10 Project

Goals

The goal of the final project is to use the concepts and techniques you learned in this course in designing a functional complete program that has some use.

We will provide some components for the user interaction with the program, but you will be responsible for the design and imlementaiton of the functionality.

Overview

There are three parts to this project.

- In the first part you will design data structures that will be used by yur program — a variant of a stack, queue, and priority queue that all implement a common interface.
- In the second part you will brainstorm and produce a written description of the design of your program — the data definitions you may need, the way the functionality will be implemented, the way the program will interact with the user.
- In the third part you will implement the design in a working program.

Project Description

Your program will allow the user to view a map of the 48 lower United States with the locations of the capitals, and compute a routing from the user-selected origin to the user-selected destination. The user can select one of three possible methods for finding the route: a depth-first search (*DFS*), a breadth-first search (*BFS*) or a search for the shortest path (*SP*).

Your program will display a graph with nodes that represent capitals of the 48 US states. For each node the program records a name — the name of the state. For each node, we record the information about the capital of that state. Each edge represents a bi-directional connection between two adjacent states. You may consider the *four corner states: Colorado Utah, Arizona and New Mexico* as connected to each other. Each edge has a value that represents the distance between the capitals of the two states. The distances

between two cities are based on the geographic distance. (See a separate announcement for a shorcut you can use to compute this distance.)

The GUI that allows the user to select one of the algorithms as well as the origin and the destination will be provided. Your job is to display the graph and then highligh the chosen path when the algorithm completes the work. You may visualy represent the steps in the search, but you are not required to.

10.1 Stacks, Queues, and Priority Queues

Graph traversal algorithms need to keep track of *work to be done* — specifically, the nodes we should visit next when searching for a path from one node to another on a map. We call this data structure an *Accumulator*. The three algorithms *DFS*, *BFS*, and *SP* differ only in the way how we add/remove items from this accumulator. Therefore, we start with a common interface, and design three different implementations of this interface.

The *Accumulator* interface is defined as follows:

```
/**
  * <P>An interface that represents a container for accumulated collection of
  * data elements. The implementation specifies the desired add and remove
  * behavior.</P>
  * <P>The expected implementations are Stack, Queue, and Priority Queue.</P>
  */
public interface Accumulator<T>{


    /**
      * Does this <CODE>{@link Accumulator}</CODE> contain any data elements?
      * @return true is there are no elements in this
      * <CODE>{@link Accumulator}</CODE>.
      */
    public boolean isEmpty();


    /**
      * Effect: Change the state of this <CODE>{@link Accumulator}</CODE>
      * by adding the given element to this <CODE>{@link
      * Accumulator}</CODE>.
      *
      * @param t the given element
      */
    public void add(T t);
```

```
    /**
     * Effect: Change the state of this <CODE>{@link Accumulator}</CODE>
     * by removing the given element to this <CODE>{@link Accumulator}</CODE>.
     * Produce the removed element.
     *
     * @return the removed element
     */
    public T remove();
}
```

1. Design the class *MyStack*<*T*> that implements the *Accumulator*<*T*> interface by always removing the most recently added element.

2. Design the class *MyQueue*<*T*> that that implements the *Accumulator*<*T*> interface by always removing the least recently added element.

3. Design the class *MyPriorityQueue*<*T*> that contains an instance of a *Comparator*<*T*> and implements the *Accumulator*<*T*> interface by always removing the element that has the highest priority as determined by its *Comparator*<*T*>.

4. Design the classes *IllegalStackOperation IllegalQueueOperation* and *IllegalPriorityQueueOperation* that extend the class *Exception* in the *java.lang* package. Modify the methods that implement the *Stack*, *Queue*, and the *PriorityQueue* so that they *throw* the appropriate exceptions.

   Explore the Java documentation and in online tutorials to see how to *throw* and *catch* an *Exception* that is not a subclass of the *RuntimeException*.

Note: You can decide on your own what will be the class of data that will provide the elements to use in testing these classes.

10.2 Algorithms

Your model should implement three graph traversal algorithms (we will discuss these in class on Monday, November 19th):

- Depth-First Seaarch
- Breadth-First Search
- Shortest Path Search

10.3 User interactions

The interactions at the minimum should have the following functionality:

- User should be able to see a graphical representation of the graph.
- User should be able to select which of the three algorithms is to be used for the subsequent task.
- User should be able to specify the origin and the destination of the desired path.
- The user should be able to see the resulting path.

The frills

Of course, the view can be much more elaborate. Here is a list of possible enhancements:

- Highlight the path is a different color in the graphics display.
- Display the steps in the search by highlighting in a different color the *v*isited nodes, the *f*ringe nodes (those currently in the queue or the stack), the *o*rigin, the *t*arget, and the *u*nseen nodes. Animate the process using either the timer, or a user advance triggered by a key press.
- Animate the reconstruction of the path by traversing from the found target back to the previous node, all the way up to the origin.
- Display in a GUI the path length and possibly the nodes along the path.

The Advice

The design part of each project typically takes the greatest amount of time. the more time you spend thinking things through, the easier it is to actually write the code.

Make sure you think the whole framework through before you start programming. Spend some time researching the Java libraries to see what tasks can be done using the existing tools. Write sample adapters to see how the existing class can be used in your setting.

For example, in our sorting assignment the Traversal interface allowed us to supply the data to the algorithm in a number of different ways — and allowed us to the produce the result in a universally readable manner as well.

Then design the key component by specifying their interfaces — the method headers, the interfaces that various classes must implement or use to get information from others.

For now, you have not learned about various tools and techniques to support such design process — other than class diagrams. Any description that you find helpful in clarifying the roles of the different classes and interfaces in your program is acceptable.

The design document you produce (which could be primarily in the form of javadocs) should describe all data definitions and the key methods, as well as give a general overview of the project organization.

The Documentation: a concise summary

You may have noticed that the style in which we write documentation for this assignment has changed. When written in the well formatted *javadoc* style, the comments can used to generate web pages of documentation with cross-references and browsing capabilities. There are a few basic rules, the rest you should learn on your own, gradually, as you become more and more skilled Java programmers.

Here are comments to specify the name of the file, and the class definition:

```
/*
 * @(#)Word.java    17 November 2007
 */

/**
 *  <P><CODE>Word</CODE> represents one word and its
 * number of occurrences counted in the
 * <CODE>{@link WordCounter WordCounter}</CODE> class.</P>
 *
 * @see Comparable
 *
 * @author Viera K. Proulx
 */
public class Word implements Comparable {
```

The @*author* and @*see* identify the author and provide a cross-reference to other classes as specified.

Each field in the class has its own comment:

```
/**
 * the frequency counter
 */
public int counter;
```

Each method has a comment that includes a separate line for each parameter as well as for the return value:

```
/**
 * Compare two <CODE>Object</CODE>s for equality
 *
 * @param obj the object to compare to
 * @return true if the two objects have the same contents
 */
public boolean equals(Object obj){
```

The @*param* has to be followed by the identifier used for that parameter. The <CODE> and < /CODE> tags specify the formatting for the document to be the teletype font for representing the code.

Eclipse helps you to write the documentation. If you start the comment line with /** and hit the return, the beginnings of remaining comment lines are generated automatically, and you only need to add the relevant information.

When you have finished all the documentation, select the item **Generate Javadoc...** in the **Project** menu. To see your web pages, just open the tab *doc* in the **Package Explorer** window under your project and double click on the *index.html*.

Enjoy