Using Ant, JavaDoc, LOG4J, JUnit, and DocBook together

Ashley J.S Mills

<ug55axm@cs.bham.ac.uk>

Table of Contents

1. Introduction

This document is an attempt to unite the various programmer tools documented here [../tutorialshome.html]. The specific tools that will be used in the development of the program in this tutorial, will be:

- Ant [../ant/anthome.html]
- JavaDoc [../javadoc/javadochome.html]
- LOG4J [../log4j/log4jhome.html]
- JUnit [../junit/junithome.html]
- DocBook [../docbooksys/docbooksyshome.html]

It will be assumed that you have read these documents. The program itself will be a little game called "Identify" that I created based on my memories of a program on the Acorn BBC micros we had at my middle school. More information about this computer can be found at http://home.wanadoo.nl/jarod/museum/bbc.htm. The program constructs a binary tree which contains a `String` as it's root. It can have left and right sub-trees which are the same type of binary tree.

Each root (apart from the leaves of the tree) contains a yes/no question such as "Is it a bird?", this is presented to the user and the users response is received and stored. If the user responds with yes then the left subtree is evaluated, if the user responds with no then the right subtree is evaluated. Eventually the user will hit a leaf node, a tree with no children. This is an identification node which specifies an object in the form of a question such as "Is it a Vesper sparrow (Pooecetes gramineus)?". The answer to this question is special and if the user answers yes, the program will ask the user if they want to refine the identification. If the user answers no, the program will ask the user to specify what the object was that they were trying to identify, it will then ask them to specify a question that distinguishes between the object in the leaf node that failed to identify the object they were trying to identify and the object they were trying to identify, a transcript from use of the program should make this clearer:

1.1. An example run of the program

```
Is it living?
y
Is it a Tree?
```

```
n
What is your Object? (be as specific as possible)

a tiger

Please enter an question that could be used to distinguish between:

"Is it a Tree?"
and
"Is it a tiger?"

Try to make the distinction as high-level as possible.
Phrase the question so that answering yes to it will display:

"Is it a tiger?"

Is it an animal?

The computer will ask:
"Is it an animal?"

Responding yes will produce the question:
"Is it a Tiger"
Responding no will produce the question:
"Is it a Tree"

Is this behaviour satisfactory? y/n
y

Would you like to play again? y/n
y

Is it living?
y
Is it an animal?
y
Is it a tiger?
y
Is the question:

 "Is it a tiger?"

specific enough to pinpoint your object? y/n
n

Please enter a question that will be displayed after
the user answers yes to:

Is it a tiger?

For example, if the old question was "Is it fictional literature?"
and your object is a copy of "Romeo and Juliet" then
you could enter the question "Is it a Shakespeare play?"

Is it white with chocolate stripes and icy blue eyes?
What is your object?

a rare White Bengal Tiger (Panthera tigris tigris)

Give me an object I can use if somebody answers yes to
"Is it a tiger?"
but no to
"Is it white with chocolate stripes and icy blue eyes?"

a Siberian (Amur) tiger (Panthera tigris altaica)

Would you like to play again? y/n
y

Is it living?
y
Is it a tiger?
y
Is it white with chocolate stripes and icy blue eyes?
y
Is it a rare White Bengal Tiger (Panthera tigris tigris)?
y
Is the question:

  "Is it a rare White Bengal Tiger (Panthera tigris tigris)?"

specific enough to pinpoint your object? y/n
y

Correct identification!
```

```
        Would you like to play again? y/n
        n
```

How it does this is pretty straightforward, when it encounters a tree with empty children (a leaf) it knows that either the user said yes to the last identification question or said no. If the user said yes then the identification is correct and the program goes on to ask whether or not the definition was sufficiently accurate or not. If the user said no then the identification is incorrect and the program asks the user to define the object and provide a new, distinguishing question.

2. The Program

The program is composed of two components:

- `QuestionTree.java`
- `Identify.java`

2.1. `QuestionTree.java`

`QuestionTree.java` is an implementation of a binary tree that provides the underlying structure for the `Identify.java` program. The class is shown below and it can be downloaded here: QuestionTree.java [files/src/QuestionTree.java].

```
import org.apache.log4j.Logger;
/**
 * A question tree for a hierarchical identification model.
 *
 * Implemented as a binary tree, the root node is a <code>String</code>
 * representing a question.
 *
 * <p>
 * The left subtree is the tree that should be evaluated if the user
 * answers yes to the question held in the root of the parent tree.
 * </p>❶
 *
 * <p>
 * The right subtree is the tree that should be evaluated if the user
 * answers no to the question held in the root of the parent tree.
 * </p>
 *
 * @author  Ashley Mills
 * @version 0.1
 *
 */
public class QuestionTree {
   private Logger logger = Logger.getLogger(QuestionTree.class);
   private QuestionTree left, right;
   private String root;

   /**
    * Creates a new, empty <code>QuestionTree</code>.❷
    */
   public QuestionTree() {
      logger.debug("QuestionTree() called");
      root = null;
      left = null;
      right = null;
   }

   /**
    * Creates a new <code>QuestionTree</code> with a root question.
    *
    * @param String The root question.
    */
   public QuestionTree(String root) {
      logger.debug("QuestionTree(String root) called");
      this.root = root;
      left = new QuestionTree();
      right = new QuestionTree();
   }

   /**
    * Creates a new <code>QuestionTree</code> with a root question, a left
    * sub-<code>QuestionTree</code> and a right sub-<code>QuestionTree</code>.
    *
```

```
     * @param String The root question.
     * @param QuestionTree The <code>QuestionTree</code> that should become the left,
     * "yes branch" of the <code>QuestionTree</code>.
     *
     * @param QuestionTree The <code>QuestionTree</code> that should become the right,
     * "no branch" of the <code>QuestionTree</code>.
     */
    public QuestionTree(String root, QuestionTree left, QuestionTree right) {
       logger.debug("QuestionTree(String root, QuestionTree left, QuestionTree right) called");
       this.root = root;
       this.left = left;
       this.right = right;
    }

    /**
     * Sets the root question of the <code>QuestionTree</code> that the
     * method was called from.
     *
     * @param String The question that should become this <code>QuestionTree</code>s root question.
     */
    public void setRoot(String root) {
       logger.debug("setRoot(String root) called");
       this.root = root;
    }

    /**
     * Sets the left sub-<code>QuestionTree</code> of the <code>QuestionTree</code>.
     *
     * @param QuestionTree The <code>QuestionTree</code> that should become the left subtree.
     */
    public void setLeft(QuestionTree left) {
       logger.debug("setLeft(QuestionTree left) called");
       this.left = left;
    }

    /**
     * Sets the right sub-<code>QuestionTree</code> of the <code>QuestionTree</code>.
     *
     * @param QuestionTree The <code>QuestionTree</code> that should become the right subtree.
     */
    public void setRight(QuestionTree right) {
       logger.debug("setRight(QuestionTree right) called");
       this.right = right;
    }

    /**
     * Returns the <code>QuestionTree</code>s root question.
     *
     * @return String The question defined in the root of this tree.
     */
    public String getRoot() {
       logger.debug("getRoot() called");
       return root;
    }

    /**
     * Returns the left sub-<code>QuestionTree</code> of the <code>QuestionTree</code>.
     *
     * @return QuestionTree The left sub-<code>QuestionTree</code>.
     */
    public QuestionTree getLeft() {
       logger.debug("getLeft() called");
       return left;
    }

    /**
     * Returns the right sub-<code>QuestionTree</code> of the <code>QuestionTree</code>.
     *
     * @return QuestionTree The right sub-<code>QuestionTree</code>.
     */
    public QuestionTree getRight() {
       logger.debug("getRight() called");
       return right;
    }

    /**
     * Determines whether or not the <code>QuestionTree</code> is empty.
     *
     * @return boolean <code>true</code> if the <code>QuestionTree</code> is empty
     * and <code>false</code> otherwise.
     */
    public boolean isEmpty() {
       logger.debug("isEmpty() called");
       return root==null;
    }
```

```
}
```

❶

```
/**
 * A question tree for a hierarchical identification model.
 *
 * <p>
 * Implemented as a binary tree, the root node is a <code>String</code>
 * representing a question.
 * </p>
 *
 * <p>
 * The left subtree is the tree that should be evaluated if the user
 * answers yes to the question held in the root of the parent tree.
 * </p>
 *
 * <p>
 * The right subtree is the tree that should be evaluated if the user
 * answers no to the question held in the root of the parent tree.
 * </p>
 *
 * @author  Ashley Mills
 * @version 0.1
 *
 */
```

`QuestionTree.java` contains JavaDoc for every method and constructor, this is the JavaDoc for the entire class, notice the use of HTML and the one line summary to start with. The author specified is "Ashley Mills" and the version is "0.1". The JavaDoc output for this section can be seen below:

Figure 1. The JavaDoc output for the listing above

Class QuestionTree

java.lang.Object
  |
  +--**QuestionTree**

---

public class **QuestionTree**
extends Object

A question tree for a heirarchical identification model.

Implemented as a binary tree, the root node is a `String` representing a question.

The left subtree is the tree that should be evaluated if the user answers yes to the question held in the root of the parent tree.

The right subtree is the tree that should be evaluated if the user answers no to the question held in the root of the parent tree.

---

❷

```
public class QuestionTree {
   private Logger logger = Logger.getLogger(QuestionTree.class);
   private QuestionTree left, right;
   private String root;

   /**
    * Creates a new, empty <code>QuestionTree</code>.
    */
   public QuestionTree() {
      logger.debug("QuestionTree() called");
      root = null;
      left = null;
      right = null;
   }
```

Notice that the logger is instantiated using the name of the class `QuestionTree.class`. It would be inefficient to create a new logger every time the constructors were called so `QuestionTree.java` inherits its logger from whichever class instantiates it. A logger statement can be seen in the QuestionTree constructor:

```
logger.debug("QuestionTree() called");
```

The QuestionTree contains many logger statements, one for every method and constructor. The methods and constructors are called many times during the execution of `Identify.class` so the *Level DEBUG* was chosen instead of *INFO*, which is usually used to produce messages indicating the entering and/or leaving of methods, to reduce the number of messages being produced. The logger level is set to *INFO* in the configuration file.

It is assumed that a basic knowledge of computer data structures is known so the program code should be self-documenting. For those that need the relevant background information on binary trees, I suggest you read the binary tree handouts produced by Martin Escardo for the *Introduction To Computer Science - B* first year course at The University Of Birmingham, they can be found at http://www.cs.bham.ac.uk/~mhe/introductionb.html#ref:lectures.

2.2. `Identify.java`

`Identify.java` provides the part of the program that gets input from the user and constructs the QuestionTree, it is too big to be shown in it's entirety here so only the *main* method is shown, the auxiliary functions will be shown and discussed separately later. The program can be downloaded here: Identify.java [files/src/Identify.java].

```
/** A program which models a hierarchical question tree which is
 * extensible, it's purpose is to identify objects through a series
 * of yes/no questions.
 *
 * @author  Ashley Mills
 * @version 0.1
 */
import java.io.*;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
public class Identify {
   private static BufferedReader reader;
   private static Logger logger = Logger.getLogger(Identify.class);
   private static QuestionTree tree;
   public static void main(String[] args) {❶
      DOMConfigurator.configure("log4jconfig.xml");
      logger.info("Entering main");
      reader = new BufferedReader(new InputStreamReader(System.in));

      logger.info("Constructing initial QuestionTree");
      tree = new QuestionTree("Is it living?",
                new QuestionTree("Is it a tree?"),❷
                new QuestionTree("Is it a computer?")
             );

      QuestionTree currentTree = tree;
      boolean correctIdentification = false,
      finished = false, stopped = false, yes;

      logger.debug("Starting outer while loop...");
      while(!stopped) {

         output("");
         logger.debug("Transversing tree...");
         while(!finished) {
            output(currentTree.getRoot());
            yes = yesOrNo();
            logger.debug("yes = " + yes);

            correctIdentification = false;
            if(yes) {
               correctIdentification = true;
               if(currentTree.getLeft().isEmpty()) finished=true;
               else currentTree = currentTree.getLeft();
               logger.debug("finished: "+finished);
            } else {
               if(currentTree.getRight().isEmpty()) finished=true;
               else currentTree = currentTree.getRight();❸
            }
         }

         if(correctIdentification) {
```

```
            output("Is the question:\n");
            output("  \""+currentTree.getRoot()+"\"\n");
            output("specific enough to pinpoint your object? y/n");

            yes = yesOrNo();
            if(yes) {
               output("\nCorrect identification!");
            } else {
               refineLeafQuestion(currentTree);
            }
         } else {
            addNewObjectToTree(currentTree);
         }

         output("\nWould you like to play again? y/n");
         yes = yesOrNo();
         if(yes) {
            currentTree  = tree;
            correctIdentification = false;
            finished     = false;
         } else stopped  = true;
      }
      logger.info("Leaving main");
   }
```

❶

```
/** A program which models a hierarchical question tree which is
 * extensible, it's purpose is to identify objects through a series
 * of yes/no questions.
 *
 * @author  Ashley Mills
 * @version 0.1
 */
import java.io.*;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
public class Identify {
   private static BufferedReader reader;
   private static Logger logger = Logger.getLogger(Identify.class);
   private static QuestionTree tree;
   public static void main(String[] args) {
      DOMConfigurator.configure("log4jconfig.xml");
      logger.info("Entering main");
      reader = new BufferedReader(new InputStreamReader(System.in));
```

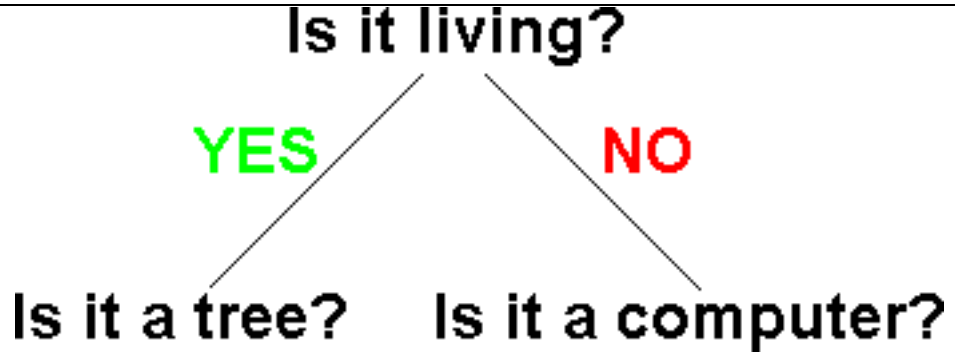
Some JavaDoc is present at the top of the program, this will not be used by any external programs as JavaDoc will not be generated for this class since it contains no public methods. The JavaDoc shown is just for persons inspecting the program code. The *import* statements import the classes used in the program. A BufferedReader is declared, this will be used to get input from the user. A logger is created with the name of the class and the QuestionTree is declared so it can be accessed by the whole class. The *main* method sees the configuration of the logger via a *DOMConfigurator*, this allows the use of an external configuration file for the logger, marked up in XML. There are also a couple of logger statements present.

❷

```
      logger.info("Constructing initial QuestionTree");
      tree = new QuestionTree("Is it living?",
                 new QuestionTree("Is it a tree?"),
                 new QuestionTree("Is it a computer?")
             );
```

There is a log message, followed by the construction of a QuestionTree, this is the QuestionTree that will always point to the root tree of the whole QuestionTree. The tree construction section defines a QuestionTree with the root question "Is it living? When this is used later on, if the user answers yes to the question "Is it living?", the user will be asked the question "Is it a Tree", if the user answers no to the question "Is it living?", the user will be asked the question "Is it a Computer?". Here is a diagram, which illustrating this:

Figure 2. The initial QuestionTree



❸

```
QuestionTree currentTree = tree;
boolean correctIdentification = false,
finished = false, stopped = false, yes;

logger.debug("Starting outer while loop...");
while(!stopped) {

   output("");
   logger.debug("Transversing tree...");
   while(!finished) {
      output(currentTree.getRoot());
      yes = yesOrNo();
      logger.debug("yes = " + yes);

      correctIdentification = false;
      if(yes) {
         correctIdentification = true;
         if(currentTree.getLeft().isEmpty()) finished=true;
         else currentTree = currentTree.getLeft();
         logger.debug("finished: "+finished);
      } else {
         if(currentTree.getRight().isEmpty()) finished=true;
         else currentTree = currentTree.getRight();
      }
   }

   if(correctIdentification) {
      output("Is the question:\n");
      output("  \""+currentTree.getRoot()+"\"\n");
      output("specific enough to pinpoint your object? y/n");

      yes = yesOrNo();
      if(yes) {
         output("\nCorrect identification!");
      } else {
         refineLeafQuestion(currentTree);
      }
   } else {
      addNewObjectToTree(currentTree);
   }

   output("\nWould you like to play again? y/n");
   yes = yesOrNo();
   if(yes) {
      currentTree  = tree;
      correctIdentification = false;
      finished     = false;
   } else stopped  = true;
}
```

This is where the construction and modification of the QuestionTree occurs the first thing that happens is some variables are initialised:

```
QuestionTree currentTree = tree;
boolean correctIdentification = false,
finished = false, stopped = false, yes;
```

A new QuestionTree is created that is a pointer to the root node of the overall tree, it is called *currentTree*, *currentTree* will vary as the tree is transversed so that *currentTree* always points to the last question asked. The other variables initialised are *boolean*, and control things like loops and whether certain paths of the program are executed. Below the initialisation section is a large loop controlled via the *stopped* variable:

```
while(!stopped) {
   .
   .
   .
   output("\nWould you like to play again? y/n");
   yes = yesOrNo();
   if(yes) {
      currentTree  = tree;
      correctIdentification = false;
      finished     = false;
   } else stopped  = true;
}
```

When the user enters no to the question "Would you like to play again?", *stopped* is set to *true* causing the outer loop to be be broken and subsequently, the program to terminate. The contents of this outer loop can be broken up into two sections, the first controls the transversal of the QuestionTree and the second controls the modification of the tree upon reaching a leaf, here is the first section:

```
   output("");
   logger.debug("Transversing tree...");
   while(!finished) {
      output(currentTree.getRoot());
      yes = yesOrNo();
      logger.debug("yes = " + yes);

      correctIdentification = false;
      if(yes) {
         correctIdentification = true;
         if(currentTree.getLeft().isEmpty()) finished=true;
         else currentTree = currentTree.getLeft();
         logger.debug("finished: "+finished);
      } else {
         if(currentTree.getRight().isEmpty()) finished=true;
         else currentTree = currentTree.getRight();
      }
   }
```

The first line outputs an empty line for formatting reasons. The while loop is controlled by the variable *finished* which was set to *false* in the initialisation section described above. The loop prints out the root of the current tree, which is a question, and gets the users response to this question via the *yesOrNo()* method. The boolean variable *correctIdentification* is set to *false* and the users response to the printed question is evaluated.

If the user said yes to the printed question then *correctIdentification* is set to *true*, this is incase *currentTree* happens to be a leaf, whereupon if the user says yes then the object they are looking for has been correctly identified (as much as is possible anyway). A "yes" answer requires that the transversal of the tree continues at the left subtree but first the left subtree is checked for emptiness. If the left subtree is empty then *currentTree* is a leaf node and transversal must stop, so *finished* is set to *true*. If the left subtree it is not empty then *currentTree* is set to the left subtree.

If the user said no to the printed question then the right subtree must be evaluated, but first the right subtree is checked for emptiness, because if it is empty it is a leaf node and transversal of the tree must stop, *finished* is set to *true* to accomplish this. If the right subtree is not empty then *currentTree* is set to the left subtree.

The loop goes round and round (that is generally what loops do ;)), transversing the question tree in response to the users answers to the questions printed. Eventually a leaf node is encountered, transversal of the tree terminates and program control flows to the next section:

```
 if(correctIdentification) {
   output("Is the question:\n");
   output("  \""+currentTree.getRoot()+"\"\n");
   output("specific enough to pinpoint your object? y/n");

   yes = yesOrNo();
   if(yes) {
      output("\nCorrect identification!");
   } else {
      refineLeafQuestion(currentTree);
   }
} else {
   addNewObjectToTree(currentTree);
}
```

If the item was correctly identified (if the user said yes to a question that was leaf), *correctIdentification* will be *true*. The user is asked whether the question they answered yes to was specific enough to pinpoint the object they were trying to identify. If the user says that it was specific enough, by answering yes, the program outputs "Correct identification!", if the user says no then *refineLeafQuestion(currentTree)* is called to refine the question. For an example of this see the section entitled An example run of the program.

If the item was not correctly identified (if the user said no to a question that was a leaf) *correctIdentification* will be *false* and the alternative of the if..then..else structure will be executed. The method *addNewObjectToTre(currentTree)* is called to add a new object to the tree since the object the user was trying to identify was not found.

Let's take a look at the *refineLeafQuestion* method which is called when an object has been identified as much as possible (down to a leaf node) but the identification is not specific enough:

```
private static void refineLeafQuestion(QuestionTree currentTree) {
   logger.info("Entering refineLeafQuestion(QuestionTree currentTree)");
   output("\nPlease enter a question that will be displayed after");
   output("the user answers yes to:\n");

   output(currentTree.getRoot()+"\n");

   output("For example, if the old question was \"Is it fictional literature?\"");
   output("and your object is a copy of \"Romeo and Juliet\" then you");
   output("could enter the question \"Is it a Shakespeare play?\"\n");

   String usersQuestion = getString();

   output("\nWhat is your object?\n");

   String usersYesObject = getString();

   output("\nGive me an object I can use if somebody answers yes to");
   output("\""+currentTree.getRoot()+"\"");
   output("but no to");
   output("\""+usersQuestion+"\"\n");

   String usersNoObject = getString();

   currentTree.setLeft(
      new QuestionTree(usersQuestion,
         new QuestionTree("Is it "+usersYesObject+"?"),
         new QuestionTree("Is it "+usersNoObject+"?")
      )
   );
   logger.info("Leaving refineLeafQuestion(QuestionTree currentTree)");
}
```

The output statements make it pretty clear what is going on, essentially this method can be summarised to:

```
   String usersQuestion  = getString();
   String usersYesObject = getString();
   String usersNoObject  = getString();
   currentTree.setLeft(
      new QuestionTree(usersQuestion,
         new QuestionTree("Is it "+usersYesObject+"?"),
         new QuestionTree("Is it "+usersNoObject+"?")
      )
   );
```

A new question is got from the user which will become the left subtree of the question that was not specific enough to identify the object the user was looking for. So if the not-specific-enough question was "Is it a book?", and the users object is Rowan-"Robinson Cosmology SECOND EDITION" the user might specify the new question "Is it a scientific book?", the user is then asked to specify the object, assume that the user specifies "Rowan-Robinson Cosmology SECOND EDITION". The user is then asked to specify an alternative to the object they just specified that could be used as the no-branch for the question they just specified. The object has to be in context, since this tree will become the left subtree of the tree with the question "Is it a book?". In this case the user would want to specify some kind of non-scientific book, for example the children's book "Enid Blyton - The Enchanted Wood". After the user has provided the necessary details, the left subtree is set to a new QuestionTree with the users question as root, the users object as the yes-branch and the users alternative object as the no-branch, the diagrams below illustrate this:

Figure 3. Part of the QuestionTree before refinement

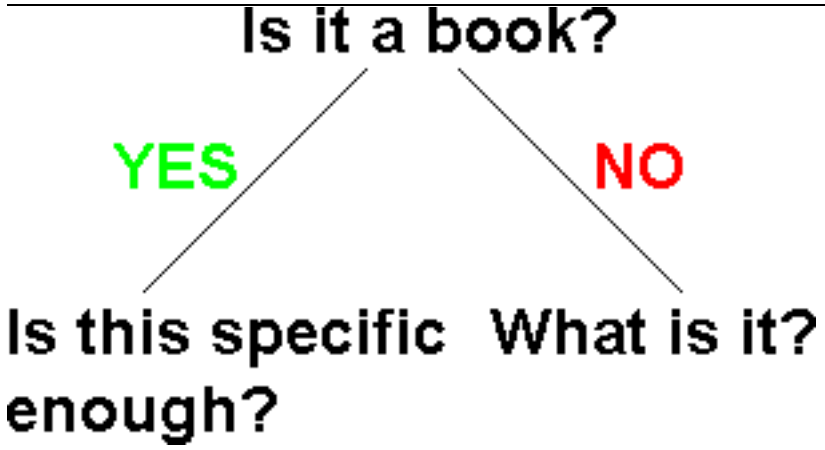
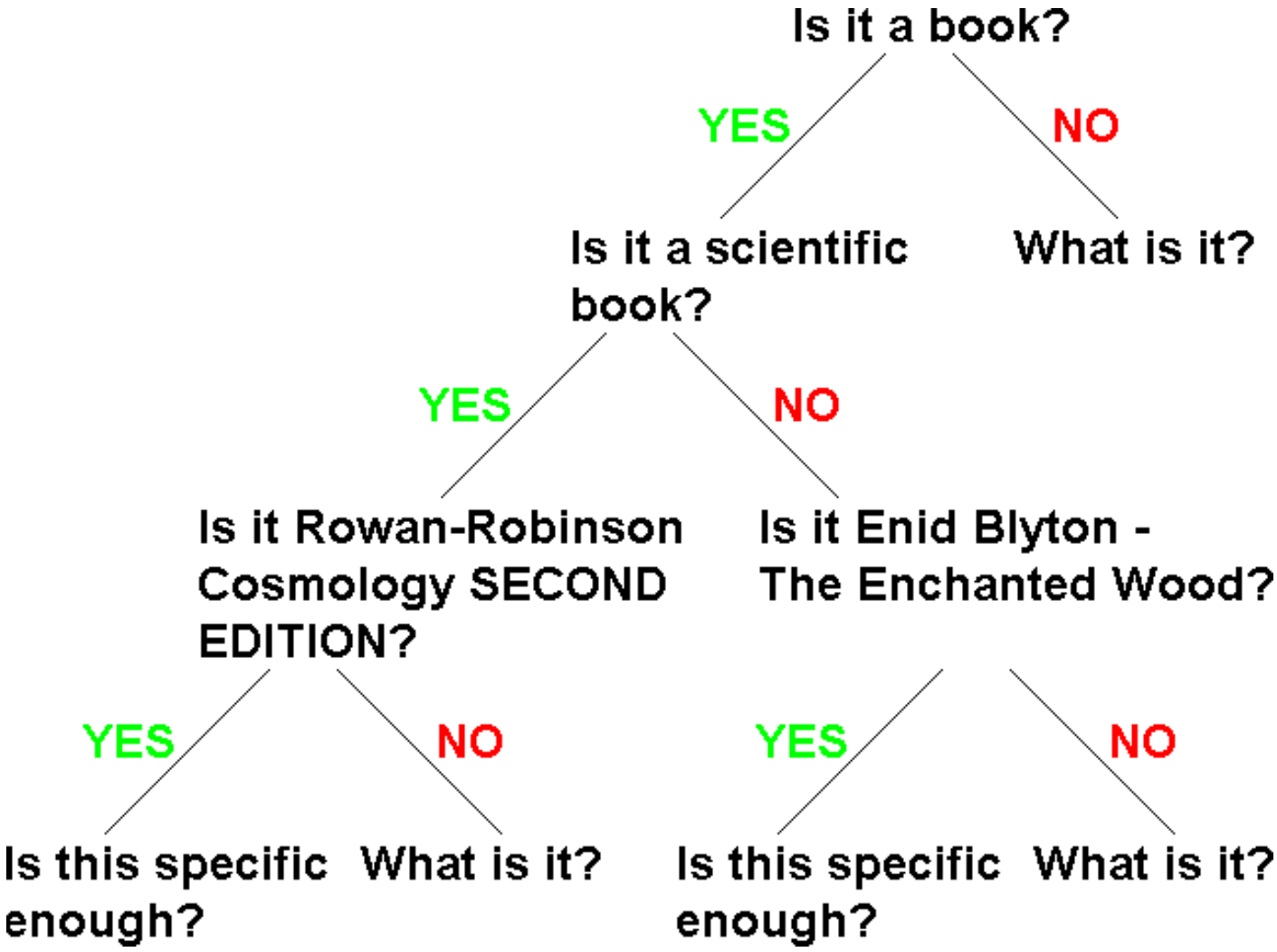


Figure 4. The same part of the QuestionTree after refinement



Let's take a look at the *addNewObjectToTree* method which is called when the tree has been transversed to a leaf node but the identification of the users object has failed. (the user said no to a question in a leaf node):

```
private static void addNewObjectToTree(QuestionTree currentTree) {
   logger.info("Entering addNewObjectToTree(QuestionTree currentTree)");

   String newQuestion = ""; boolean goodAnswer = false;

   output("What is your object?\n");

   String usersObject = getString();
```

```
      while(!goodAnswer) {
         output("\nPlease enter a question that could be used to distinguish between:\n");

         output("\"" + currentTree.getRoot() + "\"");
         output("and");
         output("\"Is it " + usersObject + "?\"\n" );

         output("Try to make the distinction as high-level as possible.");
         output("Phrase the question so that answering yes to it will display:\n");

         output("\"Is it " + usersObject + "?\"\n");

         newQuestion = getString();

         output("\nThe computer will ask:");
         output("\""+newQuestion+"\"\n");

         output("Responding yes will produce the question:");
         output("\"Is it " + usersObject + "\"");
         output("Responding no will produce the question:");
         output("\"" + currentTree.getRoot() + "\"\n");

         output("Is this behaviour satisfactory? y/n");

         goodAnswer = yesOrNo();
      }

      String temp = currentTree.getRoot();
      currentTree.setRoot(newQuestion);
      currentTree.setLeft(new QuestionTree("Is it "+usersObject+"?"));
      currentTree.setRight(new QuestionTree(temp));
      logger.info("Leaving addNewObjectToTree(QuestionTree currentTree)");
   }
```

This can be summarised to:

```
      boolean goodAnswer = false;
      String usersObject = getString();
      while(!goodAnswer) {
         newQuestion = getString();
         output("Is this behaviour satisfactory? y/n");
         goodAnswer = yesOrNo();
      }

      String temp = currentTree.getRoot();
      currentTree.setRoot(newQuestion);
      currentTree.setLeft(new QuestionTree("Is it "+usersObject+"?"));
      currentTree.setRight(new QuestionTree(temp));
```

goodAnswer is set to *false* so that the while loop will be processed, the user is asked to specify the object they are trying to identify and this is stored in the variable *usersObject*. The loop is entered, which asks the user for a question that should distinguish between the object the user is trying to identify and the false identification that the QuestionTree provided. The new state of the tree is displayed and the user is asked if the behaviour is satisfactory, the users response to this question is used to set the *goodAnswer* variable which controls the loop.

After the user enters and confirms a question the question that failed to identify the object is replaced with the new question the user provided and the object the user provided becomes the left subtree of this question, the right subtree is set to the original question which failed to identify the object the user was trying to identify. For example, the original question could have been "Is it a cat?" and perhaps the user was trying to identify "a dog", the user could provide the new question "Is it known as mans best friend?" this would replace "Is it a cat?", the yes-branch would become "Is it a dog?" and the no-branch would become "Is it a cat?, the diagrams below illustrate this:

Figure 5. Part of the QuestionTree before adding a new question

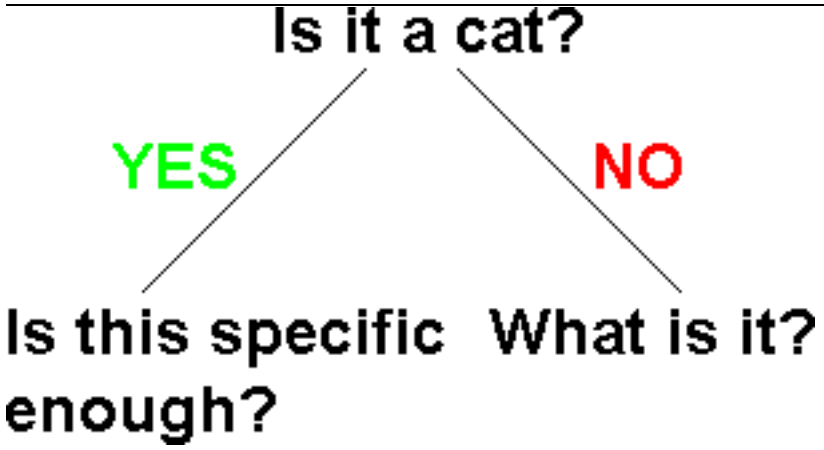
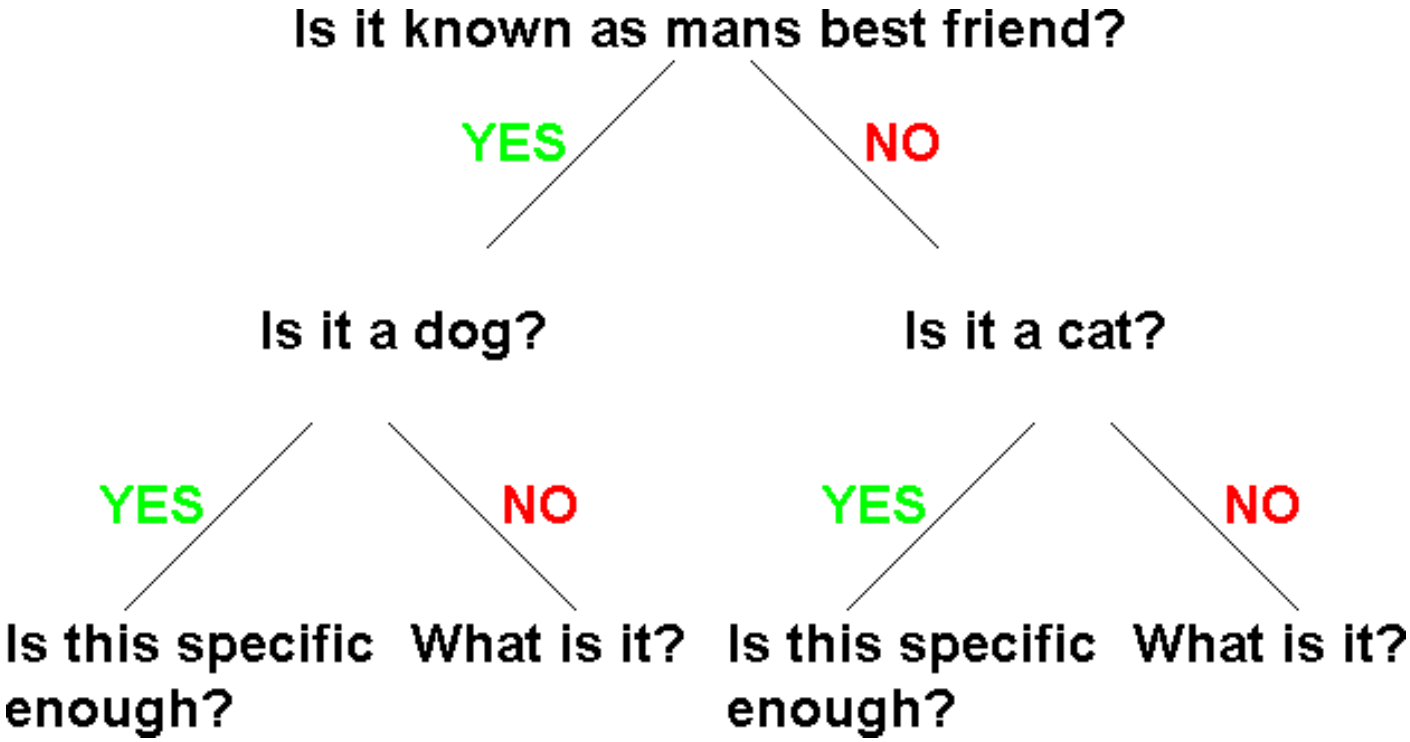


Figure 6. The same part of the QuestionTree after adding a new question



The rest of the methods in the program are auxiliary methods like *output* which outputs strings and *getString* which gets a string from the user, these need no explaining, examine the source code for more information.

3. The Test Harness

Since the QuestionTree is the underlying data structure that the program works with, it is essential that this part works exactly as is desired hence a test harness was created for it using JUnit. The test program is shown below:

```
import junit.framework.*;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
public class QuestionTreeTest extends TestCase {
   private Logger logger = Logger.getLogger(QuestionTreeTest.class);
   private QuestionTree tree1, tree2, tree3;

   public QuestionTreeTest(String name) {
      super(name);
      DOMConfigurator.configure("log4jconfig2.xml");
   }

   protected void setUp() {
      logger.info("Entering setUp()");
      tree1 = new QuestionTree("Is it living?",
```

```
                new QuestionTree("Is it a Wolf?"),
                new QuestionTree("Is it a brick?")
             );
      tree2 = new QuestionTree("Is it living?");
      logger.info("Leaving setUp()");
   }

   public void testConstructor() {
      logger.info("Entering testConstructor()");
      QuestionTree tree3 = new QuestionTree("Is it living?");
      assertEquals("Is it living?", tree3.getRoot());
      logger.info("Leaving testConstructor()");
   }

   public void testGetRoot() {
      logger.info("Entering testGetRoot()");
      assertEquals("Is it living?", tree2.getRoot());
      logger.info("Leaving testGetRoot()");
   }

   public void testGetLeft() {
      logger.info("Entering testGetLeft()");
      assertEquals("Is it a Wolf?", tree1.getLeft().getRoot());
      logger.info("Leaving testGetLeft()");
   }

   public void testGetRight() {
      logger.info("Entering testGetRight()");
      assertEquals("Is it a brick?", tree1.getRight().getRoot());
      logger.info("Leaving testGetRight()");
   }

   public void testSetRoot() {
      logger.info("Entering testSetRoot()");
      tree1.setRoot("Is it alive?");
      assertEquals("Is it alive?", tree1.getRoot());
      logger.info("Leaving testSetRoot()");
   }

   public void testSetLeft() {
      logger.info("Entering testSetLeft()");
      tree1.setLeft(new QuestionTree("Is it a Dog?"));
      assertEquals("Is it a Dog?", tree1.getLeft().getRoot());
      logger.info("Leaving testSetLeft()");
   }

   public void testSetRight() {
      logger.info("Entering testSetRight()");
      tree1.setRight(new QuestionTree("Is it a concrete block?"));
      assertEquals("Is it a concrete block?", tree1.getRight().getRoot());
      logger.info("Leaving testSetRight()");
   }

   public void testIsEmpty() {
      logger.info("Entering testIsEmpty()");
      assertEquals(true, tree2.getLeft().isEmpty());
      assertEquals(true, tree2.getRight().isEmpty());
      logger.info("Leaving testIsEmpty()");
   }
}
```

*junit.framework.** is imported so that the class can extend *TestCase*. *org.apache.log4j.Logger* and *org.apache.log4j.xml.DOMConfigurator* are imported so that a logger can be setup. The logger is setup in the *QuestionTreeTest* constructor of the class. A different logfile is used for logging than the one used for the logging from `Identify.java`. The configuration file for the logger is `log4jconfig2.xml`. The logger has a priority of *DEBUG*.

All the methods of `QuestionTree.java` are tested, they appear to be tested in a particular order but JUnit uses reflection and cannot guarantee the order that tests are executed in hence the program just displays logical grouping. The constructors are tested first (implicitly in the *setUp* method, and via *constructorTest*) followed by the *get* methods, the *set* methods and finally the *isEmpty* method is tested. The following *junit.framework.assert* methods are used:

- public static void assertEquals(boolean expected, boolean actual)

  To test the *isEmpty()* method.

- public static void assertEquals(java.lang.Object expected, java.lang.Object actual)

  To test all the rest of the methods

It is possible to use *assertTrue(boolean condition)* to test the strings for equality in the program's tests instead of *assertEquals(java.lang.Object expected, java.lang.Object actual)*. The latter can be used to compare any two objects that implement the *equals* method. The former does exactly the same thing as the latter since it actually uses the *equals* method itself.

The tests will be integrated into an Ant script later but one can manually run the tests by issuing these commands:

```
java junit.textui.TestRunner QuestionTreeTest
```

The output produced from this is:

```
.........
Time: 0.2

OK (8 tests)
```

It is interesting to note here that before the logging statements and code were implemented into this class, the output from running the test was:

```
.log4j:WARN No appenders could be found for logger (QuestionTree.class).
log4j:WARN Please initialize the log4j system properly.
........
Time: 0.14

OK (8 tests)
```

The log4j warnings occur because there was no logger for `QuestionTree.java` to inherit. Usually `QuestionTree.java` inherits and uses the logger created by QuestionTreeTest. The time for the log-free test is about (it varies slightly) 0.06 seconds less than it is with loggers added. This is an indication of the kind of performance hit one gets when using the kind of logger described in `log4jconfig2.xml` and the kind of logging system used here. For more information about the loggers used by the test harness, including description of the output, see *`QuestionTreeTest.java`'s Logger Configuration File*.

4. The LOG4J Loggers

There are two Log4J loggers used in this program, one is used by the program `Identify.java` and the other is used by the program `QuestionTreeTest.java`. `QuestionTree.java` inherits the logger from the class it is instantiated from. Both loggers are instantiated in the same manner; the logger is declared globally in the class and named after the class it is called from then the logger is configured via an external XML file, referenced by *DOMConfigurator*.

4.1. `Identify.java`'s Logger Configuration File

The configuration file used by `Identify.java` can be downloaded here: log4jconfig.xml [files/src/log4jconfig.xml], it is shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="appender" class="org.apache.log4j.FileAppender">
    <param name="File" value="Identify.log"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="[%d{HH:mm:ss}] [%-12C] [%p] - %m%n"/>
    </layout>
  </appender>

  <root>
    <priority value ="info"/>
    <appender-ref ref="appender"/>
  </root>

</log4j:configuration>
```

A *FileAppender* is created that outputs directly (non-appending) to the file `Identify.log`. The *Layout* is set as *PatternLayout* and the *ConversionPattern* is setup so that messages are output like this:

```
[13:07:15] [Identify    ] [INFO] - Constructing initial QuestionTree
```

The fields are "%d{HH:mm:ss}" for the date in the format shown above, "%-12C" to output the fully qualified classname, right-padded to 12 characters, "%p" outputs the logger priority (Level), "%m" outputs the logging message and "%n" outputs a newline. The priority is set to *INFO*

A typical log of running the *Identify* program is shown below:

```
[13:36:05] [Identify    ] [INFO] - Entering main
[13:36:05] [Identify    ] [INFO] - Constructing initial QuestionTree
[13:36:10] [Identify    ] [INFO] - Entering refineLeafQuestion(QuestionTree currentTree)
[13:36:38] [Identify    ] [INFO] - Leaving refineLeafQuestion(QuestionTree currentTree)
[13:37:02] [Identify    ] [INFO] - Entering addNewObjectToTree(QuestionTree currentTree)
[13:37:57] [Identify    ] [INFO] - Leaving addNewObjectToTree(QuestionTree currentTree)
[13:38:06] [Identify    ] [INFO] - Leaving main
```

If the *Level* is set to *DEBUG*, the amount of messages quickly becomes large so an example will not be shown here, instead, one may be downloaded from here IdentifyDebug.log [files/IdentifyDebug.log].

4.2. `QuestionTreeTest.java`'s Logger Configuration File

The configuration file used by `Identify.java` can be downloaded here: log4jconfig2.xml [files/src/log4jconfig2.xml], it is shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="appender" class="org.apache.log4j.FileAppender">
    <param name="File" value="qtreetest.log"/>
    <param name="Append" value="false"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="[%d{HH:mm:ss}] [%-16C] [%-5p] - %m%n"/>
    </layout>
  </appender>

  <root>
    <priority value ="debug"/>
    <appender-ref ref="appender"/>
  </root>

</log4j:configuration>
```

A *FileAppender* is created that outputs directly (non-appending) to the file `qtreetest.log`. The *Layout* is set as *PatternLayout* and the *ConversionPattern* is setup so that messages are output like this:

```
[13:56:49] [QuestionTreeTest] [INFO ] - Entering setUp()
```

At first sight, it looks identical to the output produced using `log4jconfig.xml`, but there is one difference, instead of using "%-14C" to output the fully qualified classname right-padded to 12 characters, "%-16C" is used to output the fully qualified classname right-padded to 16 characters because the classnames in use are a little bit longer. The output log produced when running `QuestionTreeTest.java` as a JUnit set of tests is quite long so will not be shown in this document, instead, it can be downloaded here: qtreetest.log [files/qtreetest.log].

The output produced if the *Level* is changed to *INFO* is not that long and is shown below:

```
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testConstructor()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testConstructor()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testGetRoot()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testGetRoot()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testGetLeft()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testGetLeft()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testGetRight()
```

```
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testGetRight()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testSetRoot()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testSetRoot()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testSetLeft()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testSetLeft()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testSetRight()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testSetRight()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving setUp()
[14:10:36] [QuestionTreeTest] [INFO ] - Entering testIsEmpty()
[14:10:36] [QuestionTreeTest] [INFO ] - Leaving testIsEmpty()
```

Notice that the method *setUp* in `QuestionTreeTest.java` is called before every test, if *tearDown* had been used, this would be called after every test.

5. The JavaDoc

The JavaDoc is produced by the Ant buildfile discussed in the section entitled *The Ant Buildfile*. The JavaDoc can be viewed online at index.html [files/build/javadoc/index.html].

6. The Documentation in XML DocBook

The DocBook documentation for the Identify program can be found here: Identify Documentation [files/identifyhome.html]. The XML file from which the HTML and PDF versions are created from is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
"http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
<article id="Identify">
  <articleinfo>
    <title>Identify</title>
    <author>
      <firstname>Ashley</firstname>
      <surname>Mills</surname>
      <affiliation>
        <address><email>ug55axm@cs.bham.ac.uk</email></address>
      </affiliation>
    </author>

    <copyright>
      <year>2002</year>
      <holder role="mailto:ug55axm@cs.bham.ac.uk">The University Of Birmingham</holder>
    </copyright>
  </articleinfo>

  <sect1 id="Identify-Description"><title>Think Of An Object!</title>
    <para>
      The Identify program builds a binary tree of questions that is transversed according to a users
    </para>

    <para>
      The implementation consists of a separate class to provide the special binary tree, called <file
    </para>

    <figure><title>An example of a QuestionTree</title>
      <mediaobject>
        <imageobject><imagedata fileref="qtreeexample.png" format="PNG"/></imageobject>
      </mediaobject>
    </figure>

    <para>It can be constructed like this:</para>

    <programlisting>
      QuestionTree tree = new QuestionTree(

         "Is it living?",

         new QuestionTree("Does it live in the sea?",
            new QuestionTree("Is it a whale?"),
            new QuestionTree("Is it a tiger?")
         ),
```

```
        new QuestionTree("Is it electrical hardware?",
           new QuestionTree("Is it a computer?"),
           new QuestionTree("Is it a diamond?")
        )
      );
    </programlisting>

    <para>
      The <acronym>API</acronym> for QuestionTree is available from here: <ulink url="../javadoc/index
    </para>

    <screen>
      <userinput><command>java</command> Identify</userinput>
    </screen>
  </sect1>
</article>
```

The file starts with an *articleinfo* section which declares whom the author is, the copyright and contact information. The article only has one section, which comes next and contains a little bit of information about the Identify program including an example QuestionTree in pictorial form and the source code used to create the QuestionTree. The document ends by illustrating how the program is executed.

7. The Ant Buildfile

The Ant buildfile used to build the whole project can be downloaded here: build.xml [files/build.xml]. The buildfile is also shown below:

```
<?xml version="1.0"?>
<project name="Identify" default="all" basedir=".">
  <property name="src"   value="src"/>
  <property name="build" value="build"/>

  <target name="all" depends="clean, mkdir, program, javadoc, docbook"/>

  <target name="program" description="Compiles the program source">
    <javac srcdir="${src}" destdir="${build}/bin"/>
  </target>

  <target name="javadoc" description="Creates the JavaDoc output">
    <javadoc sourcefiles="${src}/QuestionTree.java" destdir="${build}/javadoc"/>
  </target>

  <target name="docbook" description="Creates the documentation">

    <exec executable="xsltproc">
      <arg line="-o ${build}/documentation/Identify.html"/>
      <arg line="- -stringparam generate.toc 'article nop'"/>
      <arg line="file:///c:/lib/stylesheets/xhtml/customxhtml.xsl"/>
      <arg line="${src}/Identify.xml"/>
    </exec>

    <exec executable="xsltproc">
      <arg line="-o ${src}/Identify.fo"/>
      <arg line="- -stringparam generate.toc 'article nop'"/>
      <arg line="file:///c:/lib/stylesheets/fo/customfo.xsl"/>
      <arg line="${src}/Identify.xml"/>
    </exec>

    <exec executable="fop">
      <arg line="${src}/Identify.fo"/>
      <arg line="${build}/documentation/Identify.pdf"/>
    </exec>

    <delete file="${src}/Identify.fo"/>
    <copy file="${src}/Identify.xml"
          tofile="${build}/documentation/Identify.xml"/>
    <copy file="${src}/qtreeexample.png"
          tofile="${build}/documentation/qtreeexample.png"/>
  </target>

  <target name="clean" description="Cleans the whole project">
    <delete dir="${build}"/>
  </target>

  <target name="mkdir" description="Creates the required output directories">
    <mkdir dir="${build}/bin"/>
    <mkdir dir="${build}/javadoc"/>
    <mkdir dir="${build}/documentation"/>
  </target>
</project>
```

The default target is all, which looks like this:

```
<target name="all" depends="clean, mkdir, program, javadoc, docbook"/>
```

First, the target, *clean* is called:

```
  <target name="clean" description="Cleans the whole project">
    <delete dir="${build}"/>
  </target>
```

This target cleans out the last build by deleting the build directory. Next in the list of depends for the *all* target is *mkdir*:

```
  <target name="mkdir" description="Creates the required output directories">
    <mkdir dir="${build}/bin"/>
    <mkdir dir="${build}/javadoc"/>
    <mkdir dir="${build}/documentation"/>
  </target>
```

This creates the output directories, notice that the directory `build` is never explicitly created, this is because Ant will create any non-existent parent directories, for example:

```
<mkdir dir="dir1/dir2/dir3/dir4/dir5"/>
```

Would create the first four directories before creating the 5th directory. Next in the list of depends for the *all* target is *program*:

```
  <target name="program" description="Compiles the program source">
    <javac srcdir="${src}" destdir="${build}/bin"/>
  </target>
```

This compiles the program source code using the *javac* task. Next in the list of depends for the *all* target is *javadoc*:

```
  <target name="javadoc" description="Creates the JavaDoc output">
    <javadoc sourcefiles="${src}/QuestionTree.java" destdir="${build}/javadoc"/>
  </target>
```

This creates the JavaDoc output by using Ant's builtin *javadoc* task. Next, and last, in the list of depends for the *all* target is *docbook*:

```
<target name="docbook" description="Creates the documentation">

  <exec executable="xsltproc">
    <arg line="-o ${build}/documentation/Identify.html"/>
    <arg line="- -stringparam generate.toc 'article nop'"/>
    <arg line="file:///c:/lib/stylesheets/xhtml/customxhtml.xsl"/>
    <arg line="${src}/Identify.xml"/>
  </exec>

  <exec executable="xsltproc">
    <arg line="-o ${src}/Identify.fo"/>
    <arg line="- -stringparam generate.toc 'article nop'"/>
    <arg line="file:///c:/lib/stylesheets/fo/customfo.xsl"/>
    <arg line="${src}/Identify.xml"/>
  </exec>

  <exec executable="fop">
    <arg line="${src}/Identify.fo"/>
    <arg line="${build}/documentation/Identify.pdf"/>
  </exec>

  <delete file="${src}/Identify.fo"/>
  <copy file="${src}/Identify.xml"
        tofile="${build}/documentation/Identify.xml"/>
  <copy file="${src}/qtreeexample.png"
        tofile="${build}/documentation/qtreeexample.png"/>
</target>
```

The Ant task *exec* is used to execute **xsltproc** which generates the HTML and FO output. The Ant task *java* is used to execute the Java program FOP which generates the PDF output from the FO output produced by xsltproc. The temporary FO medium is deleted and the source XML and image file are copied to the output directory. Notice that the creation of the FO and HTML with xsltproc uses xslproc's ability to accept XSL parameters on the command line:

```
<arg line="- -stringparam generate.toc 'article nop'"/>
```

The XSL parameter *generate.toc* is set to *article nop* so that the generation of tables of content in articles is suppressed.

7.1. Directory Structure

Directory structure of the project, after an Ant build, is shown below:

```
build.xml
build
   bin
      Identify.class
      QuestionTree.class
      QuestionTreeTest.class
   documentation
      Identify.html
      Identify.pdf
      Indentify.xml
      qtreeexample.png
   javadoc
      allclasses-frame.html
      allclasses-noframe.html
      constant-values.html
      deprecated-list.html
      help-doc.html
      index-all.html
      index.html
      overview-tree.html
      package-list
      packages.html
      QuestionTree.html
      stylesheet.css
src
   Identify.java
   Identify.xml
   log4jconfig.xml
   log4jconfig2.xml
   qtreeexample.png
   QuestionTree.java
   QuestionTreeTest.java
```

8. References

- Ant [../ant/anthome.html]
- JavaDoc [../javadoc/javadochome.html]
- LOG4J [../log4j/log4jhome.html]
- JUnit [../junit/junithome.html]
- DocBook [../docbooksys/docbooksyshome.html]