Project #5

Computer Science 2334

Spring 2008

User Request:

"Create a utility to read and graphically display the directory structure, file names, and file sizes of a file system, starting at a user-specified directory."

Milestones:

1. Create a hierarchical database structure for file system information such as file and directory names and file sizes. The hierarchy will be a tree, to match the structure of the file system examined. This database will be called "**FileSystemDB**." *(10 points)*
2. Write code that will read in the names of all files and directories listed in a given directory, add this information to the database, and recursively descend into the listed directories (if any). *(20 points)*
3. Provide a dialog to select a starting directory for the recursive descent. *(5 points)*
4. Create an MVC model as exemplified by the **CircleModel** class from your textbook and as discussed in class. The class for this model will be called "**FileSystemModel**." This model will contain (1) the variables and methods needed to build and keep track of **FileSystemDB**, and (2) the variables and methods necessary to keep track of the view that will listen for changes to the model. *(10 points)*
5. Create an MVC view associated with the model using the **JTree** class. The class for this view will be known as "**FileSystemView**." Besides representing the file system database in a tree, this view will have buttons labeled "Pause" and "Resume" and an input field labeled "Delay Time." *(10 points)*
6. Create an MVC controller called **FileSystemController** associated with the model. When the user clicks the "Pause" button in a **FileSystemView** object, the controller will tell the model to stop adding new file system information to itself until the user clicks on a "Resume" button in a **FileSystemView** object. When the user changes the "Delay Time" value in a **FileSystemView** object, the controller will tell the model to change its update rate. *(10 points)*
7. Catch and handle I/O exceptions. *(5 points)*

- ► Develop and use a proper design. *(15 points)*
- ► Use proper documentation and formatting. *(15 points)*

Description:

As discussed in class, recursion is a very powerful and intuitive programming structure, since it often matches well with the relationships between the objects with which our code must deal. In this case, these objects are files and directories[1] in a computer file system. Because file system structure is typically defined recursively (directories can contain files or directories), a recursive structure to our software will make a

1 In some operating systems, directories are referred to as "folders." For this assignment we will use the term directories, regardless of OS.

great deal of sense.

Overview:

Modern computer file systems are typically organized in a tree.[2] "Descending" the tree from the root "down" to the leaves allows you to traverse the entire tree.[3] This descent could be done either depth-first or breadth-first.

Depth-first descent means that each time the tree branches, we follow a single branch of the tree down, until we come to a leaf, before backing up one level and dealing with other nodes at that level. Breadth-first means that we deal with all the nodes at the current level (starting at level one where the root is located) before we shift the current level to be the next level down.

One way to implement a depth-first search is to keep track of all the new nodes encountered by pushing them on a stack, then popping them back off again one at a time to deal with them. To change a depth-first program into a breadth-first program, all you need to do is switch from a stack to a queue data structure – that is, the program will add the new nodes to the back of a queue and remove them from the front of the queue to deal with them, rather than pushing them on and popping them off the stack.

Unfortunately, both of these approaches require you have to maintain a data structure (either a stack or a queue) to keep track of your "open" nodes (those that have not yet been visited). Alternately, you could implement your program recursively. Using recursive calls, the open nodes are automatically tracked for you on the call stack. This should make your code more intuitive and simpler to write and understand.[4]

Driver:

Your program should have a driver class with a main method that should request a starting directory for the directory traversal from the user using an appropriate dialog.. If no starting directory is entered (the user hits "Cancel" from the dialog, for instance), your program should use the "present working directory" as the starting directory. If a file is selected instead of a directory, your program should provide the user with an informative message in a dialog, then exit when the dialog is closed.

Once a directory is selected, your program should create the model, view, and controller and "wire" them all together so that they can interact.

Model:

You will create a model class called **FileSystemModel**. Models in the particular version of the MVC paradigm shown in the Circle example from your textbook contain data and methods for the application objects being modeled (in this case these will be files and directories) as well as data and methods to allow the model to interact with views. Your model class will follow this version of the MVC paradigm.

Your model will contain a database called **FileSystemDB**. The job of the database is to populate itself by recursively descending the directory tree. The structure of **FileSystemDB** should match the structure of the file system itself (i.e., it should be a tree). The database should descend through all the branches below the selected directory, out to the leaves. The leaves will be ordinary files or empty directories. For files encountered during the traversal, your program should determine the name and size of each one and enter

2 Although loops are permitted in some cases, for the sake of simplifying this project, we will assume the file system in question is a tree.

3 In computer science, we typically draw trees with the roots at the top and the leaves at the bottom. We really need to get out more.

4 One question for you to consider, as you prepare for your final exam, is whether the recursive code will result in a depth-first or breadth-first traversal.

that information in **FileSystemDB**. For directories, your program should record the name and the list of files and other directories it contains. In addition, it should give each directory a size equal to the size of all the files found below it on the tree. For example, if a directory contained two files, one 2MB and the other 20MB in size, the size of the directory would be recorded as 22MB. Naturally, your program will not know the correct size value for a directory until all of the files and directories below it are counted, so it will start by recording a size of zero for the directory when it is first encountered, and add to this value for each file and directory it contains. When all of the branches below a given directory have been traversed, we will say that directory is "closed." A closed directory should have a size value equal to the size of all the files below it in the tree added together.[5]

To interact with views, the **FIleSystemModel** class will have variables and methods akin to those from the **CircleModel** class in your textbook. In particular, when objects are added to the database in the model, the **FileSystetmView** object should be notified. In addition, to ensure that you can clearly see the model updates in action, the model will have a variable called **delayTime** that will specify a period of seconds for the model to delay between each update to itself. The default value for **delayTime** will be 1 second but the controller can tell the model to change this value (see below under ***Controller***).

View:

The **FileSystemView** class specifies a view of the database. This view will be created and displayed as soon as the starting directory is specified and the model is created. The hierarchical nature of the database should be reflected in the tree structure displayed in the **FileSystemView**. The starting directory will be the root. Whenever the database is updated, the view will receive notification of the change and will update itself to reflect the current state of the database.

The **FileSystemView** object will also have two buttons and a text field to accept user input. One button will be labeled "Pause" and the other "Resume." The input field will be labeled "Delay Time." When the user clicks on either button or changes the value in the input field, the controller will be notified.

Controller:

You will create a controller class called "**FileSystemController**" to handle the task of asking the model to pace itself in its updates. In particular, the controller will be responsible for the following:

1. When the user clicks on the "Pause" button, the **FileSystemController** will tell the database to temporarily stop updating itself.
2. When the user clicks on the "Resume" button, the **FileSystemController** will tell the database to resume updating itself.
3. When the user enters a new value in the input field, the **FileSystemController** will tell the database to use the value specified in the input field as the value for **delayTime**.

Extra Credit:

There are several possibilities for extra credit for this assignment, including adding additional info into the database and using an iterative method with a data structure to keep track of open nodes to allow for traversals in different orders. To determine how much extra credit will be given for any particular additional work, please check with Prof. Hougen.

5 To get file size, use **length()**. See http://www.exampledepot.com/egs/java.io/GetLength.html for an example of how to do this.

Due Dates and Notes:

1. No third-party GUI packages may be used. Only standard Java classes and packages are allowed on projects. **Using a non-standard class will result in the program not compiling on the graders' computers, and a non-compiling program will receive a grade of zero.**

2. Make sure to start early and budget your time well. Once you've got a good design, you can write a part at a time and test it before moving on to the next part. For example, you can test the **TreeView** code by importing the zips.txt file as you did in Project 3, selecting within the tree, then exporting the selected objects to a text file as you did in Project 3. Other code re-use from Project 3 may be applicable here as well, and can speed your development. **If you do not understand the design you developed in lab, it is your responsibility to attend office hours early in the project to get help with your design.**

3. Your revised design and detailed Javadoc documentation are due on Thursday, April 24th. Submit your revised UML design on engineering paper or a hardcopy produced using UML layout software, a hardcopy of the Javadoc documentation, and a hardcopy of the stubbed source code at the beginning of your assigned lab session. Submit the project archive following the steps given in the Submission Instructions by 9:00pm. ***Note that the design points will be tripled for this assignment, to encourage you to spend more time and effort getting the design ready before you begin coding!***

4. The final version of the project is due on Thursday, May 1st. Submit your final UML design on engineering paper or a hardcopy produced using UML layout software, a hardcopy of the Javadoc documentation, and a hardcopy of the source code at the beginning of your assigned lab session. Submit the project archive following the steps given in the Submission Instructions by 9:00pm.

5. You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

6. As noted in the syllabus, you are required to work on this programming assignment in a group of at least two people. It is your responsibility to find other group members and work with them. The group should turn in only one (1) hard copy and one (1) electronic copy of the assignment. Both the electronic and hard copies should contain the names and student ID numbers of all group members. If your group composition changes during the course of working on this assignment (for example, a group of five splits into a group of two and a separate group of three), this must be clearly indicated in your write-up, including the names and student ID numbers of everyone involved and details of when the change occurred and who accomplished what before and after the change.

7. Each group member is required to contribute equally to each project, as far as is possible. You must thoroughly document which group members were involved in each part of the project. For example, if you have three functions in your program and one function was written by group member one, the second was written by group member two, and the third was written jointly and equally by group members three and four, both your write-up and the comments in your code must clearly indicate this division of labor.