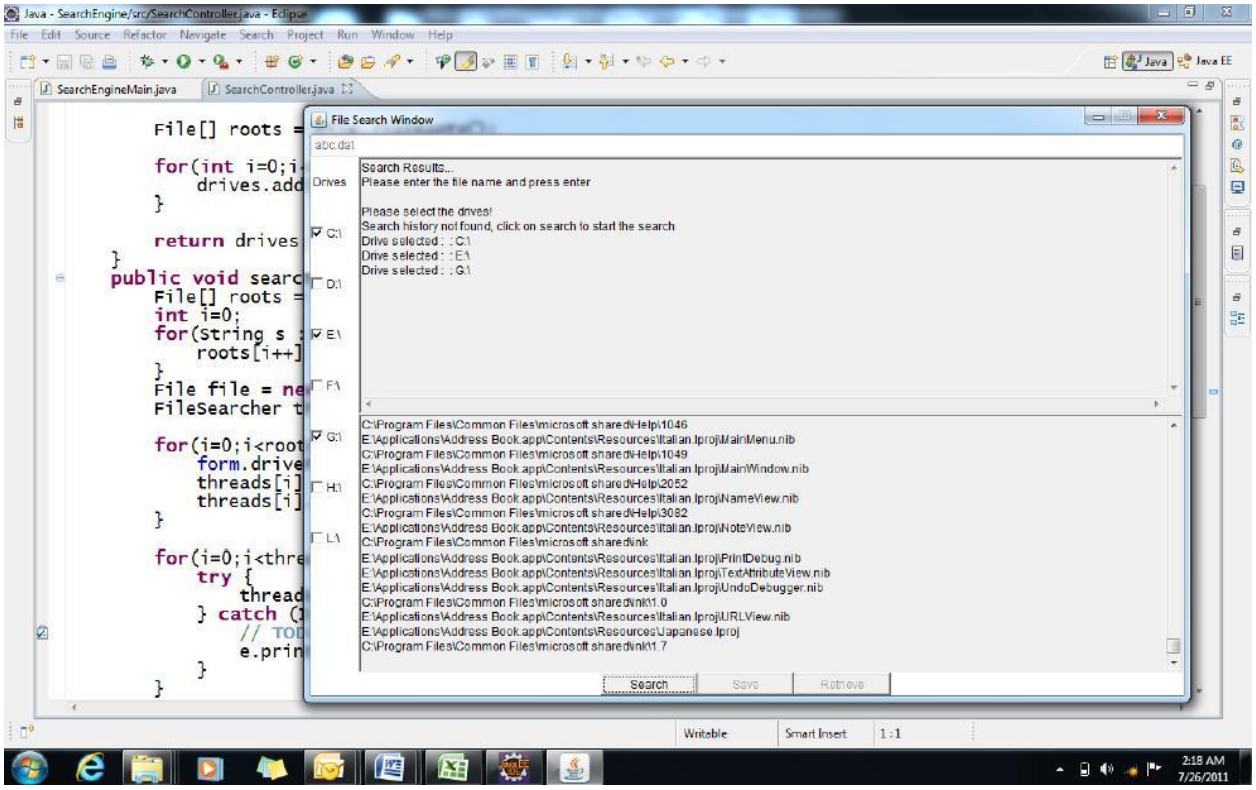
PT-2019-20-57..pdf PT-2019-20-56.pdf

Search Engine Application Project Requirements

PT-2019-20-55.pdf



We need to build a Desktop Search Application similar to Windows File Search.
This application unlike, the classical Windows Search Application will be multithreaded and hence very fast when compared to the classical search application available in Windows.

Requirements:

1. Detect the ROOTS / DRIVES in the system
2. Allow the user to select the DRIVES in which the file needs to be searched
3. Take the file that has to be searched as an input from the user
4. Perform the search by spawning one thread for each root / drive
5. Store the search results in a log
6. When a new search is initiated, check the history (or the log file) for availability of the previous search history
7. If the search history is available, display the search results after validating the existence of each file in the search history
8. If the search history is not found, prompt the user to initiate a new search

Technology Stack:

1. Java SE 6.0
2. OO Concepts
3. Abstract Classes & Interface
4. Packages
5. Exception Handling
6. IO Stream
7. Java Util Collection
8. Threads

Note: Things to be done

1. High Level Design
2. Coding guidelines (To be strictly followed and enforced)

Solution:

1. Discuss the problem. Big picture and explain that you would be working on this incrementally.
2. Identify the first step to solving the case study. Discuss the first step which will be finding out the roots of the system.
3. Discuss and get from the participants how to start the application. What would they do, how would they write code
4. Introduce them to the concept of an interface. We need a class to find out roots. Methods accordingly. If there are teams working on different aspects, one team has to wait for another to finish and then continue with search. Instead we define a contract, an interface **IRootFinder**. Create an interface IRootFinder. Implement the common methods and members.
    - List<String> roots - Member
    - getRoots - Method
    - getNumberOfRoots – Method
5. Now we shall write a class called **SystemRootFinder**, which will implement the IRootFinder and will contain the implementation of finding out roots
6. Have a **TestMain** class which will create an object of SystemRootFinder and display all the roots and the number of roots
   Accept the file name.
   Now we need to search for the file
7. We will start by having a class called **SearchManager**
   SearchManager will have a method called **startSearch**, which will take the roots to be searched and the file name as the parameters
    - void startSearch(File fileToSearch, List<String> roots)

- ? Now, do we go ahead and implement the code to search in **SearchManager** class?
  - ✓ No, this is because SearchManager is going to only manage the entire execution of the search activity.
- ? If not the SearchManager, who will perform the file search operations?
  - ✓ Create a class called **FileSearcher** which will carry out the search operation.
  - ✓ When we introduce concept of threads, our entire SearchManager need not be run on a thread. Only **FileSearcher can be implemented on the thread**.
- ? What will the Class FileSearcher contain?
  - ✓FileSearcher will have a method called searchFile, which will take the drive and the file name to be searched
  - ✓**void** SearchFile(File dir, File fileToSearch)
    - In this method we check recursively if it is a folder or a file, if folder, check if there are files again and then keep performing the search. If fileToSearch is found print the path [absolute path]
    - Now, when we keep displaying the path, you keep scrolling and you lose track of the search
    - So we will have it in an array list, collect all the files found as strings
  - ➢ **FileSearcher** will search and have the **collection of files found**.
  - ✓List<String> foundFilesPath = **new** ArrayList<String>();
- ➢ Let us also find out the number of folders it has scanned to search for files. So we will have another data member in FileSearcher class.
  - ✓**int** foldersScanned;
    - ➢ Write the get methods for the above declared members.

Now Search manager will fetch the collection with the FileSearcher objects that it has and display the results. Search manager displays all the **files found** and the **number of folders** that have been scanned

8. Now, what do we do if we want to find out the number of folders that have been scanned **in each drive**?
   Create a **SearchResult Class** which will have a list of files Found Path and number of folders scanned.
   Name the methods of this class appropriately as addFilesFoundPath and incrementNoOfFolders and respective getters
9. **Modify** the code in **FileSearcher** class to have an **object of SearchResult class** which will do both the jobs. Since we have the object of SearchResult Classes we need a getter to get the info from the class.
   Why do we need the filesFound list and noOfFoldersScanned in FileSearcher if we have the object of SearchResult?
   - ✓ We can remove the filesFound list and noOfFoldersScanned from the FileSearcher and also the associated get methods.

   FileSearcher should search and SearchResult should return the result to the SearchManager. Since we have the object of SearchResult in FileSearcher to update information , SearchResult should give the info to SearchManager.
10. Why should it search sequentially?
    - ➢ Let us implement **multithreading** to improve the speed of search
    - ➢ Which class should extend the thread?
      - ✓ FileSearcher should extend the thread and override the run()
    - ➢ How will you send directory and file name?

- ✓ Only way is to have a parameterized_ constructor in **FileSearcher Class**
  - **public** FileSearcher(File dir, File fileToSearch)
  - with members
    File dir;
    File fileToSearch;
- ➢ What code should be in the run()?
  - SearchFile(dir,fileToSearch);
- ➢ Let multithreading be an option given to the user. By default it is multithreaded.
  - ? How do we implement multithreading?
    - ✓ Let SearchManager Class have a
      1. boolean variable named multithreaded and get set methods for multithreaded variable.
      2. Default constructor which set the variable multithreaded to true by default
      3. Parameterized constructor to accept the user's choice.
      4. Check in startSearch() if multithreaded is true , and call start() else call run();

11. What if the user wants to search in a particular drive alone? Why should he be forced to search all the drives?
    - ➢ Create SearchLog Class which will have a Map<String drive, SearchResult>
      - ? What methods should SearchLog Class have?
        - ✓We should have a method that will
          - add a SearchResult,
          - return a SearchResult for a root drive and
          - getRoots of the system
      - ? How will we use SearchLog in SearchManager?
        - ✓ We can make StartSearch() to return a SearchLog
        - ✓ **public** SearchLog startSearch(List<String> roots, String filename);
      - ? Who will the StartSearch() return the Searchlog to?
        - ✓ It will return it to the main who requested for the search.
        - ✓ The searchManager just coordinates the activities, gets the search done, gets the results from searchLog and returns the searchResults to main.
12. Currently IRootFinder is an interface and SystemRootFinder implements that interface.
    Now what if the user only wants to search in active roots and not all the roots as SystemRootFinder does?
    - ✓ Create a class ActiveRootFinder -- which implements all the methods of the IRootFinder interface and which will not include drives which cannot be read.
    - ? The numberOfRoots is a System is fixed. The method getNumberOfRoots() returns the same value in both ActiveRootFinder and SystemRootFinder. So why should we replicate it in two places? Can we avoid replication?
      - ✓ Create an abstract class AbstractRootFinder which implements the IRootFinder Interface
      - ✓ ActiveRootFinder and SystemRootFinder inherits from AbstractRootFinder.
      - ✓ Abstract class AbstractRootfinder has
        - **protected** List<String> roots = **new** ArrayList<String>();
        - Overridden methods of the IRootFinder interface
        - Concrete method getNumberOfRoots() which will return the no. of roots and hence avoids duplication in the inherited subclasses.

13. How will the user choose between ActiveRootFinder and SystemRootFinder? If we have more options PendriveRootFinder and CDRootFinder then should user create all these objects?
    - Create a RootFinderFactory which will return the object wanted.
      - A RootFinderFactory will have a Create(). What can be the argument of this Create()?
        - ✓If the method signature were to be : IRootFinder create(**String** rootValue) the user has to enter ActiveRootFinder, SystemRootFinder , then we might as well create an object.
        - ✓Let us create an enum Roots where if user wants to search all directories then he will enter Roots.ALL_ROOTS and if user wants to search only **active** roots he will enter Roots.ACTIVE_ROOTS.
          - **public enum** Roots{*ALL_ROOTS*, *ACTIVE_ROOTS*;}
14. After implementing the above, go to TestMain and use the RootFinderFactory to create the type of search
15. Our SearchLog class has a Map<String drive, SearchResult>. We can effectively use this to store the searched results as a search history log file on hard disk. If the user initiates another search with the same file name, then we can first check if the log file exists, if so display the results from this log file. If user requests a fresh search then initiate a fresh search.
    - How do we create the log file and what will be the name of the log file?
      - ✓ We can create a log file say with the name of LogFile.txt itself. But if we keep on adding all the search results to this LogFile.txt, searching thru this log file can be more time consuming than doing a search on disk.
      - ✓A solution would be to create a log file for each filename searched, replacing the dot with a underscore and an extension of .log . For example Log file for HelloWorld.java would be saved as HelloWorld_java.log

    Create a class SearchHistoryMgr with the following methods
    - A method to convert the searchFileName to LogFileName
      String getLogFileName(String fileName) ;
    - To create and write to Log File on hard disk
      boolean saveSearch(String fileName, ,SearchLog searchLog)
    - Check if file exists. Get LogFileName of the given filename and then check if file exists.
      boolean isHistoryAvailable(String fileName)
    - Read from log file and add to a List of Strings and return List of filesfoundpath.
      List<String> getSearchHistory(String fileName)
    - What if one of the files present in the Log File is deleted after the previous log file has been saved?
      - ✓If History is available and if one of the files in history was physically deleted then check if the files returned from history exists using file.exists().
    - What if one or more files with the same name have been added since the last log file was saved?
      - ✓If a new file is added, then display the log in the history and ask the user if he is satisfied with the result or he wants to start a fresh search

STEPS TO INTEGRATE WITH UI

16. As a first step to integrate with UI , Please share the following java files with the participants
    - ISearchControl.java
    - ISearchForm.java – An interface of methods which adds functionality to UI
    - SearchForm.java – A Concrete class which implements the interface and all the functionality has been coded. When working in Real Time projects the UI is designed by the UI team

When working in Real Time projects the UI is designed by the UI team and in all probability the .class file will be shared. We are sharing the .java file with the participants, so that they can have a look at how this UI has been designed and coded.

17. **Given** : We are given the SearchForm UI (View)
    Have : SearchManager and other classes (Model)
    Need : A class which will talk to the SearchForm UI and to SearchManager and other classes. (Controller)
    Observe: SearchForm.java calls the methods of ISearchControl Interface.
18. Create class SearchControl which implements the ISearchControl Interface and is a Controller which talks to the SearchForm UI and to SearchManager and other classes.
    Participants must create a class SearchControl which implements the ISearchControl Interface and provide functionality for all the methods.
19. Role of Class SearchControl
    - SearchControl acts as a Controller between the UI or the View (SearchForm) and Model (SearchManager, SearchHistoryManager).
    - SearchControl should be a SINGLETON. **(Why?)**
    - Participants must create appropriate instance variables in the SearchControl Class
    - SearchControl should have a reference to the SearchForm for UI Notification based on the interactions with the SearchManager
20. Modification in **SearchManager Class**
    - Since SearchControl is taking complete control, the SearchManager need not check if history is available or not. The startSearch() of the SearchManager should only search for files in disk.
    - Create a method getRootFinder which will accept enum Roots as argument and return either an object of SystemRootFinder or ActiveRootFinder.
      ✓IRootFinder searchMgr.getRootFinder(Roots.ALL_ROOTS);
21. Role of SearchControl Class
    - We observe that the SearchForm UI has 2 sections : The first section displays the files found and the second section displays all the files being scanned.
    - The View (SearchForm) is populating as and when the Model (FileSearcher) is scanning or finding a match
    - SearchControl should get the information (files found and files scanned) from SearchManager and populate the appropriate sections.
22. How will the SearchControl get the information from SearchManager? How will SearchManager get the information from FileSearcher?
    ✓The Solution is implement CallBack Option
23. What is CallBack ? Where and how should it be implemented?
    - FileSearcher currently is searching for all the files found and populating the instance of the searchResult in the FileSearcher Class.
    - But the need is, as soon as one file is scanned or a match is found, FileSearcher has to intimate/inform the SearchManager who in turn will inform the SearchControl by calling the respective methods
      ✓SearchControl.*getInstance*().fileFound(message);
      ✓SearchControl.*getInstance*().folderScanned(folderName);
    - ? So where and what changes needs to be implemented?
      ✓ Modify FileSearcher Class
      ✓Modify SearchManager Class
24. Modifications in the SearchManager Class

- Create a method folderScanned to accept a String pathOfFolderScanned

```
public void folderScanned(String folderName) {
    SearchControl.getInstance().folderScanned(folderName);
    }
```

- Create a method fileFound to accept a String fileFoundPath

```
public void fileFound (String message) {
    SearchControl.getInstance().fileFound (message);
    }
```

25. Modifications in the FileSearcher Class
    1. Add a Reference to the SearchManager
       ✓**private** SearchMgr searchMgr;
    2. Modify the constructor to add an argument of SearchMgr
       ✓**public** FileSearcher(File dir, File fileToSearch,SearchMgr searchMgr)
    3. Whenever a Folder/File is scanned call the SearchManager's folderScanned() sending the absolute path as an argument
    4. Whenever a match is found, call the SearchManager's fileFound() sending the absolute path as an argument
26. Creating the Class SearchControl
    - Class SearchControl implements ISearchControl
    - SearchControl has to be a Singleton
      - ✓ Create a private static ISearchControl instance.
      - ✓ Static getInstance() to return the instance
      - ✓ Private constructor
    - Instance Variables needed are Instance of class
      - ✓ SearchForm
      - ✓SearchManager
      - ✓SearchHistoryManager
      - ✓SearchLog
      - ✓String fileName
    - **public** List<String> getDrives() - Get all the drives of the system
    - **public void** searchForFile(String fileName, List<String> selectedDrives)
      - ✓Makes a call to SearchMgr to startSearch which returns a SearchLog
      - ✓Enables the save button in the UI
    - **boolean** checkSearchHistory(String fileName) -
      - ✓Checks if history is available
    - **void** saveSearch()
      - ✓Calls the SearchHistoryManager object to save the History to the log file
      - ✓Sends a message to the UI to indicate the history was successfully saved
    - List<String> retrieveSearch()
      - ✓Retrieves the searchHistory from SearchHistoryManager
    - **public void** launch()
      - ✓ Create a object of SearchForm to launch the UI
    - **public void** fileFound(String message)
      - ✓ Send the message to the UI
    - **public void** folderScanned(String message)
      - ✓Send the message to the UI
27. TestMain
    - It should get the Instance of SearchControl and launch the GUI application.

28. The Trainers need to discuss the startSearch() of SearchForm class as follows.
This method helps in understanding of Threads.
We know that Main() is a thread, awt is a thread (UI thread). Apart from this we have other threads like GC that run in parallel with Main()thread and Awt thread.
Consider the code of the startSearch()

```
private void startSearch()
{
  Thread r = new Thread(new Runnable(){
                        @Override
                        public void run() {
                            results.setText("");
                            searchFile.setEnabled(false);
SearchControl.getInstance().searchForFile(searchFile.getText().trim(),selectedDrives);
                                searchFile.setEnabled(true);
                        }
                    });
                    r.start();
          }
```

? Why do we need this Thread r? Instead we could have the following code

```
private void startSearch()
{
   results.setText("");
   searchFile.setEnabled(false);
SearchControl.getInstance().searchForFile(searchFile.getText().trim(),selectedDrives);
    searchFile.setEnabled(true);
}
```

? Why is it important to have the Thread even if the above code seems to work?
- ✓Even if the code seems to work, the GUI has become unresponsive, unable to even close the UI.
- ✓Reason is : Main is a thread and it launches thro SearchControl, the SearchForm which is another thread (awt thread). When the awt thread starts it cannot be stopped till the process is complete. This awt thread will only terminate on completion of the search, hence the gui is unresponsive.
- ✓So it is important that we start a new thread for searching the file which on completion terminates and its behavior is delinked from behavior of awt thread. This thread is necessary to ensure the gui appropriately responds.

PACKAGING

29. Let package our application. Please drag and drop the classes to respective packages
- ➢ Package Name : com.pratian.searchengine.control
  - ✓ISearchControl.java
  - ✓SearchControl.java
- ➢ Package Name : com.pratian.searchengine.model.roots

- ✓IRootFinder.java
- ✓Roots.java
- ✓AbstractRootFinder.java
- ✓ActiveRootFinder.java
- ✓SystemRootFinder.java
- ✓RootFinderFactory.java
- ✓InvalidRootDescriptionException.java

➢ Package Name : com.pratian.searchengine.model.search

- ✓FileSearcher.java
- ✓SearchManager.java
- ✓SearchResult.java
- ✓SearchHistoryManager.java
- ✓SearchLog.java

➢ Package Name : com.pratian.searchengine.view

- ✓ISearchForm.java
- ✓SearchForm.java
- ✓Console.java

➢ Package Name : com.pratian.searchengine.utility

- ✓TestMain.java

30.