

# Lightweight Extraction of Object Models from Bytecode

Daniel Jackson and Allison Waingold  
Laboratory for Computer Science  
Massachusetts Institute of Technology  
{dnj, allison}@lcs.mit.edu

## 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. Unlike existing tools, it handles container classes, by inferring the types of elements stored in a container, and eliding the container itself. This feature is crucial for obtaining models that show the structure of the abstract state, and bear some relation to conceptual models. Although the tool performs only a simple, heuristic analysis that is almost entirely local, the resulting object model is surprisingly accurate. The paper explains what object models are and why they are useful; describes the analysis, its assumptions and limitations; evaluates the tool for accuracy, and illustrates its use, 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. Womble can also generate module dependence diagrams.

A crude object model can be built trivially from the declarations of fields. From the code fragment

```
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

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 *employees* association were instead implemented with a vector:

Vector employees;

This would result in an edge from *Company* to *Vector*, 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 still 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 *Vector*, 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 8).

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 in our benchmark suite to evaluate Womble 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 [RJB99]. The extraction mechanism is explained, followed by a discussion of its application to a suite of small and medium-sized programs. We present both some benchmark figures that demonstrate Womble's accuracy, and some sample diagrams to give a flavour of the output it produces. 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.

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 expressive 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 [RJB99] 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 from the code the object model as it would have been constructed in the early phases of design, 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. It is a strict subset of our Alloy modelling language [Jac00].

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. Similarly, a hatch at the  $A$  end says that the set of  $A$ 's mapped to a given  $B$  is fixed during the lifetime of that  $B$ . When the relation is implemented as a field in a class, a hatch at the head of the arrow says that the field is immutable; a hatch at the tail says that the  $B$  object referred to by the

field of an  $A$  object is never referred to by a different  $A$ -object. Womble currently only handles the former.

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 various other changes are impermissible. 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).

## 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.lang* and *java.util* packages).

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 treated as a container if it has no explicit references to objects of user-defined classes, but does handle objects of the class *Object*: that is, its references to other classes are all polymorphic. In order to accommodate the use of classes whose code is not available, Womble applies this criterion to the signatures of a class's methods, obtained from the call sites.

The methods considered in this analysis are those that are used by 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 the signatures of all of 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 *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 *Stream* (from *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 *Vector[Person]* would give an association to the class *Person*. But Java has only subtype polymorphism, and the type of object stored in a container is not evident from the declaration alone.

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 (by tracing backwards through the code to determine the classes of objects on the stack). These classes are taken to be the classes of the objects held within the container. This analysis is not sound: a container may take arguments of some class in a method that bears no relation to the class of elements stored. It is, however, well correlated with common idioms, and 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 *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 *zero or one* or *exactly one*. If the field is ever set to *null*, or there is a non-private constructor (or any method called by one) in which the field is not assigned a value, *zero or one* multiplicity is chosen. Otherwise, the field is assumed always to have a value, and a multiplicity of *exactly one* is chosen.

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

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

### 4.3 Inferring Mutability

Womble attempts to infer mutability markings only on the heads of arrows and not on their tails. That is, in analyzing an association from  $A$  to  $B$ , it tries to show that the code of  $A$  does not change the set of  $B$  objects associated with an  $A$ , but does not consider how the  $A$  objects associated with a  $B$  might change.

A field is inferred to be mutable if a value is assigned to it in any method of the class that is not a constructor, or any method of another class. 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 *Object* and returns *void* or any primitive type. This is perhaps the most egregious unsoundness in our analysis: never returning *void* is certainly no proof of immutability, but in standard practice, a mutable datatype tends to have such methods.

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 a field-setting instruction in another class will cause this inference to be revoked.

### 4.4 Summary of Analysis

Figures 2a and 2b summarize the analysis. Figure 2a gives an abstract syntax for the relevant parts of the Java bytecode that we use. The instructions *load* and *store* transfer objects to the stack from a local variable and vice versa; *getfield* and *putfield* have the analogous effect for fields. A particular sequence of instructions used in calling methods is highlighted: it involves placing an object on the stack that will be the ‘this’ object for the method call (always from a field in the cases we care about), then the arguments, then performing the method invocation, then sometimes downcasting the result.

Figure 2b defines the analysis itself semiformaly, as a collection of functions that extract information from the bytecode syntax tree. In addition to the syntactic structure shown, the functions assume that each instruction is labelled with a program point, and that the predecessor of an instruction in the control-flow graph can be obtained.

The key properties of this analysis, which can be inferred from the summary, are as follows:

*Local analysis.* Although some of the information extracted from the code involves more than one module, the analysis is not interprocedural. For example, the determination of whether a field is mutable (by the function *is-mutable*) involves an examination of field-setting operations throughout the code, but each of these is examined in isolation.

*Flow sensitivity.* There are several places in which the analysis relies on the order of instructions. For example, to determine the type of element stored in a container, the function *element-type* examines call sequences that include downcasts that follow the invocation. The types of local variables are determined from field declarations and the signatures of invoked methods by tracing stack operations: the *local-type* function, for example, assigns a type to a local at a particular program point by looking at what was pushed on the stack in the previous operation.

*Exploiting declared types.* The analysis uses declared types in various places: in field declarations, and in the method signatures of invocations, for example. The analysis could therefore not be applied to an untyped representation, such as machine code, but relies on the type information present in bytecode.

*Dependence on language style.* The analysis is Java-dependent, since it makes stylistic assumptions that are based on observations of Java code that might not hold for other languages (such as C++). For example, it assumes that after an invocation of a method that returns a value of type *Object*, a downcast on that value is performed immediately, if at all. If this were not the case, the analysis may fail more often in its attempt to infer container types. Note that the validity of this assumption depends not only on idioms in the source code, but also on how the compiler translates it to bytecode. Even if a programmer were to cast a result immediately, a compiler might not place the cast instruction next in the bytecode sequence.

## 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 Alcoa [JSS00], 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 3 shows part of the object model Womble generated from Grappa. A pattern aficionado will recognize this as an instance of the *Bridge* pattern [G+95].

*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 4, 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 Alcoa, 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; *Alcoa*, 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 tends to make 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 confirmed 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 –except in 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.

program	#class-files	#methods	#instrs	#nodes	#assocs	#subsets	#deps	time
Blob	54	361	29k	29	59	79	97	9s
Alcoa	123	668	60k	68	90	62	530	15s
Fusion	569	3843	313k	317	447	483	1233	131s
Grappa	84	704	46k	37	93	63	246	15s
Haystack	120	744	155k	59	87	73	371	11s
Rivet	102	1019	66k	49	78	59	474	31s
Womble	30	201	20k	18	19	42	69	5s

*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.*

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

*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.*

program	#nodes	#containers found	#user-def found	#containers missed	#array fields
Blob	29	29	0	0	0
Alcoa	68	12	0	0	1
Fusion	3	17	36	12	6
Grappa	37	17	0	0	3
Haystack	59	23	0	0	3
Rivet	49	6	1	1	18
Womble	18	9	0	5	3

*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.*

program	#nodes	#edges	#mutability confirmed	#mutability revealed	#mutability wrong	#multiplicity confirmed	#multiplicity revealed	#multiplicity wrong
Alcoa	68	86	79	6	1	79	7	0
Fusion	12	28	24	2	2	25	2	1
Haystack	7	19	19	0	0	4	15	0

*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 ‘multiplicity’ 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 took 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; '#multiplicity 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 is more reliable than the developer.

## 7 Examples

Figures 4–8 show samples of Womble's actual output. The diagrams are generated by AT&T Research's *Dot* tool. Due to *Dot*'s limitations, mutability and multiplicity markings appear in the edge labels as prefixes and suffixes. Markings corresponding to the head of the arrow appear as suffixes, and those corresponding to the end of the tail appear as prefixes. The multiplicity markings \*, ? and ! are used as discussed earlier. An association is static or immutable if it is marked with the symbol --. The absence of a marking indicates a mutable association. For example, the edge label "? r -- !" indicates a static relation with the name *r* and multiplicities *zero or one* and *exactly one* on the tail and head respectively. Subset edges are shown by dotted arrows.

### 7.1 AWT

Figure 5 shows part of the object model of Java's *Abstract Window Toolkit* (AWT) package. We used a filtering facility provided by Womble to limit the diagram to those classes that seemed to be involved in the structuring 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 suggests that the help menu has a special status.

## 7.2 JDK Hashtables

Figures 6 and 7 show two object models of the implementation of Java hashtables, from versions 1.1.7a and 2 of the JDK respectively. In the first version, *Hashtable* is implemented as an array of *HashtableEntry*. The array has been absorbed appropriately into the relation *table*. Each *HashtableEntry* is a linked list of key-value pairs. This is evident from the field *next*, a self-relation on *HashtableEntry*.

In fact, the *next* relation exposes a subtle flaw in Womble. Its multiplicity indicates that a single *HashtableEntry* is mapped to zero or more *HashtableEntry*'s, and not, as anticipate, to zero or one. This inaccuracy arises because Womble has actually inferred that *HashtableEntry* is a container, itself containing *HashtableEntry*'s. In a sense, this is correct. But it is problematic that *HashtableEntry* is not treated uniformly, and thus either elided or not treated as a container at all. This is an instance of Womble's poor handling of nested containers.

*HashtableEnumerator* has almost identical structure; note, however, that the *table* relation is static. Unlike *Hashtable*, *HashtableEnumerator* does not modify the array of entries, but instead maintains a reference (*entry*) to the next item to be delivered.

The second version (Figure 7) is similar, but adds some redundant storage. The *keySet*, *entrySet* and *values* fields are *Sets* and a *Collection* respectively that redundantly store the keys, key-value pairs, and values of the *Hashtable*. Some crucial invariants (suggested by the diagram, but not determinable from it) are that *keySet* contains exactly the keys in *table*, that *entrySet* contains exactly the *Entry*'s in *table*, and that *values* contains exactly the values in *table*.

The addition of the relation *lastReturned* is interesting. The new *HashtableEnumerator* stores not only the next item to be returned, but also the last one.

## 7.3 Haystack

Haystack is an object-oriented personal information repository. Figure 8 shows the part of its object model that represents the underlying data model. The data is kept as a collection of inter-related *Straw*'s. A *Straw* can be a *Needle*, which holds raw data stored as an *Object*; a *Tie*, which is a link between two *Straws*; or a *Bale*, which is an aggregation of data.

Each *Straw* has a unique id, via the association *strawID*. The multiplicity markings of the relation *strawID* show that there is exactly one id per *Straw*, but not that each such id is unique. This property cannot be inferred because the ids are not generated locally, but

are stored in another class (not shown in this model) and thus escape the *Straw* class in which they are assigned. Womble infers correctly that ids are immutable.

A *Tie* contains references (*backP* and *forwardP*) to the *Straws* it links, as well as references (*backPID* and *forwardPID*) to their ids; these latter relations seem to be redundant.

A *Straw* keeps track of its relationships to other *Straws* in two *StrawTies* objects. *StrawTies* has a field *links* that is a hashtable mapping *String* labels to *Vector*'s of *StrawID*'s. The object model does not reflect this correctly, illustrating Womble's inability to handle nested containers: *Vector* is not elided and its element type is not shown. Moreover, Womble also fails to distinguish the keys and values of the table.

A simple but crucial feature of the data model that is easily seen in Womble's output is that a *Bale* does not directly contain a group of *Straw*'s. Instead, *Bale* is a subset of *Straw*, and a *Bale* is linked to its *Straws* by *Tie*'s.

#### 7.4 CTAS

In an exercise in software design, we reengineered a component of CTAS, an air-traffic control system developed by NASA and deployed in several US airports [JC00]. We applied Womble to our Java reimplementations of the component.

Figure 9 shows part of the object model obtained by Womble from one of the packages. It focuses on a crucial aspect of the component that manages the assignment of aircraft to analysis processes. *Client* is a class whose instances are proxies for analysis processes that run on other machines; *RAManager* is a singleton class that maintains the assignment. *AircraftTable* is a database holding a collection of *Aircraft* records indexed on *AircraftID*.

A couple of invariants are suggested by the diagram. Notice the two relations from *RAManager* to *AircraftID*. One, *unassigned*, represents a set of aircraft that have not been assigned to RAs; the other, *acidRAs*, represent the keys of a hashtable (elided by Womble) that maps aircraft to their route analyzers. An aircraft that is not assigned to an RA should be in the *unassigned* set. In fact, this invariant is not maintained, and the review elicited by our examination of Womble's output exposed a serious flaw in our code.

There is redundant storage of aircraft ids in a pattern similar to that seen in Haystack. A variety of such patterns arise as common idioms in object models and are easy to identify with experience.

Each *Client* object, being a proxy for a route analyzer, is related to a *ClientGroup* by the relation *clientGroup*; correspondingly, each *ClientGroup* maintains a set of its *Client*'s. Womble correctly elides the set container and connects *ClientGroup* to *Client* by the relation *clients*. It also successfully infers that *clientGroup* is static, but *clients* is not (since new *Client*'s may be created and assigned to existing *ClientGroup*'s).

## 8 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 also 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, but fails for nested containers.

*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 reasonable. 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 or when a reference is only transiently null. 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 there are no assignments to the relevant field outside a constructor. A subclass might violate this; it might, in other words, not behave like a subtype. Womble therefore reevaluates 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 primitively-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.

## 9 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. At the time of writing, this analysis had not been implemented.

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.

We ran Rational Rose on the CTAS code for which Womble generated the object model shown in Figure 9. The result is shown in Figure 10. Rose does not produce a model of this form in a single step. We had to manually select the classes to be shown, since Rose does not determine automatically, for example, which of the standard Java library classes are used in the code. We also had to adjust the layout of the model. Rose does not include a layout algorithm comparable to Dot, and without manual adjustment, the model is not legible.

The Rose model gives much less information than the Womble model. It does not infer any mutability or multiplicity markings. Containers are not distinguished from other classes, and their elements are not inferred. As a result, there is no relationship between *AircraftTable* and *Aircraft* or *AircraftID*, or between *RAManager* and *Client*. The two-way coupling of *Client* and *ClientGroup* is missing. The model shows that *AircraftTable* and *RAManager* contain *HashTables*, but not what these hash tables contain. Given the pervasive use of containers, this results in a largely disconnected object model: in our first cut, before we manually added *HashTable* and *Vector*, there were barely any connections at

all! Surprisingly, there is no association between *Aircraft* and *AircraftID* either. It seems that Rose erroneously infers that *AircraftID* is an attribute of *Aircraft*, even though it is not a primitive type. Much of the benefit of a graphical object comes from the sharing that it shows: in the Womble model, we can see, for example, that *AircraftID*'s are used in *RAManager*, *AircraftTable* and *Aircraft*, and guess the invariant of *AircraftTable* that each *Aircraft* is indexed on its *AircraftID* field. The Rose model provides none of this.

Unlike Womble, Rose cannot be used incrementally. To analyze a program, it appears to require that every class it uses be present. This is a major burden in practice. In a large system, it can be difficult to determine the extent of dependences without tool support, so one may be forced to include every class in a system. It also rules out analysis of incomplete programs, or of programs that uses libraries that are not installed on the machine on which the analysis is being conducted.

Object models can also be constructed by processing the results of more traditional analyses. Tools such as Imagix and Sniff [Bis92], for example, can generate reports about call relationships and usage of data components, from which another tool could generate object models. Many reverse engineering tools developed by researchers (such as [CS90, LC94]) use the same underlying information as these commercial tools, but postprocess the results (eg, by computing clusters of modules to suggest reorganization). Even the simple analysis described here, however, uses more information than these tools tend to provide. Without the rather basic dataflow analysis that we perform to infer element types of containers, for example, the extracted object models would suffer from the same problems as the Rose model discussed above.

Our approach was partly inspired by Murphy and Notkin's work on lightweight source model extraction [MN96], which provides the developer with a mechanism for defining new analyses in terms of simple lexical patterns. Womble might benefit from a similar approach, but syntactic in flavour, in which simple analyses can be described by pattern matching on the bytecode syntax tree. Tweaking Womble's heuristics would have been easier had such a tool been available; it might also make sense to offer the user a variety of heuristics so the tool can be customized for the idiosyncracies of the code at hand.

## 10 Conclusions and Future Research

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. Despite its limitations, we find it an invaluable tool in our everyday work.

Womble has several deficiencies. The major analysis flaw is the inability to handle nested containers. It is quite common for example, for a class *C* to have a field that is a hashtable mapping strings to vectors of some other element type *E*; Womble fails to link *C* and *E*. Womble also fails to distinguish keys and values in a hashtable. Although Womble's heuristics work well for many of the containers that arise in practice, we have come to think that a more systematic approach is needed.

We are currently experimenting with a new version of Womble (‘Superwomble’) that has the same goals as Womble, but whose analysis is more semantic in flavour. Superwomble derives a ‘specification’ from each class that gives the types of methods and fields, in which appearances of other classes are treated as type parameters. For example, in the *Vector* class of the *java.util* package, the specification shows that the type of element in the array field matches the type of the argument to the method *addElement*. The specification is then instantiated at each use of the class.

This analysis can be viewed as a simplified form of the type inference performed by the Ajax tool [OCa98]. Fields that have the same representation may be assigned distinct types. For example, a field that uses a string to hold a person’s name and a field that uses a string to hold a social security number will have associations with different instantiations of *String*.

Womble performs this kind of ‘splitting’ only for containers: two fields that are both declared as *Vector*, for example, will not generally have associations to the same node (unless the element types of the two vectors match). Moreover, it intertwines splitting with the issue of eliding containers to hide representation details. Superwomble, by applying splitting uniformly on all classes, cleanly separates these two issues.

This approach solves several of the problems inherent to Womble’s ad hoc analysis. Nested containers are no longer a problem; using parameterized specifications and typing the stack, the types in a nested container can be determined with no more difficulty than determining the type in any container. Keys and values in a table are also easily distinguished, since a parameterized specification differentiates the types of each field in the table class.

It remains to be seen whether this technique can be made to scale as readily as Womble’s. A new strategy for eliding containers is needed, since the technique exposes the representations of all classes. Currently, Superwomble uses some built-in abstraction rules that coalesce a chain of edges in the object model into a single edge. For example, the bucket structure of hash tables is eliminated, leaving only abstract ‘key’ and ‘value’ fields.

The result of applying Superwomble to the CTAS code discussed in Section 7.4 is shown in Figure 11. Comparing it to the Womble model of Figure 9, we see that keys and values in a table are now properly distinguished. The *AircraftTable* class, for example, contains a table that maps *AircraftID* as key to *Aircraft* as value; the table itself has been elided, but is evident in the labels *aircrafts.keys* and *aircrafts.values*. Similarly, *RAManager* has two tables, one mapping *Client* to *Integer*, and one mapping *AircraftID* to *Client*. In the Womble output, however, these tables cannot be distinguished from, for example, heterogeneous lists.

We are not in fact convinced that eliding containers is the right thing to do. When examining the model, one expects to see implementation details, including the representations of classes. Including containers might not only make the diagram easier to understand, but would also allow us to show potential sharings of the containers themselves

(distinct from sharings of their elements). In Superwomble, elision of containers will be a distinct phase that will be applied more judiciously, with input from the user.

In our use of Womble, the graph layout step has been a constant irritation. Although Dot produces generally excellent layouts, getting a reasonable pagination is tricky, and we spend too much time sticking sheets of paper together. We are therefore considering a command-line version of Womble—a kind of ‘object model grep’—that will produce a list of associations in a simple textual form, to which various forms of filtering and layout can be subsequently applied.

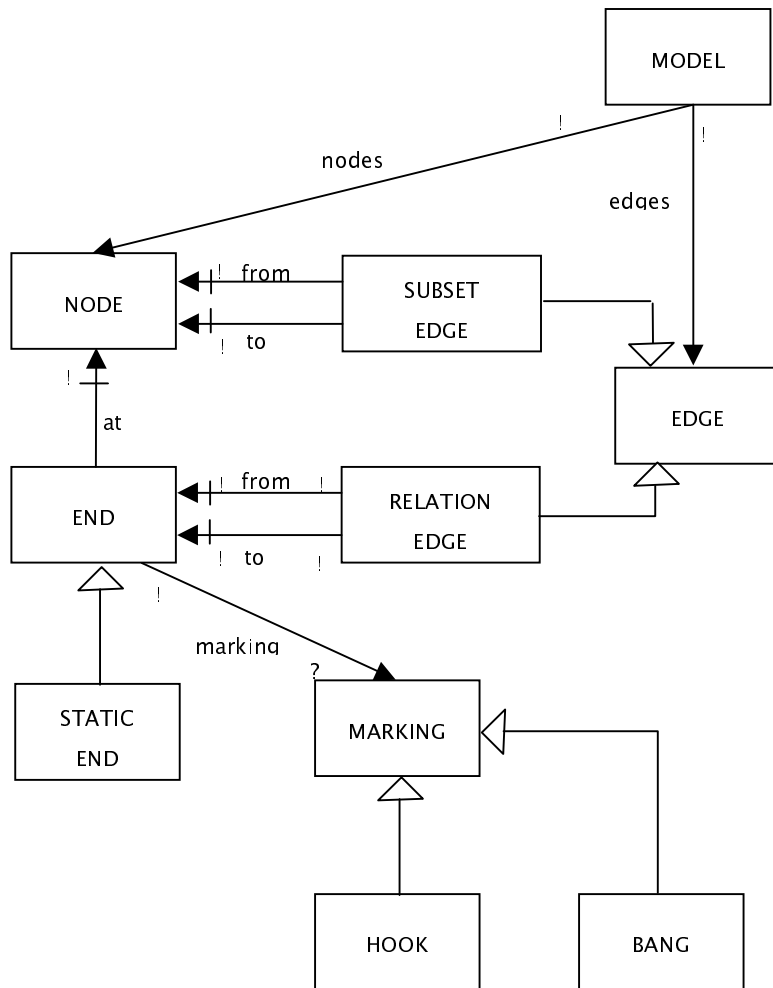
### **Acknowledgments**

We are grateful to the developers of Haystack and Fusion for participating in our experiments. Robert O’Callahan gave us the front end of his Ajax tool for processing bytecode, and gave advice generously. We gratefully acknowledge the support of the MIT Center for Innovation in Product Development, funded under NSF Cooperative Agreement Number EEC-9529140, and of the MIT Undergraduate Research Opportunities Program. Womble is freely available from our web site: <http://sdg.lcs.mit.edu/womble>.

## References

- [Bis92] W. Bischoffberger. Sniff: a Pragmatic Approach to a C++ Programming Environment. *Proc. 1992 USENIX C++ Conference*, 1992.
- [C+93] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin and Helena Gilchrist. *Object-Oriented Development: The Fusion Method*. Prentice Hall, September 1993.
- [CD94] Steve Cook and John Daniels. *Design Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.
- [CS90] Song C. Choi and Walt Scacchi. Extracting and Restructuring the Design of Large Systems. *IEEE Software*, January 1990, pp. 66–71.
- [DW98] Desmond F. D’Souza and Alan Cameron Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.
- [G+95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Jac95] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 2, April 1995, pp. 109–145.
- [Jac00] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation*. Technical Report MIT-LCS-797, MIT Laboratory for Computer Science, Cambridge, Mass., February 2000. Available at: <http://sdgl.lcs.mit.edu/~dnj/publications>.
- [JC00] Daniel Jackson and John Chapin. Redesigning Air-Traffic Control: A Case Study in Software Design. *IEEE Software*, May/June 2000.
- [JSS00] Daniel Jackson, Ian Schechter and Ilya Shlyakhter. Alcoa: The Alloy Constraint Analyzer. *International Conference on Software Engineering*, Limerick, Ireland, June 2000.
- [LC94] Panagiotis Linos and Vincent Courtois. A Tool for Understanding Object-Oriented Program Dependences. *Proc. 3rd Workshop on Program Comprehension*, Washington, DC, November 1994.
- [MN96] Gail C. Murphy and David Notkin. Lightweight Lexical Source Model Extraction. *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, July 1996, p. 262-292.
- [MNS95] Gail C. Murphy, David Notkin and Kevin Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '95)*, October 1995.
- [OCa98] Robert O’Callahan. *Scalable Program Analysis and Understanding Based on Type Inference*. PhD Thesis Proposal, School of Computer Science, Carnegie Mellon University, Pittsburgh PA. January 1998. [www.cs.cmu.edu/~roc/thesis-proposal.html](http://www.cs.cmu.edu/~roc/thesis-proposal.html).

- [R+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. *Object Oriented Modeling and Design*. Prentice Hall 1991.
- [RJB99] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [Spi92] J. Michael Spivey. *The Z Notation*. Second edition, Prentice Hall, 1992.
- [SvG98] Jochen Seemann and Juergen Wolff von Gudenberg. Pattern-based Design Recovery of Java Software. *Proc. Foundations of Software Engineering*, November 1998.



*Figure 1: An object model of our modelling notation, expressed in the notation itself*

*code* ::= *class*\*  
*class* ::= *class-name* *field-decl*\* *method*\*  
*field-decl* ::= *field-type* *field*  
*field-type* ::= *class-name* | *primitive-type*  
*method* ::= *method-sig* *instr*\*  
*method-sig* ::= *class-name* *method-name* *arg-type*\* *result-type*  
*instr* ::= *stack-pusher* | **store** *local* | **putfield** *field* | **cast** *class-name*  
*stack-pusher* ::= **load** *local* | **getfield** *field* | **invoke** *method-sig* | **push-null**  
*call-seq* ::= (**getfield** *field*) *stack-pusher*\* **invoke** *method-sig* [**cast** *class-name*]  
*primitive-type* ::= **boolean** | **short** | **int** | **long** | **char** | **float** | **double** | **byte** | **void**  
*builtin-type* ::= *primitive-type* | (*classes of java.lang*)

Figure 2a: Abstract syntax of selected Java bytecode fragments

$f\text{type} : \text{field} \rightarrow \text{type}$   
 $f\text{type}(f) = \text{declared type of } f$

$\text{pusher-type} : \text{instr}, \text{program-point} \rightarrow \text{type}$   
 $\text{pusher-type}((\text{store } x), p) = \text{local-type}(x, p)$   
 $\text{pusher-type}((\text{getfield } f), p) = f\text{type}(f)$

$\text{is-container} : \text{field} \rightarrow \text{bool}$   
 $\text{is-container}(f) =$   
 for all instructions  $(\text{invoke } f\text{type}(f) \text{ m } a_1 a_2 \dots r)$  : arg types  $a_1, \dots$  and result type  $r$  are built-ins  
 and exists an instruction  $(\text{invoke } f\text{type}(f) \text{ m } a_1 a_2 \dots \text{Object})$  with some  $a_i = \text{Object}$

$\text{local-type} : \text{local}, \text{program-point} \rightarrow \text{type}$   
 $\text{local-type}(x, p) = t$  where  
 predecessor of  $p$  is  $q$ , and  $\text{local-type}(x, q) = t$   
 or instruction at  $p$  is  $(\text{store } x)$ , instruction at  $q$  is a stack-pusher  $s$  and  $\text{pusher-type}(s, q) = t$

$\text{element-type} : \text{field} \rightarrow \text{type}$   
 $\text{element-type}(f) = t$  where  
 $\text{is-container}(f)$  and  
 exists call-seq  $((\text{getfield } f) \text{ stack-pusher}^* \text{invoke method-sig cast } t)$   
 or exists call-seq  $((\text{getfield } f) \text{ op}_1 \text{ op}_2 \dots \text{invoke method-sig } \dots)$   
 where  $\text{op}_i$  occurs at point  $p$  and  $\text{pusher-type}(\text{op}_i, p) = t$

$\text{is-mutable} : \text{field} \rightarrow \text{bool}$   
 $\text{is-mutable}(f) =$   
 $\text{is-container}(f)$  and some  $(\text{invoke } f\text{type}(f) \text{ m } a_1 a_2 \dots r)$   
 with some  $a_i = \text{Object}$  and  $r \in \text{primitive-type}$   
 or not  $\text{is-container}(f)$  and exists an instruction  $(\text{putfield } f)$

$\text{head-multiplicity} : \text{field} \rightarrow \{*, ?, !\}$   
 $\text{head-multiplicity}(f) =$   
 if  $\text{is-container}(f)$  or  $f$  declared as array then \*  
 else if (class in which  $f$  declared has constructor that contains no  $(\text{putfield } f)$   
 or code contains sequence  $(\text{push-null } (\text{putfield } f))$ )  
 then ? else !

$\text{tail-multiplicity} : \text{field} \rightarrow \{*, ?, !\}$   
 $\text{tail-multiplicity}(f) =$   
 if (class in which  $f$  declared has some non-private method with  $f\text{type}(f)$  as an argtype or result)  
 then \* else ?

*Figure 2b: Analysis functions*

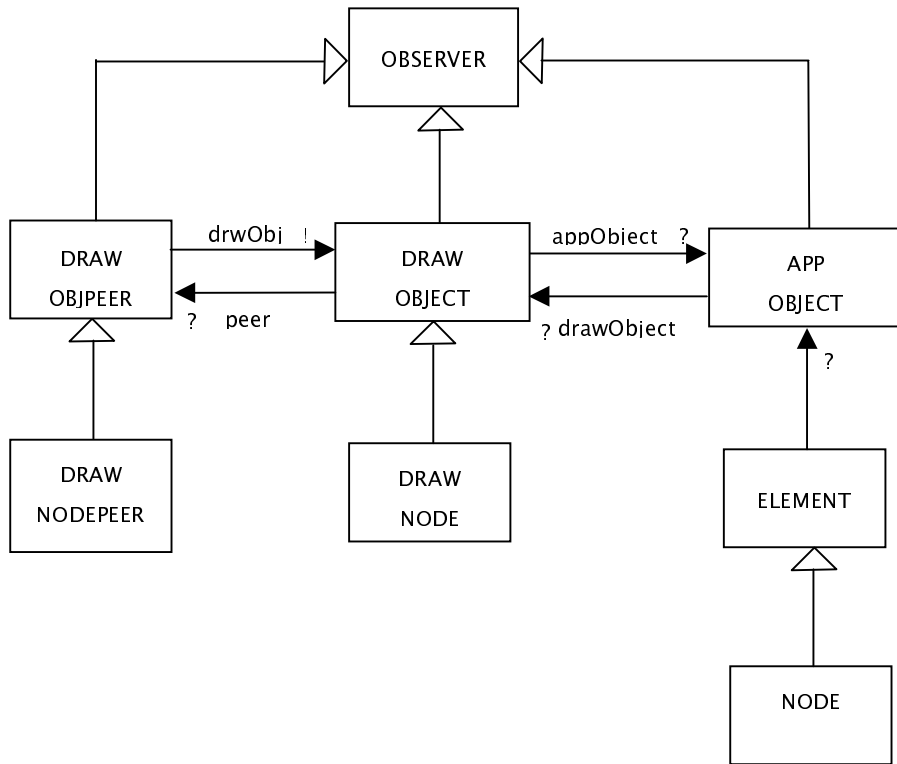


Figure 3: A design pattern in the object model of Grappa, a graph drawing tool

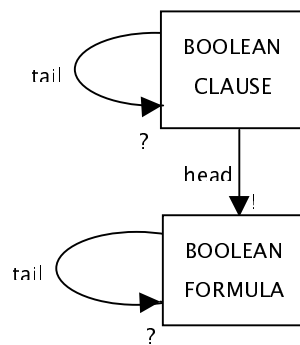


Figure 4: Disjunctive normal form representation of boolean formulas in Nitpick

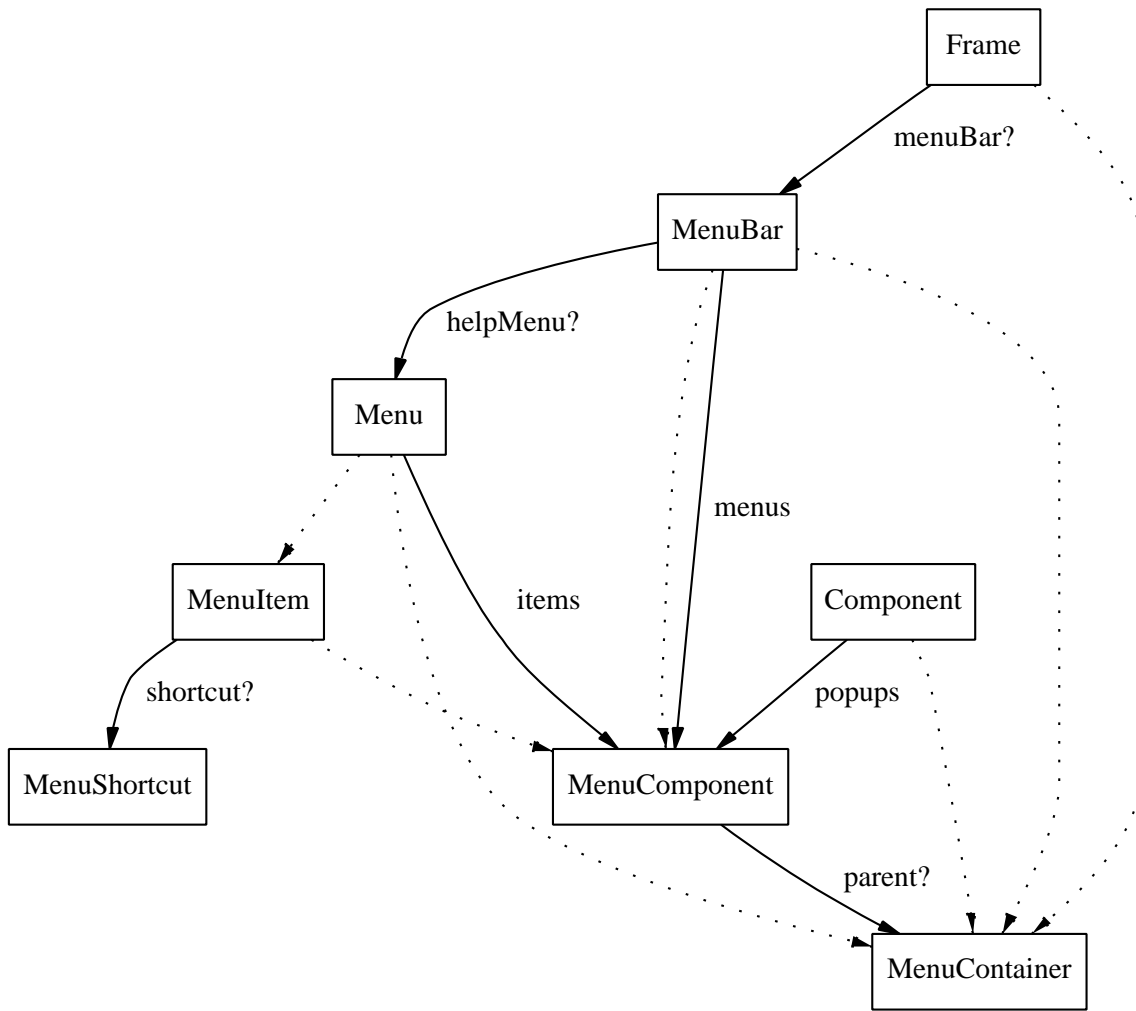


Figure 5: Java's Abstract Windowing Toolkit

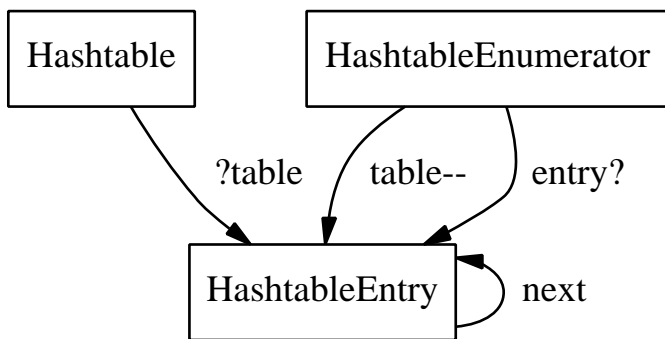


Figure 6: Hashtables in JDK 1.1.7a

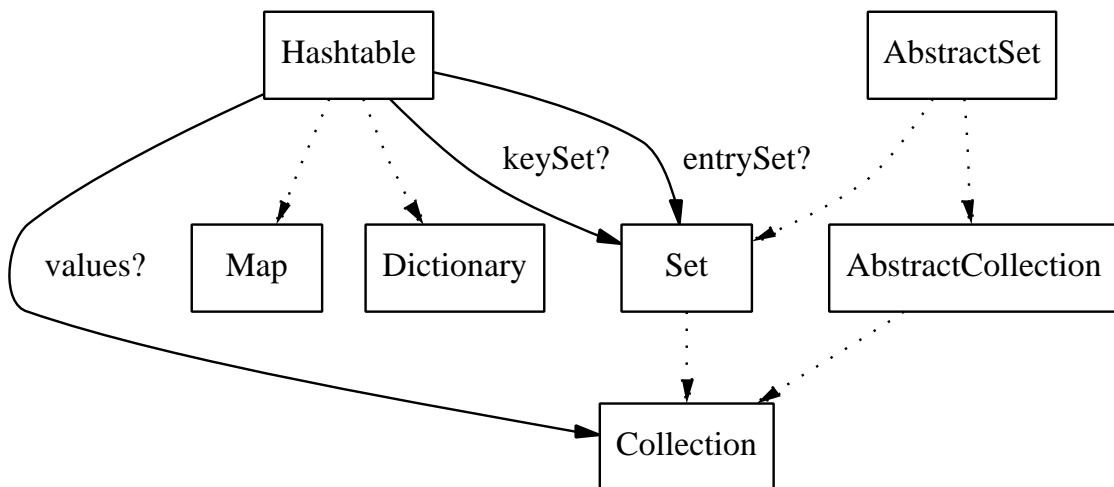


Figure 7: Hashtables in JDK 1.2

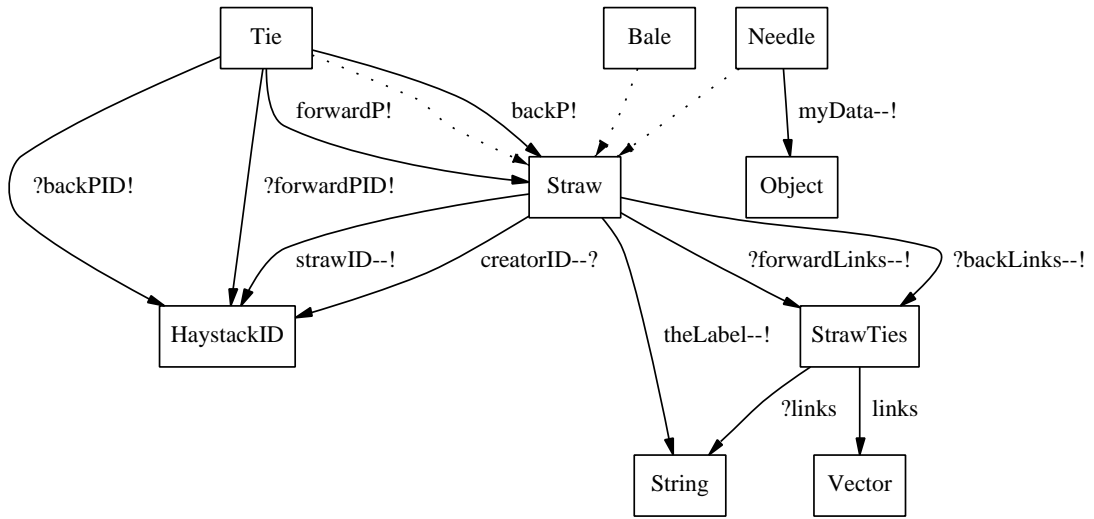


Figure 8: Haystack data model

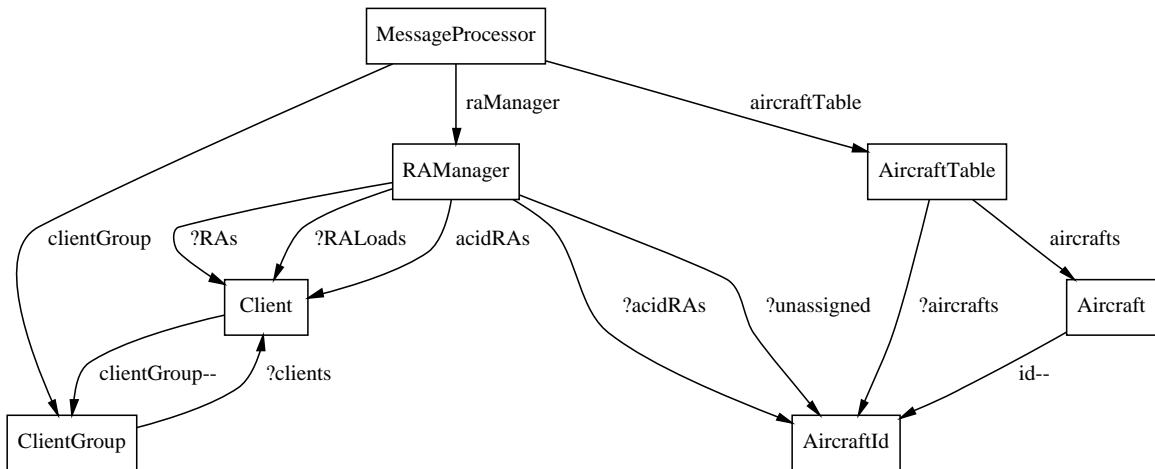


Figure 9: Part of a package from the CTAS air-traffic control system extracted by Womble

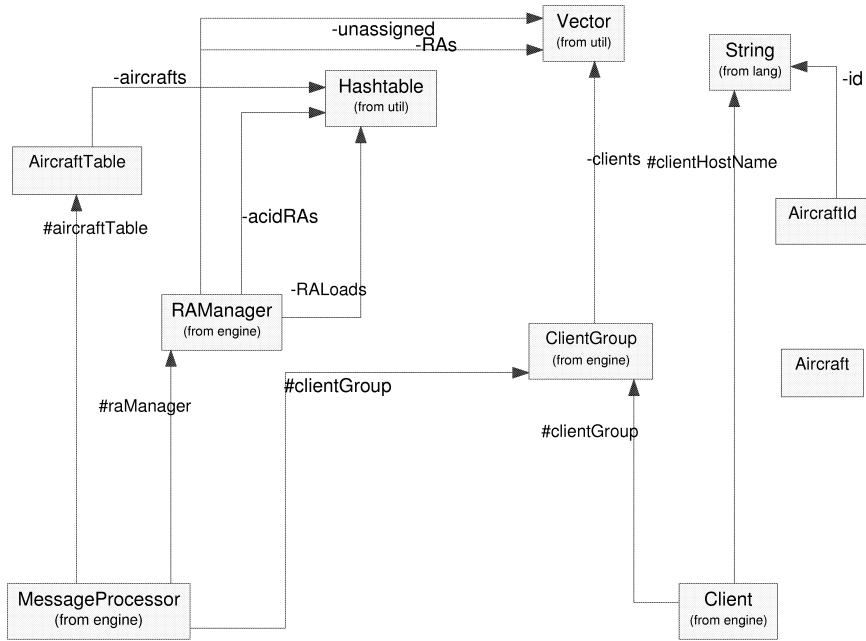


Figure 10: A model for the same CTAS code as in Figure 9, analyzed by Rational Rose

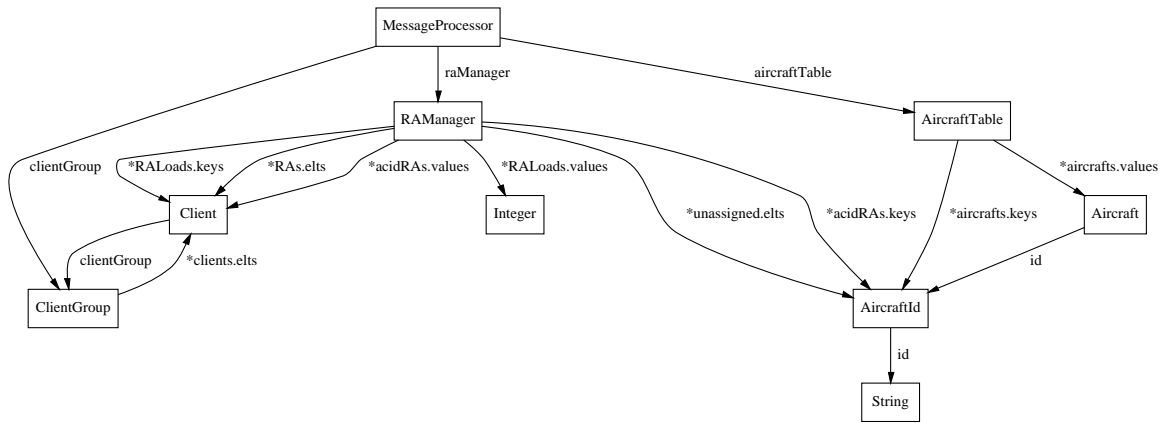


Figure 11: A model for the same CTAS code as in Figure 9, analyzed by SuperWomble