TOWARDS AN AST-BASED APPROACH TO REVERSE ENGINEERING

Xin Wang Xiaojie Yuan
College of Information Technical Science, Nankai University, China
email: wang_xin@mail.nankai.edu.cn email: yuanxj@nankai.edu.cn

Abstract

Today, it is recognized that reverse-engineering activities play an important role in round-trip development and software maintenance. However, the precision of most existing reverse-engineering tools cannot meet the requirements of developers and maintainers, thus hampering understanding of software implementation.

This paper presents an approach for recovering the UML class diagram from the Java source code, which traverses abstract syntax trees (AST) with the Visitor pattern to build the corresponding UML model elements. The source code is completely and systematically analyzed by this approach. As a result, the precision of the generated design model will be substantially promoted.

Keywords: *Reverse engineering; AST; UML model.*

1. Introduction

Reverse engineering has become an indispensable link in the process of round-trip engineering, and it aims at producing design models from software implementation details. As pointed out by Chikofsky and Cross [1], the main purpose of reverse engineering is to create representations of the subject system at a higher level of abstraction. Thus, the design models reverse-engineered by automatic tools should help program understanding.

However, the reverse-engineering facilities provided by most existing CASE tools are not usually precise enough to satisfy developers and maintainers, thus hampering program understanding. This is because analyzing the source code completely and systematically is not a trivial task, and there is no one-to-one mapping between design models and the source code.

Ideally, design models should contain enough information to regenerate a program that is equivalent to the original source program [2]. We believe that this can be achieved only if the source code is fully parsed and the design model is built by analyzing the AST completely.

In this paper, we present an approach for recovering the UML class diagram from the Java source code. This approach is based on the fine-grained analysis of the AST that is created by a real parser from each compilation unit in a Java project. To demonstrate this AST-based approach, a research prototype is implemented as a plug-in for the Eclipse platform. As a result, we add the reverse-engineering capability to the Integrated Development Environment (IDE).

The rest of the paper is organized as follows. In the next section, we describe the mechanism for mapping the Java source code to the corresponding design model elements. In section 3, we show an example session. Finally section 4 concludes the paper.

2. From Source Code to Design Model

The Eclipse platform (available at http://www.eclipse.org/) has an open, extensible architecture based on plug-ins, and provides a set of fundamental tools that can facilitate the reverse-engineering process. Our tool is dependent on Eclipse's JDT and UML2 plug-ins. The JDT core provides access to the AST of a Java compilation unit, while the UML2 is an implementation of the UML 2.0 meta-model that can be used to build the design model.

2.1. Tool Structure

To analyze compilation units, source files must be obtained from a given project. In the Eclipse platform, a Java project is essentially a tree structure that is defined in terms of files and folders. In the meanwhile, a compilation unit can be represented as an abstract syntax tree. Traversing the tree structure is a typical use of the Visitor pattern [3].

On the other hand, the UML model should be created incrementally while the source code being visited, and the process of building a model should be independent of the elements that make up the model. In this case, Builder [3] is the proper pattern for the model creation. Thus, a combination of the Visitor pattern and the Builder pattern is applied to the tool construction.

The basic structure of our tool is composed of three main parts, as shown in Figure 1.

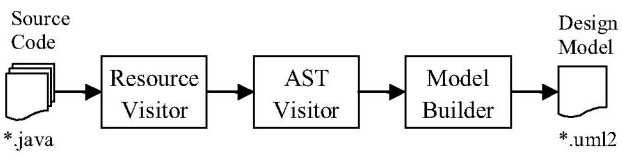


Figure 1. Tool structure

- The **resource visitor** traverses the Java project specified by the user. Whenever a Java compilation unit is visited, the AST visitor will be called to do the further analysis.
- The **AST visitor** traverses the abstract syntax tree of a compilation unit. It will examine the node types of interest down to the statement level, resolve names to their binding information when necessary, and direct the model builder to construct the UML class diagram.
- The **model builder** provides an interface that can be used for building the UML model step by step.

Apparently, the AST visitor is the key component because it implements the algorithms that map each type of nodes to the corresponding model elements. We will describe how to build several fundamental model elements in the following sections.

2.2. Packages and Types

A package is a group of elements with a common theme. It partitions a model, making it easier to understand and manage [5]. The AST visitor notifies the model builder to create a new package when a package declaration statement is encountered for the first time. An instance variable is used to keep track of the current enclosing package of compilation units. Algorithm 1 shows the process of building packages in the design model.

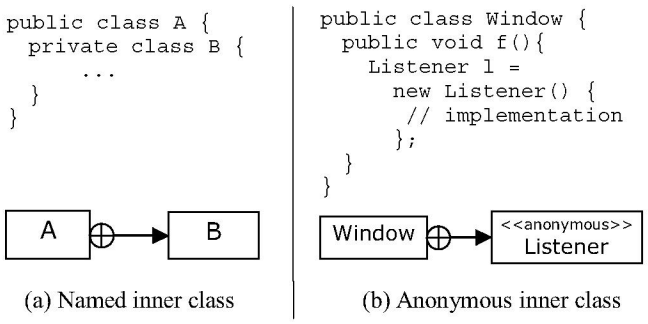
Algorithm 1 BuildPackage

Input: *CU* is a set that contains all compilation units in a project;
builder is the model builder.

```
1: Let P be a hash set;
2: current ← empty string;        // the current enclosing package
3: for each element c ∈ CU do
4:    name ← the enclosing package name of compilation unit c;
5:    if name is not contained by P then
6:       P ← P ∪ name;
         // builds a new package element in the model.
7:       builder.buildPackage(name);
8:    end if
9:    current ← name;
10: end for
```

After recognizing a package, the AST visitor continues to examine each compilation unit that the package contains. Basically, a type declaration is the main part of a compilation unit. In Java, a type declaration declares either a class or an interface, both of which are the focus of the object-oriented modeling. The corresponding AST node provides enough information about the properties of the type declaration. By querying the AST node, we can determine whether it declares a class or an interface. For a class, we can also test whether it is abstract or not.

In order to reflect the entire class diagram of a Java project, we must also recognize inner classes that are class declarations within the scope of another class. Inner classes are an implementation specific feature that cannot be represented with a standard UML model element.



(a) Named inner class (b) Anonymous inner class

Figure 2. Inner classes

Custom notations are used by Martin in [4] to denote inner classes. A named inner class is represented with an association adorned with a crossed circle, as show in Figure 2(a). An anonymous inner class is represented as a nested class that is given the <<anonymous>> stereotype, and is also given the name of the interface it implements, as shown in Figure 2(b).

Handling inner classes properly requires the AST visitor to store the names of the enclosing classes in a stack. When a class declaration node is about to be visited, the class name is pushed onto the stack. After all of the node's contents have been visited, the class name is popped out of the stack. Algorithm 2 details the process of building types in the model.

Algorithm 2 BuildType

Input: *node* is an AST node for type declaration;
pkg is the enclosing package of this compilation unit;
builder is the model builder.

```
1: Let S be a stack; // storing the names of the enclosing classes
2: function visit(node)
3:    name ← the name of node;    // gets the name of the type
4:    S.push(name);
5:    if node is an interface declaration then
         // builds a new interface element in the model.
6:       builder.buildInterface(pkg, name);
7:    else if node is a class declaration then
8:       if node is an abstract class then
9:          isAbstract ← true;
10:      else isAbstract ← false;
11:      end if
12:      cls ← S.top();      // gets the enclosing class
13:      builder.buildClass(pkg, cls, name, isAbstract);
14:   end if
15: function endVisit(node)
16:   S.pop();
```

2.3. Attributes and Operations

An attribute is a named property of a class that describes a value held by each object of the class. An operation is a function or procedure that may be applied to or by objects in a class [5]. Though attributes and operations may be of less importance at some higher level of abstraction, they serve to

elaborate the dominant modeling elements, such as classes and relationships, when detail features are concerned.

Because of the differences in concepts, Java fields and methods cannot be directly mapped to UML attributes and operations respectively. Instead, interpretations are necessary for the extraction of attributes and operations from the source code. Since constructing abstractions requires understanding and is thus known to be a process that should be carried out (semi-)manually [6], our tool provides a configuration file that allows the user to customize the mapping rules.

The default rules for building attributes and operations are specified in our tool as follows. (To simplify the discussion, we only consider the mapping rules about Java public methods in this paper.)

Rule a) If both a *getter* and a *setter* that matches the *getter* are recognized, they are interpreted as an attribute.

Rule b) If only a *getter* is recognized, it is interpreted as a read-only attribute.

Rule c) If only a *setter* is recognized, it is interpreted as an operation.

Rule d) Other public methods are directly mapped to operations.

We define a getter as a public method that must have the string "get" as the prefix of its name. Moreover, the return type of a getter must be a primitive type and cannot be of type "void". Finally, the parameter list of a getter must be empty.

In contrast to a getter, a setter is defined as a public method that must have the string "set" as the prefix of its name. The return type of a setter must be of type "void". The parameter list must contain only one parameter, and the type of the only parameter must be the same as the return type of the corresponding getter.

As an example, Figure 3 shows a typical Java class declaration with fields and methods, as well as a UML class notation with attributes and operations that are extracted from the code snippet according to the above rules.

```
public class Window {
  private int state;
  private String name;
  public int getState(){
    return state;
  }
  public void setState(int s){
    state = s;
  }
  public String getName(){
    return name;
  }
  public void setTitle(int t){
  }
  public void doSomething(){
  }
}
```

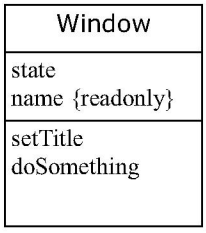


Figure 3. Attributes and operations

2.4. Generalization

Generalization is the relationship between a class (i.e. the superclass) and one or more variations of the class (i.e. the subclasses). It is important to identify generalization relationships for class modeling, because generalization allows the developer to organize classes into a hierarchical structure that facilitates the understanding of the subject system.

Once a class has been recognized and built in the model, the AST visitor will check if it has a superclass. A straightforward way is to see if there is the keyword `extends`, and the string that follows `extends` is the name of the superclass. However, if the string is just a simple type name (not a qualified name), we cannot determine which package the superclass belongs to. Though tools based on fact extractors can rescan and analyze the import declaration statements to obtain the fully qualified name of the superclass, we believe that this approach is tedious and thus inefficient.

Fortunately, the built-in Java language parser in the Eclipse's JDT plug-in can create abstract syntax trees with *bindings*. These bindings draw connections between the different parts of a program. The various names and types appearing in the AST can be easily resolved to their bindings. Obviously, this feature is a powerful tool for developers who wish to analyze the structure of a program more deeply.

With the help of the binding information, the AST visitor navigates to the compilation unit where the superclass is declared. In doing so, we can query a number of properties about the superclass, such as the unqualified name, the fully qualified name and the name of the enclosing package. Note that if the subclass and the superclass are located in two different packages, the AST visitor will create a placeholder to represent the superclass in the package where the subclass is declared. The placeholder refers to the actual superclass in the other package. (The placeholder may be a class icon that contains only the fully qualified class name.) Algorithm 3 outlines the process of building generalization relationships in the model.

Algorithm 3 BuildGeneralization

Input: *node* is an AST node for type declaration;
pkg is the enclosing package of this compilation unit;
builder is the model builder.

1: *subName*←gets the name of the subclass represented by *node*;
2: *type* ← gets the superclass of *node*;
3: **if** *type* != **null then** // if *node* has a superclass.
// resolves the superclass by bindings.
4: *binding* ← resolves and returns the binding for *type*;
5: *fullName* ← gets the fully qualified name of the superclass;
6: *pkgName* ← gets the enclosing package of the superclass;
7: *superName* ← gets the unqualified name of the superclass;
8: **if** *pkgName* == *pkg* **then** // in the same package.
// build the superclass, if necessary.
9: *builder.buildClass*(*pkg*, *superName*, ...);
10: **else** // if the superclass is in another package.

```
11:       builder.buildClassPlaceholder(pkg, fullName, ...);
12:       superName ← fullName;
13:    end if
14:    builder.buildGeneralization(pkg, subName, superName);
15: end if
```

3. An Example Session

In this section, we demonstrate an example session in which the Draw2d plug-in will be reverse-engineered by our tool. The Draw2d plug-in shipped with Eclipse is a lightweight toolkit for graph drawing and layout. Draw2d has been selected because its scale is moderate and its class hierarchy is more typical. As a maintainer of Draw2d, you can carry out the following steps to obtain the UML class diagram.

- In the Package Explorer view, select the Draw2d project. Click "**Generate Class Diagrams**" button in the toolbar.
- The wizard for creating UML class diagrams will be opened. Make sure that the "**Automatic Layout**" option is checked. Then click "**Finish**".

The process of reverse engineering takes 30 seconds on a Windows machine with an 866 MHz Pentium III CPU and 384 MB main memory. The Draw2d Java project consists of 244 compilation units currently. All of them are parsed and analyzed by our tool. Finally, the class diagrams generated according to the default mapping rules contain 8 packages, 210 classes (including inner classes), 34 interfaces, 185 attributes, 1365 operations, and 156 generalization relationships, as shown in Figure 4.

4. Conclusions

In this paper, we have described an AST-based CASE tool for reverse engineering. The key component of our tool is the AST visitor that implements the algorithms for mapping the Java source code constructs to the design model elements. Since every compilation unit is fully parsed and analyzed, the AST represents the entire structure of the source code. Therefore, the AST visitor can query any kind of properties it needs to build the design model. As a result, the design model generated by the AST-based approach is more precise and thus more useful than that generated by the conventional approach.

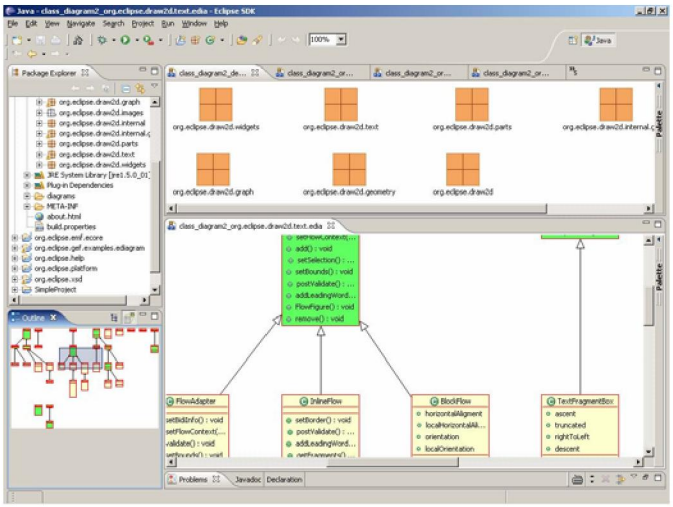


Figure 4. Generated class diagrams of Draw2d

References

[1] E.J. Chikofsky and J.H. Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software,* vol. 7, no. 1, pp. 13-17, January, 1990.

[2] Y. Lin, R.C. Holt, and A.J. Malton, "Completeness of a fact extractor", In *Proceedings of 10th Working Conference on Reverse Engineering (WCRE 2003),* pp. 196-205, IEEE Computer Society, Washington DC, USA, November, 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE.* Boston MA, USA: Addison-Wesley, 1995.

[4] R. C. Martin, *UML FOR JAVA PROGRAMMERS.* Englewood Cliffs NJ, USA: Prentice Hall, 2003.

[5] M. Blaha and J. Rumbaugh, *OBJECT-ORIENTED MODELING AND DESIGN WITH UML, SECOND EDITION.* Englewood Cliffs NJ, USA: Prentice Hall, 2005.

[6] R. Kollmann, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf, "A study on the current state of the art in tool-supported UML-based static reverse engineering", In *Proceedings of 5th Working Conference on Reverse Engineering (WCRE 2002)*, pp. 22-32, IEEE Computer Society, Washington DC, USA, November, 2002.