Abstract
A program’s object model captures the essence of its design. For some programs, no object model was developed during design; for others, an object model exists but may be out-of-sync with the code.
This paper describes a tool that automatically extracts an object model from the classfiles of a Java program. Although the tool performs only a simple, heuristic analysis that is almost entirely local, the resulting object model is surprisingly accurate. The paper explains the form of the object model, the assumptions upon which the analysis is based, and its limitations, and evaluates the tool on a suite of sample programs.

1 Introduction
Womble is a tool that extracts object models from Java bytecode. In this paper, we describe its motivation, design and application to a suite of programs.
In short, Womble processes a Java program presented as a collection of class files. It applies a simple, lightweight analysis to the code to produce an object model: a graph whose nodes are Java classes, and whose edges represent either subclassing or associations. The object model may be visualized using Blob, a graph layout tool we have developed, or using Bell Labs’s Dot, for which Womble generates an appropriate input file. Womble can also generate module dependence diagrams.
A crude object model can be built trivially from the declarations of fields. From the code fragment

```java
class Company {  
  Person boss;  
...  
```

for example, an association edge labelled boss from Company to Person is obtained. Multiplicity is inferred from the presence or absence of arrays; for the declaration

```java
Person employees[];
```

an edge labelled employees would be added, with a marking at the Person end showing a zero or more multiplicity.

This approach has a crucial flaw. Object models should not expose the representations of objects. The associations of an object are an abstract property that could – in theory at least – be inferred from a behavioural specification of the object’s methods. At the very least, the object model should hide how a particular association is implemented. In this scheme, however, a one-to-many association is shown only for fields represented as arrays. Suppose the Company class were to contain a table that maps names to employees:

```java
Company hashtable;  
```

This would result in an edge from Company to Hashtable, both inappropriately exposing the representation of the employees field, and failing to show an association between Company and Person. Womble does not suffer from this problem, and would correctly show an edge from Company to Person labelled employees.

This paper explains a new approach to object model extraction whose key features are as follows:

1. Associations are inferred by examining the methods of a class in addition to its declared fields. How a field is used provides more useful clues to its role than its declaration alone. Multiplicity and mutability of associations can both be determined.
2. Container classes, such as Hashtable, are handled correctly most of the time: the container class does not appear as a node, but results in appropriate associations.
3. Almost all the analysis is performed locally, classfile by classfile. It is therefore insensitive to missing files, and usually scales linearly with the size of the system (but see non-local effects in section 7).

We have been shocked by how useful Womble is. Even the inventors of static analysis tools do not use them all the time on their own code. But we have found ourselves running Womble frequently on our Java programs – including Womble itself. Furthermore, the developers of the programs we used to evaluate Womble with have asked us to produce updated models as they have modified their code.

Womble’s utility seems to be due to two factors. First, it places practically no burden on the user at all. It works on bytecode, which for the purpose of our analysis, contains all...
the information available from the source code, but is more compact and more readily available. The user simply points the tool to some directories containing classfiles, and it generates a picture. Womble is tolerant of omissions; as more classfiles are presented, it adjusts the picture, usually just by extending it.

Second, object models seem to be a natural and expressive representation of programs. They are generally much smaller than dependence diagrams or call graphs; the picture resulting from a program of 20,000 lines can usually be viewed without any filtering or reorganization.

Our paper begins with an explanation of what object models are and why they are useful. It then gives the syntax and (informal) semantics of our modelling notation, which is roughly a subset of UML [Rat97]. The extraction mechanism is explained, followed by a discussion of its application to a suite of small and medium-sized programs. The paper closes with a review of the main challenges of extracting object models, and a discussion of related work.

2 Why Extract Object Models?

An object model is a representation of the abstract state of a program. It takes the form of a graph whose nodes represent sets of objects, and whose edges represent either subset relationships or associations between objects.

At the same time, object-oriented programs often exhibit a code structure that corresponds closely to an object model, with subclassing implementing subset relationships and fields implementing associations. An object model is an architectural view of a program, showing its essential components and how they interact.

Traditional design representations, such as dependency diagrams and call graphs, are based more on the control structure of the code. Since software architectures are increasingly dynamic, with components being assembled at runtime, control structure is becoming less significant. An object model, in contrast, naturally accommodates this dynamism, by fixing only the sets of objects that exist and their potential relationships, and not objects and links individually.

Object models play a useful role at several stages in development. In requirements, object models can capture the structure of the problem domain; indeed, such ‘semantic data models’ were their original motivation. In specification, an object model can describe a program as an abstract state machine; Z specifications [Spi92] in which binary relations dominate are essentially object models, albeit in a more elaborate syntax. In design, object models can be used to articulate overall structure [R+91], to document patterns [G+95] and to guide implementation. In maintenance, an object model is invaluable; it is hard to make any progress fixing or reworking a program without an understanding of what objects exist and how they are related to one another.

Legacy code is, of course, unlikely to be documented with an object model. But this is not the only scenario for which automatic extraction of a model from code is useful.

The relationship between an object model of a program and its code is not always straightforward.

Our notion of object model is to some degree at variance with the prevailing one. In the last few years, there has been a trend away from models that succinctly summarize the essential state of a program towards models that mirror the code structure closely, and become repositories for all kinds of programming language specific details. To see this, just compare UML [Rat97] to its predecessor OMT [R+91].

In our view, the value of object modelling in specification and design is precisely in describing abstract features that are not so readily expressed in code. The same argument, of course, has been made many times for declarative specification in general. A number of recent object-oriented methods take a similar position: see, for example, Fusion [C+93], Syntropy [CD94] and Catalysis [DW98].

In this context, the role of a tool is not to recover an object model from design time, but rather to extract an object model of the code that can be compared to the design model. This comparison might be automated given a user-supplied abstraction, in the style of Aspect’s abstraction function [Jac95] or a reflexion map [MNS95].

3 Object Model Syntax and Semantics

There are many varieties of object model syntax. Our syntax includes only what we judged to be the essential features of an object model, and is roughly a subset of UML [Rat97].

An object model is a graph whose nodes represent sets of objects: for the models Womble constructs, these will correspond to Java classes or interfaces. There are two kinds of edges between nodes. An arrow labelled \( r \) from a node labelled \( A \) to a node labelled \( B \) represents a relation named \( r \) consisting of pairs whose first elements belong to \( A \) and whose second elements belong to \( B \). An unlabelled arrow with a large triangular arrow head from \( A \) to \( B \) asserts that the set of objects denoted by \( A \) is a subset of the set denoted by \( B \); for the models Womble constructs, these arrows always correspond to ‘extends’ or ‘implements’.

Relation arrows may be annotated further in two ways. First, markings may be added to show multiplicity: \( * \), \( ? \) and \( ! \) denote, respectively, zero or more, zero or one, and exactly one. The absence of a marking is equivalent to \( * \). So, for example, an edge from \( A \) to \( B \) marked only with a question mark at the \( B \) end indicates a relation that associates zero or one \( B \) with each \( A \) — that is, a partial function.

Second, the ends of the arrow may be marked to show mutability. A small hatch through the \( B \) end of the arrow marks \( B \) as ‘static’ in the relation: it says that the set of \( B \)'s associated with a given \( A \) cannot change during its lifetime. The tail end of the arrow may also be marked as static, but Womble only infers immutability at the head.

Figure 1 shows the syntax of the modelling notation given as an object model itself. Fortuitously, it illustrates all the features of the notation. To include mutability, we have assumed that the model describes the state of a diagram.
being constructed in a drawing tool, and added mutability markings accordingly. They show that nodes and edges can be added, and that the markings associated with a relation end can be changed, but that no other changes may be made. The hatch on the Node end of the at relation, for example, says that the node with which a relation end is associated may not be changed (but an edge with its ends may be deleted and replaced by another).

(Given that UML is becoming a widespread standard, some justification for our departure is needed. In short, UML diagrams are more cluttered: they are harder to draw, by hand and by machine, and harder to read. Mutability hatches would be rendered instead by the annotation ‘frozen’; relation edges would not be directional, the arrowhead instead being written as a symbol next to the relation name; the multiplicity markings would be range expressions such as ‘0..1’ for ‘?’. For a commercial tool, however, the advantage of using a standard notation might outweigh any of its disadvantages.)

4 Extraction Mechanism

During the analysis of a program, Womble maintains a database. Object models and dependence graphs are constructed from this database. The user can choose to add classfiles at any point; Womble parses and analyzes them, and includes them in the database. The analysis is mostly monotonic: adding a classfile tends to add a class with its edges, although for reasons that will become clear later, the extra information may cause annotations on existing edges to change.

Womble makes use of most of the information in the classfile. In particular, it uses the name of the class and the names of classes it extends or implements; the declarations of fields, static and virtual; the signatures of methods; and the bytecode instructions within methods. Order of instructions is significant.

The basic structure of the object model is extracted as follows. A node is introduced for each class, with a subset edge for each extends or implements relationship. And, roughly speaking, for each field declaration in a class $A$ of a field $p$ of class $B$, an association labelled $p$ from $A$ to $B$ is added.

4.1 Inferring Associations

The handling of associations is complicated by two factors. First, many fields represent attributes of objects that are not of interest at the level of the object model, and should be suppressed. To filter out bogus associations, Womble ignores fields of primitive type. It also gives the user the option of ignoring fields whose type is a Java class in the standard Java library (eg, the classes in the java.util package).

Second, some fields contain references to classes that are containers of classes of interest. We would like to omit the container class from the object model and link directly to the contained object class. If a field called employees in the class Company is a vector or a hashtable of Person objects, for example, we would like to show an association called employees from Company to Person. This inference is more complicated than it might seem, since container classes may also be user-defined.

Womble uses the following criterion. A class is a container if it is truly polymorphic: this means that its code only explicitly references primitive types. A class is deemed to be a container if none of its methods have an argument or result of a non-primitive type, and at least one method has an argument or result of class Object.

The methods considered in this analysis are those that appear in the class whose associations are being determined, so the code of the candidate container class is not needed. If that class is provided, however, the analysis is performed instead on its methods. This is an example of the non-monotonicity alluded to above: Womble might decide that a class represents a container, but revoke the decision when more information is available.

What constitutes a primitive type in this analysis is problematic. Clearly int is primitive, and it makes sense to
treat types from \texttt{java.lang} as primitive too. This works most of the time, but is not foolproof: a container with a method that serializes its contents to a \texttt{Stream} (from \texttt{java.io}) would not be recognized as a container.

So far we have described only how Womble determines that a class represents a container. It is still necessary to find the class at the other end of the association. In a language with parametric polymorphism, this would be trivial; a field declared as having type \texttt{vector[Person]} would give an association to the class \texttt{Person}. But Java has only subtype polymorphism, and the type of object stored in a container is not evident from the declaration.

Womble therefore does an elementary dataflow analysis. It identifies (1) classes appearing in downcasts that are applied to objects returned by container methods, and (2) classes passed as arguments to container methods. These classes are taken to be the classes of the objects held within the container. This analysis is not sound: some downcasts might be expected always to fail, for example. But we have yet to see it produce incorrect results in practice.

4.2 Inferring Multiplicity

Multiplicity and mutability annotations are obtained by analyzing how fields are used within methods. Let’s start with the annotations for the head of an association arrow, which are easier to determine.

A multiplicity of \texttt{zero or more} conveys no information. It is therefore the default, and Womble looks for evidence of a tighter constraint. If the field is not an array, and does not reference a container class, the multiplicity is \texttt{zero/one} or \texttt{exactly one}. If the field is ever set to \texttt{null}, or if the class has a non-private constructor (or any method called by one) in which the field is not assigned a value, \texttt{zero/one} multiplicity is chosen. Otherwise, the field always has a value so a multiplicity of \texttt{exactly one} is chosen.

The multiplicity on the tail of the association arrow is likewise set to \texttt{zero or more} as a default. Recall that a multiplicity of \texttt{zero/one} on an association marked \texttt{p} from \texttt{A} to \texttt{B} means that \texttt{p} associates each \texttt{B} with at most one \texttt{A}; if \texttt{p} is a field, this will be true only when different objects of the class \texttt{A} do not share objects of the class \texttt{B} through \texttt{p}. A sound inference would therefore require alias analysis.

Womble uses a cheap heuristic. We have observed that on many occasions in which there is no such sharing, the \texttt{B} object is created within the \texttt{A} object and not passed around. We therefore set the multiplicity to \texttt{zero/one} when the \texttt{B} object is created in \texttt{A}, when the field is private, and when no method in \texttt{A} returns an object of class \texttt{B}. No attempt is made to identify a multiplicity of \texttt{exactly one}.

4.3 Inferring Mutability

Womble only attempts to adorn the heads of arrows with mutability markings. That is, in analyzing an association from \texttt{A} to \texttt{B}, it tries to show that the code of \texttt{A} does not change the set of \texttt{B} objects associated with an \texttt{A}, but does not consider how the \texttt{A} objects associated with a \texttt{B} might change. Like the multiplicity constraint at the tail of an arrow, this requires an alias analysis in general.

A field is regarded as mutable if any method of the class that is not a constructor (or any method of another class) assigns a value to it. Otherwise, it is regarded as immutable and a marking is added to the corresponding association in the model. A container is regarded as mutable if it has a method that takes arguments of class \texttt{Object} and returns \texttt{void} or any primitive type.

This information may be revised when other classes are analyzed. If the field is non-private, it may be inferred to be immutable. A later discovery of an assignment to the field in another class will cause this inference to be revoked.

5 Using Extracted Object Models

Extracted object models can be used for many purposes. In this section, we give some examples of how we have used models extracted by Womble in the course of our work in the last few months.

Sketching gross structure. When encountering a program for the first time, it is useful to have a sketch that shows the essential elements of its structure. An object model often fits the bill; it gives a much smaller graph than a module dependence diagram, and its edges are more informative. For example, we wanted to assess Grappa, a graph layout tool developed at Bell Labs, to determine if it would be suitable for laying out Womble’s models; a quick examination of the object model revealed that, contrary to our expectation, Grappa does not include a graph layout algorithm. We have used Womble in the development of Fox, an object constraint solver: before integrating new packages constructed by a student, we used Womble to show their structure.

Comparing to design. We designed our own drawing tool, Blob, in our modelling notation. When the tool had been coded, we used Womble to extract the object model and compare it to the model of the design.

Identifying design patterns. Design patterns that underlie the structure of the code often reveal themselves in the object model. Figure 2 shows part of the object model Womble generated from Grappa, a graph layout tool developed at Bell Labs. A pattern aficionado will recognize this as an instance of the \texttt{Bridge} pattern [G+95]. The name of the superclass suggests an additional pattern at play.

Identifying data structures. An object model may reveal essential data structures. The model Womble extracted from Nitpick, a specification analyzer, for example, included the fragment shown in Figure 3, which is easily seen to be a list of lists. This structure, it turns out, corresponds to the representation of a boolean formula in disjunctive normal form. (The classes shown were not identified as containers because they were not polymorphic, and contained additional fields tailored to boolean formula manipulation.)
Looking for design anomalies. A careful examination of the object model of a program can reveal anomalies in its design. Poring over a model of Haystack (an information retrieval program) with one of its designers, we found an apparently useless redundancy: the program associated objects with unique identifiers generated for persistent storage in several places, when a single table would have sufficed.

Looking for bugs. By correlating the details of the object model with the designer’s expectations, it is sometimes possible to find bugs in the code. We found a bug in Fox, for example, by noticing that an association that we had expected to have a multiplicity at its head of exactly one instead had a multiplicity of zero or one. A constructor failed to initialize the relevant field.

6 Experimental Results
To evaluate the accuracy of the models extracted by Womble, we applied it to a suite of small programs: Blob, a graph drawing tool that we built as a front-end for Womble; Fox, an object constraint solver; Fusion, an environment for teaching graphics programming; Grappa, a graph drawing toolkit; Haystack, an information retrieval system; Rivet, an open Java virtual machine for dynamic analysis; and Womble itself, the smallest program in the collection. All of these were developed at MIT, with the exception of Grappa, which is from Bell Labs.

Some statistics about the sizes of these programs are given in Table 1. Note how many more dependence edges there are than association edges: this factor alone makes object models more useful. (Subset edges occur in both object models and module dependency diagrams, and are not counted as dependences.)

To assess the quality of the object models produced, we first looked at how frequently Womble was able to mark an association as static, or to annotate an association with a more informative multiplicity than zero or more. Table 2 shows the results of this analysis. It can be seen that there are few cases in which Womble can infer that a field is never null (#1, head); a perusal of the code confirms that this is not an artifact of the analysis.

We then conducted two analyses to determine the accuracy of the generated object model. First we examined how well Womble identified container classes. Table 3 shows, for each program, how many nodes there are in the constructed model (#nodes), how many nodes have been eliminated and absorbed into edges (#containers found), how many of these corresponded to user-defined containers (#user-def found), and how many nodes should have been eliminated but were not (#containers missed). The final column shows how many associations with a multiplicity of zero or more arose from fields declared as arrays.

This table shows that Womble’s container inference strategy works well; the failure cases usually correspond to nesting of containers. It also shows that arrays are rarely used – with the exception of Rivet, which uses arrays extensively for performance reasons – so that a tool that generates zero or more associations for arrays alone will give poor results.
<table>
<thead>
<tr>
<th>program</th>
<th>#classfiles</th>
<th>#methods</th>
<th>#instrs</th>
<th>#nodes</th>
<th>#assocs</th>
<th>#subsets</th>
<th>#deps</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
<td>54</td>
<td>361</td>
<td>29k</td>
<td>29</td>
<td>59</td>
<td>79</td>
<td>97</td>
<td>9s</td>
</tr>
<tr>
<td>Fox</td>
<td>123</td>
<td>668</td>
<td>60k</td>
<td>68</td>
<td>90</td>
<td>62</td>
<td>530</td>
<td>15s</td>
</tr>
<tr>
<td>Fusion</td>
<td>569</td>
<td>3843</td>
<td>313k</td>
<td>317</td>
<td>447</td>
<td>483</td>
<td>1233</td>
<td>131s</td>
</tr>
<tr>
<td>Grappa</td>
<td>84</td>
<td>704</td>
<td>46k</td>
<td>37</td>
<td>93</td>
<td>63</td>
<td>246</td>
<td>15s</td>
</tr>
<tr>
<td>Haystack</td>
<td>120</td>
<td>744</td>
<td>155k</td>
<td>59</td>
<td>87</td>
<td>73</td>
<td>371</td>
<td>11s</td>
</tr>
<tr>
<td>Rivet</td>
<td>102</td>
<td>1019</td>
<td>66k</td>
<td>49</td>
<td>78</td>
<td>59</td>
<td>474</td>
<td>31s</td>
</tr>
<tr>
<td>Womble</td>
<td>30</td>
<td>201</td>
<td>20k</td>
<td>18</td>
<td>19</td>
<td>42</td>
<td>69</td>
<td>5s</td>
</tr>
</tbody>
</table>

Table 1: Basic statistics for the suite of programs. #instrs is an estimate of the number of bytecode instructions in the program; #nodes, #assocs and #subsets are the numbers of nodes, association edges and subset edges in the object model. #deps is the number of dependence edges in the dependence diagram: one class depends on another if it calls a method of that class or uses an object of that class.

<table>
<thead>
<tr>
<th>program</th>
<th>#assocs</th>
<th>#static</th>
<th>#0/1, head</th>
<th>#1, head</th>
<th>#≥0, head</th>
<th>#0/1, tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
<td>59</td>
<td>38</td>
<td>29</td>
<td>1</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>Fox</td>
<td>90</td>
<td>76</td>
<td>71</td>
<td>2</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>Fusion</td>
<td>447</td>
<td>410</td>
<td>400</td>
<td>6</td>
<td>41</td>
<td>111</td>
</tr>
<tr>
<td>Grappa</td>
<td>93</td>
<td>79</td>
<td>72</td>
<td>1</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>Haystack</td>
<td>87</td>
<td>69</td>
<td>61</td>
<td>0</td>
<td>26</td>
<td>47</td>
</tr>
<tr>
<td>Rivet</td>
<td>78</td>
<td>71</td>
<td>54</td>
<td>0</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>Womble</td>
<td>41</td>
<td>34</td>
<td>25</td>
<td>1</td>
<td>15</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Measures of Womble’s ability to infer multiplicity and immutability. #assocs is the number of association edges. #static is the number of association edges which received a static marking. The remaining columns give the number of cases in which the end of an association was marked with a particular multiplicity: ‘#1, head’, for example, is the number of heads of association arrows with ‘exactly one’ multiplicity.

<table>
<thead>
<tr>
<th>program</th>
<th>#nodes</th>
<th>#containers found</th>
<th>#user-def found</th>
<th>#containers missed</th>
<th>#array fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
<td>29</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fox</td>
<td>68</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Fusion</td>
<td>3</td>
<td>17</td>
<td>36</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>Grappa</td>
<td>37</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Haystack</td>
<td>59</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Rivet</td>
<td>49</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Womble</td>
<td>18</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Accuracy of Womble’s inference of container classes. #nodes is the number of nodes in the constructed model; #containers found is the number of classes identified as containers and thus eliminated as nodes; #user-def found is the number of user-defined classes identified as containers; #containers missed is the number of container classes that were not identified correctly, and appeared erroneously as nodes. #array fields is the number of associations with a multiplicity of zero or more arising from fields declared as arrays.

<table>
<thead>
<tr>
<th>program</th>
<th>#nodes</th>
<th>#edges</th>
<th>#mutability confirmed</th>
<th>#mutability revealed</th>
<th>#mutability wrong</th>
<th>#mult confirmed</th>
<th>#mult revealed</th>
<th>#mult wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox</td>
<td>68</td>
<td>86</td>
<td>79</td>
<td>6</td>
<td>1</td>
<td>79</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Fusion</td>
<td>12</td>
<td>28</td>
<td>24</td>
<td>2</td>
<td>2</td>
<td>25</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Haystack</td>
<td>7</td>
<td>19</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: A comparison of association annotations inferred by Womble and provided by a programmer. #edges and #nodes refer to the fragment of the object model considered. The 3 columns labelled ‘mutability’ count instances of agreement or disagreement on the marking of association heads with mutability annotations; the columns labelled ‘mult’ refer to the marking of association heads with multiplicity annotations. In ‘confirmed’ cases, Womble and the programmer agreed; in ‘revealed’ cases, Womble was right and the programmer wrong; in ‘wrong’ cases, the programmer was right and Womble wrong.
Second, and finally, we attempted to evaluate the accuracy of the annotations on association edges. Because we did not have tools available against which to benchmark Womble, we were led to a less scientific (but more entertaining) approach.

For 3 of the sample programs, we asked a programmer who had been involved in the program's construction to identify a part of the program he or she was familiar with. We then presented the programmer with the relevant portion of the object model, with multiplicities and mutability markings erased. We then asked the programmer to complete the diagram by hand (with access to the code permitted). The annotations provided by the programmer and by Womble were then compared.

Table 4 shows the results. The first two columns give the size of the object model fragment examined. The remaining columns count instances of agreement or disagreement between programmer and tool. 'Confirmed' means that both agreed; 'revealed' means that Womble produced a different result, which on further examination of the code was shown to be correct; 'wrong' means that Womble produced a different result which was found to be wrong. '#mutability confirmed', for example, gives the number of cases in which Womble and the programmer agreed on whether an association end should be labelled static or not; '#mult wrong' gives the number of times Womble produced an incorrect multiplicity annotation.

Of course, both tool and programmer might be wrong for the cases labelled 'confirmed'. But the outcome of the disagreements is encouraging: Womble won most of the time.

A sample of the Womble's actual output is shown in Figure 4. Mutability and multiplicity annotations appear in the edge labels as prefixes and postfixes due to limitations of our drawing tool. Subset edges are shown as dotted lines for the same reason.

The diagram is a part of the object model of Java's Abstract Window Toolkit (AWT) package illustrating the structure of menus. A number of useful observations can be made. Since a Menu contains MenuComponents, and Menu is itself a MenuComponent, it is apparent that menus can be nested. It can also be seen that items in menus can optionally have keyboard shortcuts; that only a Frame can have a menu bar, but that any Component can have popup menus in it. The presence of a helpMenu field suggest that the help menu has a special status.

7 Challenges & Assessment

In this section, we briefly review the challenges of extracting an object model from code and assess Womble’s performance.

Containers. The most basic challenge is to determine which classes should appear at all; polymorphic collections (tables, lists, vectors, etc) should be omitted, whether defined in a library or user-defined. Womble’s simple strategy works much of the time, but fails for nested containers (for example, a table of tables). Womble cannot distinguish the key and value classes in a table, so it is unable to produce ‘qualified associations’ [R+91].

Associations. The challenge in placing associations is to determine their target. In a language like Java lacking parametric polymorphism, the class of objects contained in a homogeneous container is not explicit in the code but must be inferred. Womble’s strategy of examining downcasts in local methods seems to work reliably.

Mutability. Determining whether an association end is static appears at first to be easy, and Womble’s strategy (marking an association as static when no mutating operations, or field assignments by other classes, are found) seems to be sound. An association due to a container, however, presents problems. First, the container class itself must be analyzed to decide whether the container is mutable. Second, the field might be aliased, and a container method executed in another class might cause the contents of the container to change. Womble ignores both of these problems.

Multiplicity. Choosing the multiplicity of the head of an association arrow amounts to determining whether a field can have a null value. Womble’s crude strategy – assuming a field can be null if it is not assigned in a constructor or is later assigned null – works fine, even though, being flow insensitive, it may produce inaccurate results when there is dead code. Choosing the multiplicity of the tail of an association arrow amounts to performing an alias analysis to see whether referenced objects are shared. Womble only scratches the surface here.
Non-local effects. When the code of a subclass is analyzed, it may be necessary to revise the portion of the object model constructed for its superclass. An association of the superclass may, for example, have been marked as static because the relevant field is never assigned to outside a constructor. A subclass might violate this; it might, in other words, not behave like a subtype. Womble therefore re-evaluates the properties of a superclass when a subclass is analyzed. A similar issue arises with non-private fields assigned to outside a class, and with container inference.

Abstraction. Containers aside, Womble does not attempt to infer abstractions. A class with a boolean instance field might be representing two distinct sets of objects; a primitives-typed field might in fact be an abstract object from a design viewpoint. It seems implausible that such abstractions could be performed without direction from the user. In a sense there is no single object model, but rather a collection of models at different levels of abstraction. The most abstract model is not always what is wanted; a less abstract model gives useful information about how state is represented.

Other features. We have discussed only the essential components of the object model. An elaborate language such as UML has a huge number of features that might be inferred. Some, such as the names and signatures of methods, are mundane and in our opinion not of much use. Others would require special analysis. Aggregation, for example, if interpreted as a lifetime dependency, might be inferred in a manner similar to mutability.

8 Related Work

There is a large body of work on reverse engineering of code, so we shall confine our comments only to closely related work.

A recent paper [SvG98] shows how patterns can be inferred from Java code. First a database of simple relations is extracted (class contains method, class contains field of class, method calls method, etc). An object model is then constructed from the database: an association edge, for example, is added between two classes whenever the first contains both a call to a method of the second and additionally contains a field of that class. Like ours, this technique is lightweight and heuristic. The object models are less detailed; because the instructions executed within methods are not considered, association edges cannot be annotated with multiplicity and mutability markings. There is also no inference of containers.

Most commercial CASE tools provide some reverse engineering of object models from code: Rational's Rose, CayenneSoft's ObjectTeam, and ASTI's GDPro, for example. These tools take a very concrete approach. Fields are translated directly into associations, for example, and there is no inference of multiplicity, mutability or containers.

Object models can also be constructed, albeit less conveniently, by manually processing the results of more traditional analyses. Tools such as Imagix and Sniff [Bi92], for example, can generate reports about call relationships and usage of data components. Many reverse engineering tools developed by researchers (such as [CS90, LC94]) use the same underlying information as these commercial tools, but process the results in a more sophisticated fashion (eg, by computing clusters of modules to suggest reorganization).

Our approach was partly inspired by Murphy and Notkin's work on lightweight source model extraction [MN96], although Womble is syntactic and not lexical. We were also influenced by Robert O'Callahan's recent work on type-based analysis of Java bytecode [OCa98].

9 Conclusions

Womble uses a collection of simple heuristics to extract object models from bytecode. The analysis is linear, mostly monotonic (as classes are added), and reasonably accurate. After only a few months of use of the tool, we are finding it indispensable.

In the future, we hope to improve Womble by basing its analysis on type information of the sort produced by a tool such as Lackwit [OJ97]. This would allow better resolution of association targets, better container inference, and better analysis of mutability and multiplicity.

A Java implementation of Womble is freely available from http://sdg.lcs.mit.edu/womble.

References


[G+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


