6.170 Laboratory in Software Engineering
Fall 2005
Problem Set 3: Mechanical Extraction of Object Models from Bytecode
Due: **Thursday**, October 6, 2005 at 1:00pm

Contents:

Introduction

This problem set will deepen your understanding of object models by having you write a tool to mechanically extract them from Java bytecode. (In other words, you will write a program that reads in compiled Java `class` files and draws a picture of them.) By completing this problem set you will learn: how easy or difficult it is to express each feature of the object model notation in Java; and what is easy for a programmer to understand from reading code, but is difficult for an automatic extraction tool to deduce.

We have listed the *estimated required time* for each problem. The time is loosely based on the amount of time that the TA's spent working on it, as well as the level of learning curve we estimate from one question to the next. It is only meant to help guide your effort; do not feel discouraged if you spend more time on a question. In fact, all the questions are cumulative, so if you spend more time figuring out how to tackle 1 question, it will help you solve the following questions as well.

How to Get Started:

Step 1: First of all, you must make sure you have access to the `dot` program. On Athena, type `add outland` to add it to your search path. Elsewhere, download a pre-built binary from graphviz.org. If you are working from home, you must make sure that `dot` is on your program search path before loading Eclipse (otherwise Eclipse won't find it). More instructions can be found on the working at home document in the tools section.

> **Note:** To verify whether your DOT is working or not, type "dot -V" on a command line prompt. You must have version 1.11 or later. (If you're on Windows, you must make sure your DOT version is 2.6 or newer).

Step 2: Start up Eclipse and check out ps3. Go to the left side of the window, under `ps3`, you should see a file named `build.xml`. Right click on it, then click `Run As`, then click `Ant Build`. Make sure there are no error messages of any sort.

> In the past, students have reported many troubles with running Dot on a Sun Athena. Dot appears to work fine on Linux Athena and on home machines. If Dot is giving you trouble, you may need to switch to a Linux Athena machine, then type
>
> ```
> cp -r 6.170 6.170.backup
> ```
>
> to make a copy of your old 6.170 files. Then type:
>
> ```
> eclipse
> ```
>
> and then repeat the initial setup steps you did in problem set 0 (to create the two CVS repositories, and then set Java Compiler level to 5.0)

Step 3: Start a web browser, then load the file `ps3/doc/report.html` into your browser. Make sure the file looks okay on your browser. There must be 1 picture under question 1 and 8 pictures under question 6. Throughout this problem set, you will be typing your answers into the `report.html`, so it is very important that you can complete all the steps listed so far. If there are any problems, please post your question to the 6.170 web forum or email a TA.

Problem 1 and Problem 2: Preliminary Exercises

Approximate time to completion: 15 to 30 minutes

This pset will require you to type your answers into an HTML file. You do not need to know any HTML tags to do this: just open up the HTML file, search for "your text here", and write your answer in.

Please use a regular text editor (eg, Eclipse's default text editor, emacs, vi, notepad) to edit the HTML, or a program that is designed to edit HTML. Please do not use a program like Microsoft Word, that can save HTML files but is not really an HTML editor.

How to draw the diagrams:

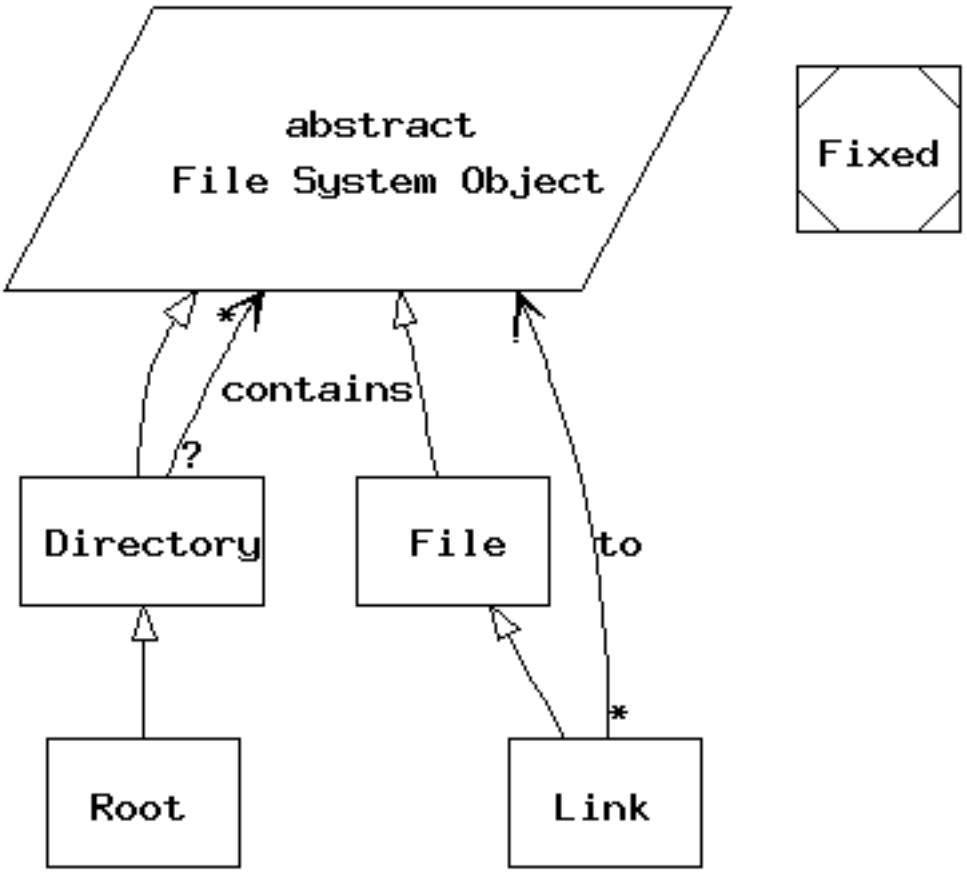
During this problem set you will be producing diagrams using the "Graphviz/Dot" tool. To use this tool, you (or your Java analysis tool) should write out the list of nodes and list of edges into a ".dot" file. Then you can use the "dot" program to automatically convert your ".dot" file into a graphical ".png" file.

Here is a sample file that will probably have all of the features that you will need to know:

```
    digraph FileSystem {

        // The following 2 lines define the default NODE and EDGE
attributes
        node [shape="box", fontname="courier", fontsize="12"];
        edge [arrowhead="open", arrowsize="0.8", fontname="courier",
fontsize="12"];

        // subsets
        // note that the edges are nominally backwards
        // in order to get dot to draw them how one would expect
        // use the parallelogram shape for abstract sets
        FSO [label="abstract\nFile System Object",
shape="parallelogram"];
        FSO -> Directory [arrowhead="none", arrowtail="empty"];
        Directory -> Root [arrowhead="none", arrowtail="empty"];
        FSO -> File [arrowhead="none", arrowtail="empty"];
        File -> Link [arrowhead="none", arrowtail="empty"];

        // relations
        Directory -> FSO [arrowhead="open", label="contains",
        headlabel="*", taillabel="?"];
        Link -> FSO [arrowhead="open", label="to",
        headlabel="!", taillabel="*"];

        // use the Msquare shape for fixed sets
        Fixed [shape="Msquare"];
    }
```

If you save that in a file called `example.dot` and run the following commands you'll get the picture below:

```
    dot -Tpng -o example.png example.dot
```


abstract
File System Object
Fixed
*
contains
?
to
!
Directory
File
Root
Link
*

TODO:

Problems 1 and 2 are listed in the file `ps3/doc/report.html` that we provided for you. Please open up the file, and fill in your answers to problems 1 and 2.

Problem 3: Compilation, Bytecode, and Virtual Machines

Approximate time to completion: 15 to 30 minutes

A *compiler* is a program that translates a program written in a textual form (ie, source code), to an equivalent binary representation. The textual form is convenient for people to read and write, and the binary form is convenient for a computer to execute. This translation process is called *compilation*.

In Java, source code is written in `.java` files, which are compiled to `.class` files.

The Eclipse development environment compiles your source code automatically as you edit it. Eclipse puts the `.class` files that it generates in the `bin` subdirectory of your project, which is hidden from view in the Eclipse Java perspective; you'll see it if you look in the file system, or use the Eclipse Resource perspective. (Eclipse can be configured to put `.class` files somewhere other than the `bin` subdirectory, but that is their conventional location.)

If you work with another set of tools, you may have to manually invoke a Java compiler to translate your `.java` files to `.class` files.

In the case of Java, Java source code is usually compiled to Java *bytecode* (stored in `.class` files), which is the binary format for the Java *virtual machine* (JVM). The JVM translates Java bytecode to the native machine instructions at program runtime.

Java Bytecode

The Java `.class` file format is documented in the Java Virtual Machine Specification, chapter 4.

Hint: you don't need to read the whole thing, and you don't want to print any of the documentation. It is long, so please just treat it as an optional reference material that you can browse through when you encounter specific questions.

In Java bytecode, constructors are invoked with the `invokespecial` opcode. Here's a nice optional article about it.

The `javap` tool that comes with the Java SDK may be used to disassemble Java `.class` files so that you can look at what a CLASS file is composed of. **Hint:** This kind of dissassembly is not necessary for this problem set; it is just mentioned here, just in case you're curious about it.

TODO:

Please edit the file `ps3/doc/report.html` and fill in your answers to problem 3.

Problem 4: The BCEL library

Approximate time to completion: 15 to 30 minutes

BCEL is a library that gives you an API for reading Java `.class` files. The BCEL Manual gives a good overview of the `.class` file format and the BCEL API.

- BCEL home page
- BCEL Manual
- BCEL API
- InvokeSpecial the BCEL class representing the bytecode instruction to invoke a constructor

Hint: you don't need to read the whole thing, and you don't want to print any of the documentation. It is long, so please just treat it as an optional reference material that you can browse through when you encounter specific questions.

If you are using the Kaffe VM, you may see the warning message (WARNING Bad bytecode!...) when running the precompiled BCEL binaries, but it may be safely ignored.

TODO:

Please edit the file `ps3/doc/report.html` and fill in your answers to problem 4.

Problem 5: THINKING about writing a static analysis tool for Java bytecode

Approximate time to completion: 15 to 30 minutes

At the end of Problem Set 3, you will produce a simple static analysis tool to extract object models from Java bytecode.

We expect your program to start out fairly short and simple, and to not grow that much more complicated as you progress through the problem set. If you find yourself writing a large or complicated program, or feeling that such a program is necessary, please speak with one of the course staff. Read this entire document and do the exercises before starting to program.

TODO:

Please edit the file `ps3/doc/report.html` and fill in your answers to problem 5. For each question, write a brief comment about how your tool will extract it (if at all: some features can not be extracted automatically). If you believe a feature can not easily be extracted automatically, just say it can't be done. The first two boxes have been filled in for you as an example.

Problem 6: Writing a static analysis tool for Java bytecode

Problem 6.1: Arrays

Approximate time to completion: 1 hour

Look at `inputs/arrays/Student.java` and `inputs/arrays/Recitation.java`.

TODO #1:

First of all, draw the object model diagram by hand. You must use the Dot tool and save it to a file called `doc/manual.dot/arrays.dot` so that we know where to look for it.

To use Dot:

- 1. save your Dot file in `doc/manual.dot/arrays.dot`
- 2. In Eclipse, look at the left of the window, and find `build.xml` under ps3
- 3. Right click on `build.xml`
- 4. Choose Run.. then choose "Ant Build".
- 5. This will then automatically convert your `arrays.dot` into `arrays.png`

TODO #2:

We have already provided `ObjectModel.java` file for you. Your job is to add features to it, modify it, or enhance it until it can **automatically** create an Object Model Diagram that is as close to the **manually-drawn** diagram as possible.

Your job can be summarized like this:

- 1. Edit `ObjectModel.java`
- 2. Run `build.xml` which will automatically generate `doc/extracted.dot/arrays.png`
- 3. Load `doc/report.html` in your web browser
- 4. Go to section 6.1 and compare the manually-drawn picture with the picture produced by your tool.
- 5. If it's as close as you can get, then stop. Otherwise, go back to step 1...

In fact, it is not possible for the tool to derive all the relaionships and all the multiplicity markers automatically. So don't get frustrated if your tool can only get half way there. Think about your answers to Problem 5: some of the features you should be able to extract automatically, and some you shouldn't be able to.

We require that you use the Apache Foundation's Byte Code Engineering Library (BCEL) for parsing Java bytecode into memory. You will find the BCEL JAR file in the `lib` subdirectory of your ps3 project, and on your classpath (if you check out the project using Eclipse).

We also require that your extractor produce files in the Graphviz/Dot format for textually describing graphs. The Dot tool reads these textual descriptions of graphs and automatically renders a graphical image of the graph.

TODO #3:

After you're done, look at the 2 pictures, and answer the question at the end of "Problem 6.1" in the `doc/report.html` file.

Problem 6.2: Collections

Approximate time to completion: 30 minutes

Look at `inputs/collections/Student.java` and `inputs/collections/Recitation.java`.

TODO #1:

Draw the object model diagram by hand, using the Dot language, and save the output to the file `doc/manual.dot/collections.dot`. The `build.xml` script will automatically convert it to a graphical file `doc/manual.dot/collections.png` which will be automatically included in `doc/report.html`.

TODO #2:

Enhance your `ObjectModel.java` until its automatically-generated picture is as close to the manually-drawn picture as possible.

TODO #3:

After you're done, look at the 2 pictures, and answer the question at the end of "Problem 6.2" in the `doc/report.html` file.

Problem 6.3: Yourtests

Approximate time to completion: 30 minutes

TODO:

Make up a few Java files of your choosing, and put them in the `inputs/yourtests/` directory. Your example must include some interesting features (such as inheritance) so that they can increase your confidence in your tool.

Enhance your `ObjectModel.java` until it can properly handle your sample examples.

After you're done, answer the question at the end of "Problem 6.3" in the `doc/report.html` file.

Bonus points:

If you have the time and the inclination, do something above and beyond what's called for here. Do something that perform more in-depth analysis, or implement some more advanced heuristics for bonus points.

Problem 6.4: Singleton / Fixed Set

Approximate time to completion: 30 minutes to 2 hours

Singleton / Fixed Set

Given a class C, if it meets all the following criteria:

1. all constructors of C are private
2. a constructor of C is invoked by C's `<clinit>` method (that's the name of the *class initializer* in bytecode)
3. no other static method in C invokes a constructor of C
4. C has at least one static field of type C

Then it is **likely** that the set C is a fixed set (also known as a *singleton*). The criteria represent a **heuristic**: when the criteria is met, it is highly likely that C is a fixed set. It is still possible for C to be a fixed set and not meet these criteria, and it is possible for C to not be a fixed set if it meets this criteria. Sometimes it is more useful if the tool uses some heuristics such as this, at the risk of occasionally being wrong, rather than only giving information that is provably correct.

TODO #1:

Look at `inputs/singleton/Singleton.java`.

Draw the object model diagram by hand, using the Dot language, and save the output to the file `doc/manual.dot/singleton.dot`. The `build.xml` script will automatically convert it to a graphical file `doc/manual.dot/singleton.png` which will be automatically included in `doc/report.html`.

Use the `Msquare` shape to represent a fixed set in Dot: eg, `digraph g { fixed [shape="Msquare"]; }` .

TODO #2:

Enhance your `ObjectModel.java` until its automatically-generated picture is as close to the manually-drawn picture as possible.

Your tool should identify the `inputs/singleton/Singleton` class as a fixed set.

TODO #3:

After you're done, look at the 2 pictures, and answer the question at the end of "Problem 6.4" in the `doc/report.html` file.

Problem 6.5: Function Table

Approximate time to completion: up to 30 minutes

TODO:

Look at `inputs/functiontable/*.java`

This is some non-trivial code that is large enough to be interesting, but small enough that your tool should be able to draw a reasonable object model for it. This code has the qualities discussed above that your tool can handle: arrays, no use of collections, and a fixed set.

You do not need to know what this code does. As far as you're concerned, it's just an input to your tool. If you are interested, here's a brief description: This code is used to represent conditional probability tables in a Bayes net solver for 6.825. A conditional probability table may be considered as a mathematical function that maps discrete input variables to a continuous output variable (a probability). Since the input variables are discrete, the entire domain of the function can be enumerated in memory (which is what this code does). For example, the table describing an unbalanced coin might be something like (HEADS=0.4, TAILS=0.6). This code uses arrays instead of the Java collections classes because all of the data is totally ordered, immutable, and known at object creation time: so arrays with binary search are feasible because the contents of the arrays never changes, as well as being time and space efficient. The `Compute` class contains a bunch of static methods that do various computations with these tables.

Run your tool to automatically generate an object model diagram for it. Don't worry about the appearance of the diagram (it will be big and messy). You don't have to examine the graph in too much detail. You don't have to compare the graph to the actual code. This exercise is basically meant to stress-test your tool on a large input. If there are glaring mistakes in the diagram, that means you have to fix your ObjectModel.java.

Problem 7

Approximate time to completion: up to 30 minutes

In problem 6, the Graphviz/Dot tool did the graph layout for you, so your program didn't really need to have any data structures: it could just dump out the data as it was read in. Now consider writing a tool such as Javadoc: one of the things that Javadoc does is produce a textual rendering of a program's inheritance hierarchy (eg, hierarchy for java.util). Therefore, such a tool must have some internal representation of the data it is manipulating, so that it can be printed out in some specific order.

TODO:

Enhance your tool so that it produces a rendering of the hierarchy similar to the one produced by Javadoc. What's important is the essential structure and ordering, not the fancy formatting. Your tool should append the output onto the end of the `.dot` file.

For example, the output for one of the test might look something like this: (where the hierachy output is appended to the end of the `.dot` file that your tool will produce)

```
   digraph ObjectModel {

     // The following 2 lines define the default NODE and EDGE
attributes
     node [shape="box", fontname="courier", fontsize="12"];
```

```
    edge [arrowhead="open", arrowsize="0.8", fontname="courier",
fontsize="12"];

    // Here are the actual nodes in the diagram...
    TheaterListing [shape="box", label="TheaterListing"];
    TheaterListing -> Object [arrowhead="empty"];
    Theater [shape="box", label="Theater"];
    Theater -> Object [arrowhead="empty"];
    Movie [shape="box", label="Movie"];
    Movie -> Object [arrowhead="empty"];
    Byte [shape="box", label="Byte"];
    Byte -> Number [arrowhead="empty"];
    Number [shape="parallelogram", label="Number"];
    Number -> Object [arrowhead="empty"];
    Object [shape="box", label="Object"];
    Movie -> Byte [label="thumbsUp",headlabel="!"];
    Theater -> Movie [label="movies",headlabel="+"];
    TheaterListing -> Theater [label="theaters",headlabel="+"];
  }

  // java.lang.Object
  //    java.lang.Number
  //       java.lang.Byte
  //    yourtests.Movie
  //    yourtests.Theater
  //    yourtests.TheaterListing
```

The last 6 lines above are new.

In contrast to what your program was doing before, now your program needs to have some data structures to model the input so that the output may be produced in the correct order, etc.

Checking In

Sometimes, Eclipse doesn't realize that a file has been changed and therefore will not commit your changes to the repository. To avoid this problem, make sure that you right-click on the topmost "ps3" item (in the left window), and click REFRESH. Then right click on it again, and click TEAM... then click COMMIT. You should be sure to check in at least the following files:

- `doc/report.html` with your answers written in the spaces provided; as well as all the `*.dot` files that you manually wrote or that your tool created.
- your code (ie, modifications to `ps3.ObjectModels`, and any other classes you create)

After you're done, you should run `validate6170 ps3` on Athena to make sure all your files are checked in and that your Java files are compilable.

Grading

The grading of this problem is separated into four categories.

- 15 points - for questions 1, 2, 3, and 4
- 60 points - for question 5, and for the design and construction of your object model extractor (needed for question 6 and 7)
- 25 points - for the questions about the object models your tool extracts (ie, question 6 in `doc/report.html`)

Hints

Don't Panic! You will probably feel overwhelmed by this problem set the first time you look at it. Some hints are available in the assignments section.

Errata

There are no known problems with the problem set thus far.

Q & A

This section will list clarifications and answers to common questions about problem sets. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the problem set handout and the specifications) when you have a problem.

Further Reading

If you liked this problem set, you may be interested to read a little bit more about program analysis. This reading is optional, and it will not help you with the problem set.
