Chris Piech
CS 106A

Assn #7
March 8th, 2019

Assignment #7—Shahidi
Due: 5pm on Friday, March 15th (no late days allowed)
This assignment may be done in pairs

Created by Brahm Capoor, Chris Piech and Peter Maldonado

Introduction

In the wake of a hotly contested presidential election in Kenya in 2007, programmers across the country came together to build a website and piece of software collectively called *Ushahidi* (Swahili for 'testimony'). These programs, designed to allow Kenyan citizens to report violence across the nation, quickly drew the attention of NGOs and media organizations worldwide due to the speed and accuracy with which crises were reported relative to the Kenyan mainstream media. In the years since then, Ushahidi—or various pieces of software inspired by it—has been employed to aid recovery efforts after the 2010 earthquake in Hawaii, to report oil spill sites after the Deepwater Horizon explosion, to track forest fires in Italy and even to monitor political violence in the 2016 U.S. Presidential elections.

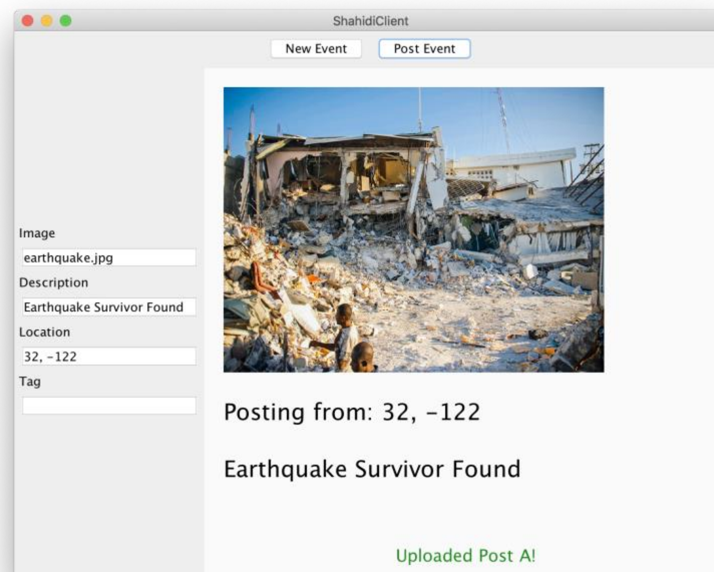
Ushahidi is a significant piece of software for two reasons. Firstly, it was one of the progenitors of Kenya's rapidly growing technology industry (sometimes called 'The Silicon Savannah'). Secondly, and arguably more relevant to you, it tells the story of how the skills you gain in CS 106A can be applied towards important problems in order to come up with effective and innovative solutions. As we near the end of the quarter, you may be thinking about how you can begin to use Computer Science to answer the questions that mean the most to you and this assignment allows you to gain some appreciation for the requisite thought processes.

In this assignment, you will be implementing a basic version of Ushahidi called *Shahidi* (Swahili for 'witness'). Shahidi is an internet-based program that allows users to anonymously upload details about events that have occurred near them, for instance, a description of the event, its location and an image. Your job is to implement only the server for Shahidi, which is responsible for storing information about all the events that have been uploaded, as well as serving details about those events to clients that request them. However, we provide multiple different clients in order to better demonstrate the use cases of the application, as well as to highlight the difference between the server you are writing and the clients that will communicate with it. The first client is a Java Graphics program that acts somewhat like a mobile application, allowing you as a user to upload images and other details about events. The second is a webpage that allows you to browse around the world for all of the events that occurred near a particular location. Finally, we give you a Console Program that subjects your server to a variety of tests to ensure that it is not misbehaving.

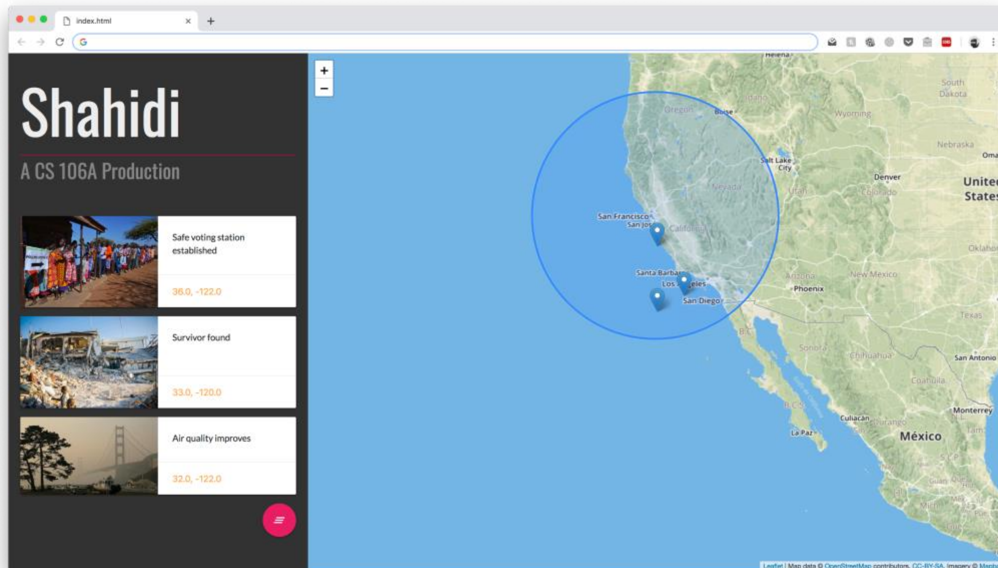
Summary

There are 4 portions to this assignment, across the two Eclipse projects you need to import:

- **The Shahidi Server** (**`ShahidiServer.java`** in the **`ShahidiServer`** project): In this file, you will implement the server responsible for storing and managing all the events for Shahidi. It will comprise all of the code you will be writing for this assignment, written either in this file or in supporting classes you will be implementing.
- **The Testing Framework** (**`ShahidiServerTests.java`** in the **`ShahidiServer`** project): This file contains a battery of tests—which have already been implemented for you—that check whether your server is returning what it is expected to, and which you can use to make sure your server is correctly implemented.
- **The 'Uploader' client** (**`ShahidiClient.java`** in the **`ShahidiClient`** project): This program—already implemented for you—simulates a mobile application that allows you to describe an event and specify its location, as well as optionally include an image, before posting the event to the Shahidi server. You may use this program to make individual requests to your server. We also provide the code in **`ShahidiClient.java`** and **`ShahidiEventView.java`** for your perusal and if you wish to make any extensions.



- **The Browser client** (`webpage/index.html` in the `ShahidiClient` project): This webpage—also already implemented for you—is a webpage that uses your server to display information about various events, as well as to find all the events that occurred within a particular radius of some location. To use the webpage, navigate to the client Eclipse project's `webpage/` directory in your file explorer and double click `index.html` in order to open it in your browser.



Note that only the first of these components is code that you are expected to be familiar with. You are welcome to use and read the code for the other parts to gain a better understanding of how they work (as well as to add additional tests, if you wish), but this handout should serve as an exhaustive reference and your `ShahidiServer` should work with **unmodified** versions of these files.

If your program passes all the tests provided in `ShahidiServerTests.java`, you can take that as a sign that for the most part, it is correctly implemented and you need only consider if there are additional edge cases you must account for. Neither the web nor uploader client are necessary to test your program but serve to emphasize the abstraction between the server you are writing and the various kinds of clients that can make requests to it.

Milestone 1: The `Coordinate` class

As an initial step towards your implementation of Shahidi, you will be implementing a class representing a single coordinate in order to be able to effectively identify and represent locations in the world. The `Coordinate` class encapsulates an x and y coordinate, which we'll refer to as the *longitude* and *latitude*, respectively. Each of these components is represented as a double. Your job is to implement this class in `Coordinate.java` with the following public methods:

`public Coordinate(String coordString)`

In this constructor, initialize the state of a new coordinate from the given string. **`coordString`** consists of the coordinate's latitude, followed by a comma and space, followed by its longitude. For example, the coordinate with a latitude of 42.5 and a longitude of 100.6 would be represented by the string **`"42.5, 100.6"`**. You may assume the string passed in represents a valid coordinate.

`public double getLongitude()`

In this method, return the longitude associated with this coordinate.

`public double getLatitude()`

In this method, return the latitude associated with this coordinate.

`public String toString()`

In this method, return a string representation of this coordinate. The string should be formatted as the coordinate's latitude, followed by a comma and space, followed by its longitude (just like the string passed into the coordinate constructor).

In a later portion of this assignment, you'll be calculating the distance between different coordinates. To aid you in this calculation, we provide an implementation of the following public method in the **`Coordinate`** class:

`public double distanceTo(Coordinate other)`

Returns the distance between this coordinate and the coordinate represented by `other`.

We'll discuss this method in more detail later on but for now, note that this method will only work with a correct implementation of the getter methods outlined above.

As a cautionary note, avoid using **`long`** as the name of any of your variables. **`long`** is a variable type in Java (used to represent large numbers) and so is an invalid variable name.

In order to test your implementation of the Coordinate class, use the test suite provided in **`ShahidiServerTests.java`**. The first group of tests in this section test the various different methods you are required to implement.

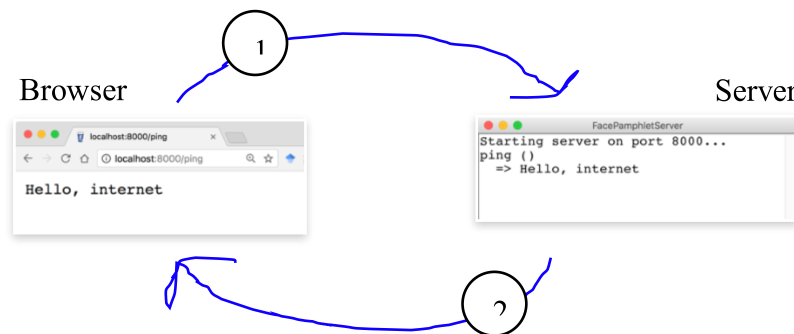
Milestone 2: Ping

As a first milestone towards writing a server for Shahidi, have your server respond to the "ping" command. If your server receives a request with the command "ping", you should simply return the string "Hello, internet". Open **`ShahidiServer.java`** to get started.

The starter code is already set up with a **`SimpleServer`** instance variable. In the **`run`** method when we call **`start()`** on the server variable, this signifies your program saying it is ready to receive incoming requests. Every time a request is sent to your program, the method **`requestMade`** will be called with the details of the request. As we talked about in class there are two methods that you can call on the request that you are passed in: **`getCommand()`**, which returns the request's command, and **`getParam(`*`key`*`)`** which returns the value associated with a request parameter. Your task for this milestone is to update the **`requestMade`** method to check if a given request has the command **`"ping"`**, and if so return **`"Hello, internet"`**.

For now, we can test out our ping response by making a request to your server from a web browser (e.g. Chrome or Firefox). Run your server program and navigate to http://localhost:8080/ping in your browser to send a request to the server with the command "ping". Your browser will display the string that your server sends back.

The browser sends a request to the server with the command "ping"



The server responds with the string "Hello, internet" which the browser displays

The server in the picture above **`println`**s the received request and the response returned. You do not have to imitate this functionality—what matters is that your server returns the appropriate string. That said, console output is useful for debugging.

In order to test your implementation of this request, use the test suite provided in **`ShahidiServerTests.java`**. The second group of tests (which in fact consists of only one test) ensures that you are returning the correct string.

You now have a program that is receiving and responding to an internet request!

Milestone 3: Distance between two coordinates

Having established much of the infrastructure you'll need for more sophisticated behavior in Shahidi, your next task is to implement the following request, also in **ShahidiServer.java**:

Command	Parameters	Response
distanceBetween	coord1, coord2	Returns the distance in kilometers between the two coordinates represented by coord1 and coord2. Each coordinate string is formatted as the coordinate's latitude, followed by a comma and space, followed by its longitude (just like the string passed into the coordinate constructor).

In order to do this, you'll be leveraging the **distanceTo** method in the **Coordinate** class. As its method header suggests, this method takes as another coordinate and returns the distance between the two coordinates. Assuming the rest of your class implementation is correct, you would be able to use this method as follows:

```
Coordinate c1 = new Coordinate("10, 20");
Coordinate c2 = new Coordinate("42, 5");
double distance = c1.distanceTo(c2);
```

In order to test this request, use the third group of tests in the test suite provided in **ShahidiServerTests.java**. The **distanceTests** method makes two **distanceBetween** requests to the server and ensures that the correct value is returned.

Milestone 4: The `Event` class

You'll now turn your attention to designing and implementing an **`Event`** class in **`event.java`**, which encapsulates all the information associated with a particular event's occurrence. Specifically, the class must keep track of the following information for each event:

- A textual description of the event
- Its location
- An image of the event

However, it is up to you to determine how best to represent this information, and consequently which public and private methods you must implement as well as what information you require in the constructor for the class. In Shahidi, you are guaranteed that every event will have a description and a location but might not have an associated image. You may wish to consider the request types you will be implementing in milestone 5 as you design your class.

We also recommend you add a public **`toString`** method to your class. While not strictly required, it does greatly facilitate testing and debugging your program.

Appropriately designing classes is more an art than a science and as you proceed through the rest of this assignment, you might find that you need to reevaluate the decisions you made while implementing this class.

Milestone 5: Event Requests

Now, we turn to the various kinds of server requests associated with managing Events on Shahidi. Your job in this milestone is to implement the following requests:

Command	Parameters	Response
`addEvent`	`description, coordinates, image (optional)`	Create an event with the given description, coordinates and image. Returns the unique id of the event (see below). If the event has no image, request has no **`image`** parameter. **`coordinates`** is formatted as the coordinate's latitude, followed by a comma and space, followed by its longitude (just like the string passed into the coordinate constructor).
`deleteEvent`	`eventID`	Deletes the event with the given id from Shahidi. Returns "success" or, if the event doesn't exist, returns an error message.
`getDescription`	`eventID`	Returns the description of the event with the given id or, if the event doesn't exist, returns an error message.
`getCoordinates`	`eventID`	Returns the coordinates of the event with the given id or, if the event doesn't exist, returns an error message.
`getImage`	`eventID`	Returns the image of the event with the given id or, if the event doesn't exist, returns an error message.

An error message is any string which starts with "Error:". The remainder of the string describes what went wrong. For example:

`"Error: Event with ID 'xyz' does not exist"`

In order to be able to easily refer to Events, your server must also assign each event a unique ID. To get a unique ID for an event, use the following method, provided for you in **`ShahidiServer.java`**:

```
private String getNextID()
```

This method returns an ID that is guaranteed to be unique every time you call it. In your server, think carefully about the kinds of collections that allow you to easily associate an ID with its corresponding Event.

In addition, two commands in this milestone have to do with getting and setting images. When sending information over the internet, *everything* has to be text; even images. For this reason, when handling images in your server, you will need to convert between **GImage** and **String**. We have provided two methods to help you do this. For instance, when you receive a request with command **addEvent**, the parameter **image** is a string, not a **GImage**. To convert it to a **GImage**, you could write:

```
String imageString = request.getParam("image");
GImage image = SimpleServer.stringToImage(imageString);
```

Similarly, when you receive a request with command **getImage**, to convert a **GImage** on the server to a **String** to send it as a response, you could say:

```
GImage image = ...
String imageString = SimpleServer.imageToString(image);
```

Note: a null image converts to the empty string, and vice versa. Calling **stringToImage** with a poorly formatted input string throws an **IllegalArgumentException**.

Milestone 6: Nearby Events

Your job in this milestone is to implement the following request

Command	Parameters	Response
`nearbyEvents`	`coordinates`, `radius`	Finds all the events within `radius` km of the specified coordinates, and returns a list of their IDs.

For example, if the events with IDs 'ab' and 'zxbcd' are within 10 miles of the coordinates (42, 5), then when the server receives a **`nearbyEvents`** request with the parameter **`coordinates = "42,100"`** and **`radius = 10`**, return:

> **`"[ab, zxbcd]"`**

which is a string representation of the list of nearby event IDs. If there are no nearby events, simply return the string:

> **`"[]"`**

As a hint, **`ArrayList`**s have a **`toString`** method which returns exactly this string representation.

To find all the events within this radius, you will need some way of processing every event in turn and finding whether its distance from coordinates is less than radius kilometers. You may find the **`distanceTo`** method you used in Milestone 3 helpful as you do this.

Tips and Tricks

- *Test early, and often.* This assignment is heavily incremental, in that almost every milestone is reliant on the milestones preceding it. Thus, in order to be confident in your implementation of a particular milestone, think carefully about how to test it. What sorts of inputs are you expected to deal with? What are the edge cases?
- *Try each of the clients.* Each of the client tests a particular kind of behavior. **`ShahidiServerTests.java`** tests the behavior of your program under a relative amount of duress. In particular, it is the only client that tests your **`distanceBetween`** and **`deleteEvent`** requests. **`ShahidiClient.java`** tests your ability to post individual events to your server. Finally, the webpage tests your **`nearbyEvents`** request, as well the way in which you handle your individual **`getDescription`**, **`getCoordinates`** and **`getImage`** requests.
- *Use logs in your server.* As you test and debug your server, it is crucial to understand exactly why it might be misbehaving. One of the easiest ways to do this is to liberally **`println`** various pieces of information about the values of variables and request parameters. You can refer to the code for the chat client in lecture for examples of how to do this.

Suggested Extension: Event Tags

As you might have noticed, **`ShahidiClient`** also contains a **`JTextField`** that allows you to add tags to an event. While you are *not* expected to implement this feature server-side for the required part of this assignment, we encourage you to do so as an extension. If you wish to do so, implement the following (modified) version of the **`addEvent`** request:

`addEvent`	`description,`
	`coordinates,`
	`image (optional),`
	`tags (optional)`
		Create an event with the given description, coordinates, image and tags. Returns the unique id of the event (see below). If the event has no image, request has no **`image`** parameter. **`coordinates`** is formatted as the coordinate's latitude, followed by a comma and space, followed by its longitude (just like the string passed into the coordinate constructor).
		
		In addition, the `tags` parameter is formatted as the **`toString`** of an **`ArrayList`** of tags, for example:
		
		`"[stanford, weather, today, sunny]"`

You are also free to add any other tag-related requests you like, as well as to modify **`ShahidiClient.java`** and **`ShahidiEventView.java`** for the purposes of this extension. Some ideas for such requests are **`getTags`**, which returns all the tags for a particular event and **`withTag`**, which return all the eventIDs whose corresponding events have a particular tag.