ends with a .*, it's an on demand import
            if (impt.endsWith(".*")) {
                odi.addElement(impt.substring(0, impt.length() - 2));
            }
            // otherwise, it's a class import
            else {
                String name = impt.substring(impt.lastIndexOf('.') + 1,
                                             impt.length());
                ci.put(name, impt);
            }
          }

      } { }

      // give the implementation the import information
      impl.setImports(odi, ci);
      impl.setRegistry(registry);

      // select type names for the transformation
      :: TypeName oldType = Select("/*/TypeName") {
          String fqType = impl.resolveName(oldType);

          TypeName newType = ExtractAPI.createTypeName(fqType);

          ApplyTransform(SubstituteTypeName(), oldType, newType);
      } { }
    }

    transform SubstituteTypeName () {
      (TypeName oldName, TypeName newName) {
          ExtractAPI.replace(oldName, newName);
      }
    }
}
```

The TypeResolver requires a reference to an IClassRegistry_props object. Recall that the LocalClassRegistrator contains properties of this type; specifically, it provides the name resolution hashtable. The TypeResolver creates two local objects; a Vector for on-demand imports (e.g., java.util.*), and a hashtable for

class imports (i.e., `import java.util.Vector` would place the following pair in the hashtable: `Vector -> java.util.Vector`). We then add the default on-demand import of `java.lang.*`. Next, we examine the declared imports of the compilation unit. If the import statement ends with a "`.*`", we add it to the on-demand imports. Otherwise, it is a class import and we add it to the hashtable.

Once we are done processing the imports, we send the implementation a reference to the import vector and hashtable as well as the `IClassRegistry_props` object. Finally, we select all `TypeNames` in the source file. We then use the implementation to resolve the name to a fully-qualified name (the implementation class has been omitted for space) and use the `SubstituteTypeName` transform to replace the original `TypeName` with a fully-qualified `TypeName`.

In the implementation class (`TypeResolver_impl`), we use Java reflection to resolve type names. The only weakness to this approach is that it requires that all source files that are being transformed are already compiled.

Chapter 8
Case Study: Behavioral Contract Checking

In the paper Behavioral Contracts and Behavioral Subtyping [13], Findler, et al, present a mechanism for performing run-time checks on pre- and post-condition contracts. Contracts are stated after method declarations as follows (example from [13]):

```
interface I {
  int m(int a);
    @pre { a > 0 }
}
```

A contract compiler generates classes to enforce contracts on interface methods and generates wrapper methods to enforce contracts on class methods (this process is described in more detail below). The implementation described by the authors augments `.class` files generated for each interface with information that will insert the appropriate byte-code into all classes that implement that interface.

The effort described by the authors in [13] includes constructing a special compiler to add the appropriate support code to implement each contract condition, and then modifying byte-code so that the contract checking code is called. It is not clear that this implementation was completed, and the publication is two years old.

To demonstrate the benefits of using EXTRACT, we have implemented a contract checking system analogous to the system presented in [13] using only the EXTRACT system. Our intent is to show that the effort required to implement this system in EXTRACT is considerably less than the effort described by the authors of [13].

The Behavioral Subtyping Condition

Behavioral subtyping ensures that all objects of a subtype preserve all of the original type's invariants. Thus, objects of a subtype are substitutable for objects of a supertype without any effect on the program's observable behavior. When evaluating pre- and post-conditions, care must be taken to ensure that contracts are enforced. The authors of [13] point out that all previous contract checkers for Java fail to handle the behavioral subtyping condition correctly. Simply put, for a given method the subtype's pre-condition may be stricter than the base type's and the subtype's post-condition may be less strict that the base type's. The figures below (taken from [13]) describe the behavioral subtyping condition and how contracts need to be checked.

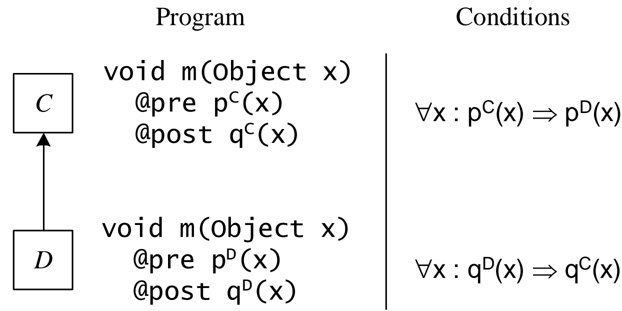

Figure 9: The Behavioral Subtyping Condition

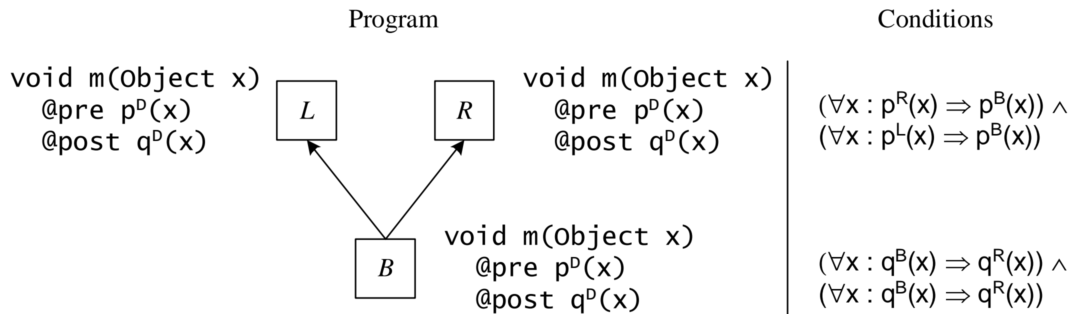

Figure 10: The Behavioral Subtyping Condition, Generalized to Multiple Inheritance

Given a method with pre- and post-conditions, checker methods are generated which perform the condition check. Since the conditions rely on the context in which a method is called, a checker method is generated for each context that the method can be called in. For instance, a method `m` in class `C` which implements interface `I` can be called from objects of type `C` or `I`. Consequently two checker methods, `m_C` and `m_I`, are generated.

In these checker methods, we first evaluate the method's pre-condition in the context of the appropriate class (i.e., `m_C` would check m's pre-condition in class `C`; `m_I` would check m's pre-condition in interface `I`). If the pre-condition fails, the calling class is blamed and the program exits.

Hierarchy checking methods are generated for all interface and class methods. The new methods are directly inserted into classes. Checker classes, containing the hierarchy checking methods, are generated for interfaces. After the pre-condition described above is performed, the pre-condition checking recursively traverses the class and interface hierarchy to see if the behavioral subtyping implication holds. If it does not, the hierarchy is malformed; the hierarchy is blamed for the contract failure and the program exits.

Next, the original method is called. In the examples presented in [13], none of the methods being checked return a value; all are of type void. The mechanism presented does not lend itself easily to the checking of return values. This reduces the power of the post-condition checking facility.

After the original method is called, we evaluate the method's post-condition in the context of the appropriate class. If the post-condition fails, the declaring class (which contains the method) is blamed and the program exits.

Finally, the post-condition checking recursively traverses the class and interface hierarchy, in the same order as it did in the pre-condition checking, to see if the behavioral subtyping implication holds. If it does not, the hierarchy is malformed; the hierarchy is blamed for the contract failure and the program exits.

In order for the contract checking code to be called, all method calls must be transformed to reflect the context that it is being called by. For instance, consider a method `m` in class `C` which implements interface `I`. If `m` is called from an object of type `C`, the call should now be `m_C`; likewise, if `m` is called from an object of type `I`, the call should now be `m_I`.

As described above, [13] states pre- and post-conditions after a method declaration. This is not desirable, however, because the resulting code will not compile under a standard Java compiler. For our purposes, we embed the pre- and post-conditions in the Javadoc comment for that method (under the `@pre` and `@post` tag). This can contain any arbitrary Java code; however, return values of a method unfortunately cannot be accessed by the post-condition.

The EXTRACT modules to perform the necessary transformations progress as follows. First, we collect all pre- and post-conditions from the methods' Javadoc comments. If no condition is defined, we assume it to be `true`. Next, we generate the necessary checker code. Checker methods are generated for classes and checker classes

are generated for interfaces. After that, type analysis, similar to that done by the code obfuscator, is performed. This allows us to determine under what context a method is being called (i.e., the type of the object that the method is being called on). We then transform each method call to reflect this context.

This case study was implemented in less than a week using EXTRACT technology. As stated above, after two years, the implementation promised in [13] has yet to be seen. Our implementation was accomplished using five modules containing a total of four transforms.

Chapter 9
Case Study: Code Obfuscator

Java byte code is easily decompiled. The `javap` utility that is included in the Java Development Kit translates Java byte code into Java assembly code. Other software exists that decompiles Java byte code into Java source code that is almost identical to the original code. There are many situations where this is undesirable.

Our code obfuscator addresses this issue. It takes a Java class and mangles its method and field names. The resulting code is almost impossible for a human to understand, yet it maintains the semantic content of the original code. Only method names that can be mangled are; methods that are required by an external interface (e.g., `actionPerformed` in `java.awt.event.ActionListener`) are not changed.

Execution of the code obfuscator proceeds as follows. First, the set of package names contained in the processed source files is recorded. Next, all interfaces and classes are processed to determine which methods are in scope and can be changed. Methods that are not part of an external interface or superclass cannot be obfuscated because this would break the inheritance hierarchy.

Before completing the obfuscation, it is necessary to generate a symbol table for the Java source files being processed. While EXTRACT does not support this directly, the underlying OpenJava library does. Thus, a symbol table is created allowing us to perform Java type analysis on the AST.

Finally, we obfuscate the code. First, all method calls are selected. We use the symbol table information to determine the type of the object that the method is being called on. Using the scope analysis performed in the first step of the obfuscator, we

determine whether to obfuscate this call. After that, we examine all of the class and interface declarations being processed. Here, we rename the original method declarations to the new obfuscated names, again using the earlier scope analysis.

The code obfuscator is being developed by Professor George Heineman, this project's advisor. It is in its final stages of development, and should be available very soon. Preliminary evaluations show that the obfuscator works properly under all test cases chosen thus far. It has been used to obfuscate a number of large Java applications, including the EXTRACT software itself.

Chapter 10
Conclusion

Having developed EXTRACT and then using it to create the case studies described above, we have made a number of observations regarding the technology and how it can be improved. This chapter discusses these observations and examines areas for future work.

First, the AST generated by OpenJava does not contain symbol table information. This is often useful, as seen in the contract checker and code obfuscator. However, type analysis is not always necessary and would cause an unnecessary performance penalty for transformations that do not need it. The code necessary to perform type analysis on an OpenJava AST was written for the contract checker and code obfuscator. In the future, it is possible to add this code to the EXTRACT library.

EXTRACT is closely tied to OpenJava. If the underlying AST library was completely abstracted, it would be possible to apply EXTRACT to languages other than Java. However, exposing the OpenJava library allows developers to write more sophisticated transforms.

JPath expressions, like regular expressions, are limited in their expressiveness. Often times, a more expressive mechanism for selection AST nodes would be useful. For instance, consider a Java source file that contains inner or anonymous classes. Evaluating a JPath expression to select all class declarations (i.e., `/*/ClassDeclaration`) would select all classes in the file without easily being able to determine which are inner or anonymous classes. The selection mechanism used by EXTRACT is arbitrary; another can be developed and used in its place.

JPath allows for analysis of selections, and thus analysis of modules. One can examine a JPath expression and determine what parts of the source tree will be modified. Using this information, it is possible to determine if and when two modules will conflict. For instance, consider one module that changes all for-loops to while-loops, and another module that removes all method bodies to create an interface. Obviously, these two modules conflict. Knowing this, it is possible to determine a partial ordering of modules.

This thesis has presented technology that allows for the creation of arbitrary transformations on Java code. We begin with a library (OpenJava) which parses Java source code and generates an AST. Given the AST, we provide JPath as a mechanism for determining which parts of the AST are to be transformed. On top of this technology, we provide EXTRACT as a means of expressing transformations on the AST.

A number of supporting libraries and programs have been developed. The EXTRACT compiler (`ExtractC`) translated EXTRACT code into Java code. The EXTRACT API provides an interface to the code fragment parsing facilities provided by OpenJava, and provides a number of tree modification routines for use in modules. The JPath API provides facilities to evaluate JPath expressions, retrieve attribute data from an OpenJava AST, and wraps various types of attribute data in scalar and list values.

Finally, we present three case studies to demonstrate the effectiveness of EXTRACT. In our type name qualifier, we developed a standard transformation that is often a prerequisite for other transformations. We provided an implementation for the behavioral contract checking mechanism presented in [13]. Finally, our code obfuscator

transforms entire Java applications, making method and field names unreadable to a human while maintaining the semantic content of the code.

References

[1] ADAPT Project. <http://www.cs.wpi.edu/~heineman/adapt/>.

[2] R. Balzer, N. Combs, D. Garlan, P. Gross, G. Heineman, G. Kaiser, B. Schmerl, D. Wells, and D. Wile. An Infrastructure for Instrumenting, Measuring, and Controlling Software. DASADA Whitepaper, 2002.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] B. Bokowski. Barat – A Front-End for Java. Technical Report B-98-09, Freie Universitat Berlin, 1998.

[5] D. Box, A. Skonnard, and J. Lam. *Essential XML: Beyond Markup*. Addison-Wesley, 2000.

[6] S. Chiba. Load-Time Structural Reflection in Java. In *ECOOP – Object-Oriented Programming, LNCS 1850*, pages 313-336, 2000.

[7] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, World Wide Web Consortium, 1999. <http://www.w3.org/TR/xslt>.

[8] K. T. Claypool, E. A. Rundensteiner, and G. T. Heineman. Extending Schema Evolution to Handle Object Models with Relationships. Technical Report WPI-CS-TR-99-15, Worcester Polytechnic Institute, Compter Science Department, 1999.

[9] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, World Wide Web Consortium, 1999. <http://www.w3.org/TR/xpath>.

[10] Document Object Model (DOM) Level 1 Specification Version 1.0. W3C Recommendation, World Wide Web Consortium, 1998.

[11] Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA) Program Website. <http://schafercorp-ballston.com/dasada/index1.html>.

[12] H. Dörr. *Efficient Graph Rewriting and Its Implementation, LNCS 922*. Springer-Verlag, 1995.

[13] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral Contracts and Behavioral Subtyping. *Foundations of Software Engineering*, FSE 2001.

[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1995.

[15] P. Gill. Probing for a Continual Validation Prototype. MS Thesis. Worcester Polytechnic Institute. 2001.

[16] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[17] G. T. Heineman. A Model for Designing Adaptable Software Components. In *22nd Annual International Computer Science and Application Conference (COMPSAC-98)*, pages 121-127, Vienna, Austria, 1998.

[18] Java 2 Platform, Standard Edition (J2SE). <http://java.sun.com/j2se/>.

[19] Java TreeBuilder. <http://www.cs.purdue.edu/jtb/docs.html>.

[20] JavaAssist. <http://www.csg.is.titech.ac.jp/~chiba/javaassist/>.

[21] javacc - Java Compiler Compiler <http://www.webgain.com/products/java_cc/>.

[22] JJTree. <http://www.webgain.com>.

[23] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Wiswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[24] B. Kurtz. SoftViz. MS Thesis. Worcester Polytechnic Institute. Forthcoming.

[25] OpenJava. <http://openjava.sourceforge.net/>.

[26] ProbeMeister. <http://www.objs.com/DASADA/ProbeMeister.htm>.

[27] rmic – The Java RMI Stub Compiler. <http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/rmic.html>.

[28] Siena: A Wide-Area Event Notification Service. <http://www.cs.colorado.edu/~carzanig/siena>.

[29] M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pages 56-60, 1998.

[30] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering, LNCS 1826*, pages 119-135. Springer-Verlag, 2000.

[31] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of 'Legacy' Java Software. In *ECOOP – Object-Oriented Programming, LNCS 2072*, pages 236-255, 2001.