Reverse Engineering of UML Specifications from Java Programs

Martin Keschenau
Chair for Computer Science II
Programming Languages and Program Analysis
RWTH Aachen University
markes@i2.informatik.rwth-aachen.de

ABSTRACT

In the present work, we outline a reverse engineering approach for UML specifications in form of class diagrams from Java bytecode. After a brief introduction to the subject we present some analyses which go beyond mere enumeration of methods and fields. A glance onto some related work shows that there seems to be no pat solution for the reverse engineering of the more difficult class diagram elements.

We sketch our method of determining association multiplicities, being, in a sense, representative of our approach in general: "intuitive" analyses, producing results that can be understood by a programmer when inspecting the source code of a given class.

Finally, we introduce a tool that implements this work and we apply it onto a small real life example, discussing the results it gave.

Categories & Subject Descriptors

D.2.7. Distribution, Maintenance and Enhancement

General Terms

Documentation, Design, Languages

Keywords

UML, Class Diagrams, Reverse Engineering, Java, Bytecode

1. UML CLASS DIAGRAMS

In the domain of software engineering, UML [1] has grown to a widely known and readily used graphical standard for representing various aspects of an object oriented software system, and its class diagrams have been proven to be particularly useful. They are the kind of specifications that are to be reverse engineered in this work, with the goal of e.g. comparing an actual implementation against an existing design document, or, of course, design recovery.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

2. DIFFICULTIES IN CLASS DIAGRAMS

Examination of the features of some well-known tools ([2], [3]) has shown that it is difficult to reverse engineer class diagram elements beyond the class boxes themselves or the simpler relationships. Additionally, [4] succeeds in listing different kinds of dependencies. [5] however seems rather powerful, offering detection of associations, static recognition of multiplicities, aggregations and association classes - a function scope similar to what we present in here - but the corresponding tool could not be tested due to a currently ongoing revision of the program. It is a source code based approach though, having the disadvantage that the sources must be present in order to perform the analysis.

In this work, we will focus on analysing some of those more difficult notions of UML class diagrams, namely

- association multiplicities,
- compositions, which are particular, stronger versions of associations and
- query methods: methods that do not alter the system state.

We use *intuitive analyses*, in a sense that we deliberately renounce on existing theories in order to present algorithms and results a programmer might understand and be able to verify in the code.

3. OUTLINE: MULTIPLICITY ANALYSIS

Let us outline the approach for the associations multiplicities:

1. Prior to the analysis, we have to interpret the notion of UML association and its multiplicity in a precise way:

   An *association* f as shown in fig. 1 represents a field

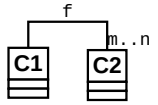

Figure 1: Association

   f in the definition of C_1, defined either as `C2 f;` or as an array, `C2 f[];` . The C_2-end multiplicity of $m..n$ is interpreted as the number of C_2 objects f can reference during the lifetime of a C_1 object.

2. We then make a key observation, which will be the underlying idea for the multiplicity analysis:

   Every assignment to f, $f = X$, implies a *local multiplicity*. E.g., the intuition for $f = new\ C_2$ would be a local multiplicity of 1..1.

3. Finally, we realise that generally, there is another aspect to consider with this approach:

   Local cardinalities depend on where they are encountered. A constructor, called at most once, differs fundamentally from other methods which can be called an arbitrary number of times. Thus, local cardinalities have to be "fine tuned" - in our example, $f = new\ C_2$ would imply a local multiplicity of $0..*$ if encountered in an ordinary method, since we do not know how often that method might be called.

To sum up, the multiplicity analysis for a field f collects all assignments to this field. Thus, it is a code pattern based approach. On every match, we identify a local cardinality which will be added up to the previously established result in an appropriate way.
Note that point 3. illustrates what we call "intuitivity of analyses": Even if it might be possible to approximate how often a method could be called using data flow analysis, we choose deliberately not to. We believe that $0..*$ is less confusing than say "0..42", where it is not straight-forward to see where these numbers arise from.

4. TOOL

As a realisation of those concepts, a tool called `class2uml` [6] has been developed. It implements the aforementioned analyses for Java bytecode offering the following features:

- Recursive construction of a class diagram starting on one given class (limitation of recursion depth and/or limitations in package prefixes are possible).
- Generation of class/interface boxes (including query methods), generalisations, dependencies, associations with their multiplicities and finally, compositions.

An advantage of the bytecode approach is the possibility of analysing classes of which the sources are not available.

5. EXAMPLE: JAVA.LANG.BOOLEAN

Let us discuss a sample diagram produced by `class2uml`. It is the diagram constructed for `java.lang.Boolean` of Sun's JDK 1.4, manually adjusted in order to fit the page layout.

The three associations `TRUE`, `FALSE` and `TYPE` have been constructed from the following field declarations:

```
public static final Boolean TRUE = new Boolean(true);
public static final Boolean FALSE = new Boolean(false);
public static final Class TYPE =
  Class.getPrimitiveClass("boolean");
```

These fields are declared *final*, thus, we can be certain that those assignments are the only assignments to these fields, even without knowing the rest of the `Boolean` code.

`TRUE` and `FALSE` obtained the multiplicities 1..1. This is correct, since they are new-initialised and these are their only assignments ever.

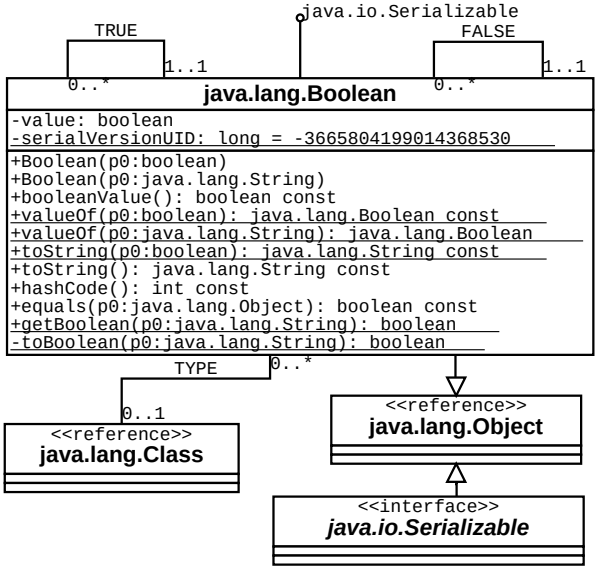

Figure 2: Constructed diagram of java.lang.Boolean

On the other hand, `TYPE` obtained 0..1. It is initialised from a method call and we cannot know whether that call will return a reference or `null`, hence 0..1 is correct.

Note also that query methods, here shown as `const`, have been detected in no fewer than 6 cases. Unfortunately, we cannot elaborate their analysis any further in this paper.

The example above might suggest that our approach to multiplicity analysis is a simplistic one, yet it is not. Think e.g. of a field f of an object array type; then, there are assignments to *the contents* of f to be considered, too: $Y[\ldots] = X$ where Y is a piece of code that produces a reference stored in f, not necessarily $Y \equiv f$, as propagation of variables may occur. This requires an entire subanalysis and is only one of the hidden challenges in the analyses listed in section 2.

6. CONCLUSION

This work has shown a tool for automatic generation of UML class diagrams from Java bytecode. The key issue has been to detect those diagram elements which require some efforts in analysis: association multiplicities, compositions and query methods. We have demonstrated an intuitive, code pattern based solution, approaching the analysis of association multiplicities from a programmer's point of view.

7. REFERENCES

[1] OMG: *UML Specification Version 1.4*, `http://www.omg.org/cgi-bin/doc?formal/01-09-67`

[2] *FUJABA*, `http://www.fujaba.de/`

[3] *ArgoUML*, `http://argouml.tigris.org/`

[4] Jan Pechanec: *Reverse Engineering of Programs in Java Language*, Master Thesis, 2002, `http://www.sweb.cz/pechanec/Java2UML/`

[5] Martin Gogolla, Ralf Kollmann: *Re-Documentation of Java with UML Class Diagrams*, `http://www.db.informatik.uni-bremen.de/publications/Gogolla_2000_euroREF.ps`

[6] *class2uml*, `http://www-i2.informatik.rwth-aachen.de/~markes/class2uml/`