Improving the Accuracy of UML Class Model Recovery

Kun Wang and Wuwei Shen
Department of Computer Science
Western Michigan University
Kalamazoo, Michigan 49008, USA
{kun.wang, wuwei.shen}@wmich.edu

Abstract

The gap between UML class models and their implementations impedes program understanding and analysis, and is a source of program errors. Although many reverse engineering techniques were proposed to bridge this gap, two major problems still exist. First, the accuracy of association inference from container classes is not adequate without considering iterators. Second, associations implemented by inherited fields are missed by existing techniques. In this paper, we present an approach to precisely and automatically recover a class model from Java bytecode. Our approach tackles the above problems and improves the accuracy of the recovered models. The preliminary empirical results show that our approach achieved a higher accuracy for association inference than existing reverse engineering tools.

1. Introduction

UML class models describe the static structure of a software system in terms of classes and inter-class relationships. They do so effectively at a high level of abstraction by depicting significant components of a system. Yet, there is a gap between UML models and their implementations. For some projects, a class model may not be available during the system's design phase. Consequently, developers could lose the big picture of their code as the system is being implemented. For others, class models are often inconsistent with their implementations due to two major factors. First, it is common to change either a class model or its implementation without updating the other promptly. As a result, the code is rarely true representation of its model. Second, corresponding constructs for some modeling concepts (e.g., associations) are absent in programming languages. This absence introduces misunderstandings, which result in errors in implementations.

It is thus important to investigate techniques for recovering class models in order to bridge the gap between designs and implementations. Although it is straightforward to detect some inter-class relationships such as generalizations and realizations, the detection of associations presents two major challenges. First, if one-to-many associations are implemented by weakly typed containers (also known as collections), inappropriate associations may be produced. Second, some associations are hard to be detected if they are implemented using inherited fields.

Although a few approaches [4, 5, 11] were proposed to address the first challenge, the accuracy of container type inference is still inadequate. One of the reasons is that these approaches did not consider *iterators*, which provides important information for inferring object types in a collection. Ignoring inherited fields poses another threat to the correctness of the association inference. To our best knowledge, existing techniques (including the concurrent work by Kang et al. [6]) extract associations only from explicitly declared fields of a class. This can miss many associations in some programs. In recently adopted UML 2.0 [9], for example, a large number of associations such as {*subsets*} and {*redefines*} are defined in the context of an inheritance hierarchy. These associations are usually implemented using inherited fields and thus, are missed by existing reverse engineering techniques.

In this paper, we present an approach to precisely and automatically recover a class model from Java bytecode [7]. An important reason for us to work on Java bytecode is that the source code is no longer needed for program analyses or transformations, which can therefore be applied to any applications, including closed source and commercial ones.

The presented approach improves the accuracy of container type inference by considering not only the *getter* and *setter* call sequences on collections, but also the *iteration* call sequences on *iterators*. In addition, our approach infers associations from inherited instance fields as well, which reduces association misses. We have developed a tool called UMLCore to implement our approach, and evaluated it on a suite of third-party programs. The preliminary empirical

31st Annual International Computer Software and Applications Conference(COMPSAC 2007)
0-7695-2870-8/07 $25.00 © 2007 IEEE

results show that our approach can quickly and more precisely infer associations than existing reverse engineering tools.

2. Background

An association at design level specifies a semantic relationship that can occur between typed instances [9]. In this paper, we only consider unidirectional associations since bidirectional associations can be expressed as two unidirectional ones. For a unidirectional association, we often use source class to denote the class at the non-navigable association end, and target class to denote the class connected by the navigable association end. In addition, we often use the notation $X \rightarrow (y)Y$ to denote an association from *X* to *Y*, with the navigable association end *y* connected to class *Y*.

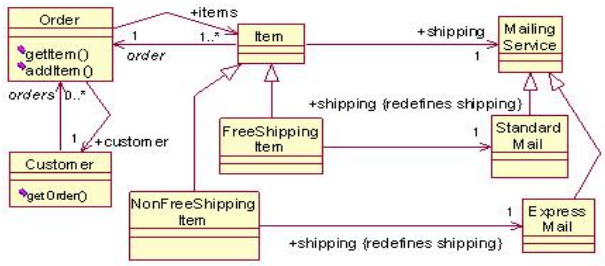

Figure 1. Simplified UML class diagram for an online shopping system

Figure 1 depicts a simplified UML class diagram for an Online Shopping System (OSS). A customer can place an order consisting of a number of items. Each item can be either a free-shipping or a non-free-shipping item. Each item is associated with a mailing service. The {*redefines*} property on each of the specialized associations declares that they be special cases of the general association $Item \rightarrow (shipping)MailingService$. A free-shipping item is shipped with standard mail, and a non-free-shipping item is shipped with express mail.

Typically, a unidirectional association from class X to class Y is implemented by declaring an instance field in class X [8]. A single-valued association (multiplicity 1 or 0..1 at the target end) is normally declared as a simple field whose type is the target class (e.g., Y), while a multi-valued association (multiplicity upper bound greater than 1 at the target end) is declared as an instance field of array or container type [10]. For example, Figure 2 illustrates a possible implementation of the above diagram. The association $Order \rightarrow (customer)Customer$ is declared as an instance field *customer* of type *Customer*, while the association $Order \rightarrow (items)Item$ is declared as an instance field *items* of type *List* in the class *Order*.

```
1. class Order {
2.   Customer customer;
3.   List items = new ArrayList();
4.   Item getItem(String id) {
5.     for (int i = 0; i < items.size(); i++) {
6.       Item item = (Item)items.get(i);
7.       ... }
8.   }
9.   void addItem(Item item) {
10.     items.add(item); }
11. }
12. class Item {
13.   MailingService shipping;
14.   void setShipping(MailingService shipping) {
15.     this.shipping = shipping;}
16. }
17. class Customer {
18.   List orders = new ArrayList();
19.   Order getOrder(String id) {
20.     Iterator ordersIt = orders.iterator();
21.     while (ordersIt.hasNext()) {
22.       Order order = (Order)ordersIt.next();
          ... }
        }
    }
```

Figure 2. A possible implementation of the OSS

A commonly used approach to recovering associations from code is to map each instance field to a unidirectional association, with the field's declaring class being the source class and the field's declaring type being the target class. This approach works well if the field is a simple field. However, if the field is a container, erroneous associations might be resulted. Modern reverse engineering tools such as Rational Rose [1] and REV-SA [3] do not address this problem and produce inconsistent class diagrams.

The difficulty in inferring associations from container fields lies in the fact that containers in the Java Collections Framework [2] are weakly typed. They can store objects of any subtype of *java.lang.Object*, and the specific types of objects stored in a container are not directly known from the field's declaration.

Another commonly used approach to implementing a unidirectional association $X \rightarrow (y)Y$ is to use an inherited field y from *X's* superclass. For example, the association $FreeShippingItem \rightarrow (shipping)StandardMail$ in Figure 1 can be implemented as follows:

```
class FreeShippingItem extends Item {
    StandardMail getShipping() {return (StandardMail)shipping;}
    void setShipping(StandardMail shippingArg) {
      super.setShipping(shippingArg); }
}
```

This association is implemented without declaring any corresponding field in *FreeShippingItem*. Rather, *FreeShippingItem* reuses the inherited field *shipping*, and the semantics of the association redefinition is implemented by its two

0-7695-2870-8/07 $25.00

accessor methods. *setShipping* constrains the type of values that can be assigned to *shipping*; *getShipping* guarantees to return an instance of *StandardMail*. In our empirical studies, we found that a large number of associations are implemented using inherited fields in some programs and they go undetected by existing reverse engineering tools, which examine only the declared fields of a class.

3. Association Inference Mechanisms

In the following, we discuss two primary issues in association inference: 1) how to infer the correct target type of an association implemented by a container, and 2) how to identify most, if not all, fields that represent associations.

3.1. Inferring Object Types of Containers

A container is used to store and retrieve objects of other types. In this context, we define a class as a container if it directly or indirectly implements *java.util.Collection*. The basic idea to infer the contained object types of a container is as follows: tracking the operations that put objects into a container and those that retrieve objects from a container. The types of all these objects would be an approximation of the types held by the container at runtime. The lowest common supertype of the contained types is then computed as the target type of the inferred association.

For the type inference purpose, we classify the methods in the *java.util.Collection* inheritance hierarchy into *getter*, *setter*, and *iteration* methods.

The *getter* methods must have a return type *java.lang.Object*. The contained types can be derived from the types appearing in downcasts that are applied to the returned objects by *getter* methods. For example, *(Item)items.get(i)* at line 6 in Figure 2 indicates that the field *items* contains instances of *Item* because the returned object by the *getter* method is cast to *Item*.

The *setter* methods must have at least one parameter of type *java.lang.Object*. The contained types can be derived from the types of the arguments to the *setter* methods. For instance, *items.add(item)* at line 10 in Figure 2 indicates that field *items* contains instances of *Item* because the argument to the *setter* method is of type *Item*.

An *iteration* method must have a return type *java.lang.Object*. The contained types of a collection can be inferred through its associated *iterator*, by examining classes appearing in downcasts that are applied to the *iteration* methods. From lines 20-22 in Figure 2, for example, we can deduce that the container field *orders* contains instances of *Order* because the element returned by its *iterator* (*ordersIt*) is cast to *Order*.

The UMLCore performs a data-flow analysis on a set of classes and the output is the mappings from collection fields to their contained types. Specifically, the UMLCore looks for evidence showing the contained types of a collection by examining three types of call sequences: *getter*, *setter*, and *iteration*. The abstract syntax of the selected bytecode instructions is given below:

getter-call-seq:
  field-pusher value-pusher method-invocation CHECKCAST t*
setter-call-seq:
  field-pusher value-pusher method-invocation*
iteration-call-seq:
  field-pusher method-invocation ASTORE x [...] ALOAD x
    method-invocation CHECKCAST t
value-pusher:
  field-pusher | *ALOAD x*
field-pusher:
  GETFIELD f | *method-invocation*
method-invocation:
  INVOKEVIRTUAL m | *INVOKESPECIAL m* | *INVOKEINTERFACE m*

In the abstract syntax, terminals written in all uppercase letters represent specific bytecode instructions. For instance, CHECKCAST *t* casts the value on the top of the operand stack to type *t* [7]. The nonterminal *field-pusher* represents a bytecode instruction that loads the value of an instance field (i.e., the callee object) onto the operand stack. The nonterminal *value-pusher** represents a sequence of instructions that push the arguments for a method invocation (represented by the *method-invocation*) onto the operand stack.

Nonterminals *getter-call-seq*, *setter-call-seq*, and *iteration-call-seq* express the abstract syntax of the *getter*, *setter*, and *iteration* call sequences, respectively. The type *t* in the *getter-call-seq* and *iteration-call-seq* represents the type appearing in downcasts that are applied to the returned objects by the getter and iteration methods, respectively. The type of each argument to the *setter* method is obtained by examining the type of the value that was pushed onto the operand stack by the argument's corresponding *value-pusher*.

3.2. Inferring Associations from Inherited Fields

A model recovery technique should consider not only the declared fields of a class, but also all its inherited fields during the analysis. Otherwise, some associations may be undetected. Whether or not an inherited field represents a unidirectional association from the inheriting class depends on the field's declaring type and the field's contained type that can be inferred from the inheriting class.

• Associations from Inherited Simple Fields

Let T be the type of a simple field f declared in C, and C' be a subclass of C. During the analysis of C', UMLCore concludes that f represents an association $C' \rightarrow (f)T'$ in

either of the following two cases: 1) when an instance of C' points to instances of $T_1, ..., T_n$ ($n > 1$), and that T', the lowest supertype of $T_1, ..., T_n$, is a subtype of T; 2) when an instance of C' only points to instances of T' and that T' is a subtype of T.

When class C' is being analyzed, UMLCore seeks the types of objects that can be linked to an instance of C', by checking the types appearing in downcasts applied to the values of f and by tracking the types of values assigned to f. If all these objects are of the same type T', then T' is compared with T to check whether it is a subtype of T; otherwise, the lowest supertype of all these objects is computed and compared with T for the same purpose.

Note that within C', values can be assigned to the inherited field f either directly (if f is a non-private field), or indirectly by calling its setter method. A method m of C is defined as a setter method of field f if m takes a parameter of type T and assigns the parameter to f. For instance, the *setShipping* method of class *Item* is a setter method of the field *shipping*, and is called from the *setShipping* method of class *FreeShippingItem* (i.e., *super.setShipping(shippingArg);*). The abstract syntax of the instruction sequences corresponding to field casts and field assignments is omitted here due to space limit.

Hence, in the class *FreeShippingItem*, UMLCore deduces that there is an association from *FreeShippingItem* to *StandardMail* through its inherited field *shipping*, because the expression *(StandardMail)shipping* and the statement *super.setShipping(shippingArg);* in the class's two methods indicate that *shipping* can only point to objects of type *StandardMail*, which is a subtype of *shipping's* declaring type *MailingService*.

• Associations from Inherited Collection Fields

Let f be a container field declared in C and T be the lowest supertype contained in f and be inferred from C. Moreover, let C' be a subclass of C. During the analysis of C', UMLCore concludes that f represents an association $C' \rightarrow (f)T'$, if it infers that an instance of C' points to instances of T' through f, and that T' is a subclass of T.

When class C' is being analyzed, UMLCore infers the contained types of each inherited field f by applying the type inference algorithms. The lowest supertype T' is then computed from the contained types. To determine whether an association $C' \rightarrow (f)T'$ should be extracted, UMLCore must check whether T' is a subclass of T. However, type T may not be available when C' is being analyzed, because UMLCore analyzes each class in a random order. In this case, UMLCore records an inference dependency. That is, the truthfulness of the association $C' \rightarrow (f)T'$ depends on $f's$ contained types inferred from C. Later on after T is determined by analyzing class C, UMLCore either confirms or revokes the association $C' \rightarrow (f)T'$, depending on whether or not T' is a subclass of T.

4. Conclusions and Future Work

A class model recovered from its implementation can help engineers understand the system at a high level of abstraction and detect implementation or design flaws. In this paper, we proposed an approach to precisely and automatically recover a UML class model from Java bytecode. Compared with existing techniques, our approach analyzes more language constructs and improves the accuracy of association inference. The preliminary empirical results showed that our approach can quickly produce accurate results in many programs that exist in practice. We plan to work on the solutions to detecting associations from some special implementations such as those implementing the $\{union\}$ semantics, which is newly adopted in the UML2.0 [9]. We also plan to develop techniques to better handle user-defined containers that do not implement *java.util.Collection*.

References

[1] IBM Rational Rose. http://www-306.ibm.com/software/rational/.

[2] J. Bloch. Java Collections Framework. http://java.sun.com/docs/books/tutorial/collections/.

[3] D. Cooper, B. Khoo, B. R. von Konsky, and M. Robey. Java Implementation Verification Using Reverse Engineering. In *Australasian Conference on Computer Science*, pages 203–211, 2004.

[4] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314, 2004.

[5] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. In *International Conference on Software Engineering*, pages 194–202, 1999.

[6] Y. Kang, C. Park, and C. Wu. Reverse-engineering 1-n Associations From Java Bytecode using Alias Analysis. In *Information and Software Technology, volume 49, issue 2*, pages 81–98, 2007.

[7] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, Second Edition. http://java.sun.com/docs/books/vmspec/.

[8] A. Milanova. Precise Identification of Composition Relationships for UML Class Diagrams. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 76–85, 2005.

[9] OMG. Unified Modeling Language: Superstructure. Version 2.0, formal/05-07-04. http://www.omg.org/.

[10] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Section Edition*. Addison-Wesley Professional, 2004.

[11] P. Tonella and A. Potrich. Reverse Engineering of the UML Class Diagram from C++ Code in Presence of Weakly Typed Containers. In *IEEE International Conference on Software Maintenance*, pages 376–385, 2001.

0-7695-2870-8/07 $25.00 © 2007 IEEE