Lab: A Festival of Lights

Some Light Reading

This lab will feature a hypothetical system, in which various light panels are connected to a light system. Each panel features a light bulb. When any panel's "on" button is pressed, all the lights turn on. Likewise, each "off" button turns off all the lights.

We will simulate such a system by creating instances of the Java classes LightSystem and LightPanel. These instances may reside on different computers, as pictured above.

Go ahead and download all the files from /home/cs/332/sp2016/lab1/ to a new directory under your home directory.
For the most part, all we'll do with the `LightSystem` class itself is make new instances of it. Here is a summary of what is in the class.

**LightSystem Class**

`DEFAULT_PORT`

- `LightSystem()`
- `LightSystem(int port)`
- `static Random getRandom()`

The `LightPanel` class allows you to check if the light is on, and turn the light on or off. In addition, each `LightPanel` has an ID (an integer from 1 to 15). Here is a summary of the `LightPanel` class.

**LightPanel Class**

- `LightPanel()`
- `LightPanel(String host, int port)`
- `int getID()`
- `boolean isOn()`
- `void switchOn()`
- `void switchOff()`
- `String toString()`

Although a `LightPanel` may conjure up a visual image in your mind, you may create and use `LightPanel`s without ever seeing one appear on the screen. If you would like to see a `LightPanel`, you must create an instance of the `LightDisplay` class, and pass it the `LightPanel` you wish to view. Here is a summary of the `LightDisplay` class.

**LightDisplay Class**

- `LightDisplay(LightPanel panel)`

Each `LightDisplay` keeps track of a particular `LightPanel`, as shown in the following instance diagram.

```
LightDisplay
panel:  
<table>
<thead>
<tr>
<th>LightPanel</th>
</tr>
</thead>
<tbody>
<tr>
<td>LightSystem</td>
</tr>
<tr>
<td>LightPanel</td>
</tr>
</tbody>
</table>
```

```
LightDisplay
panel:  
<table>
<thead>
<tr>
<th>LightPanel</th>
</tr>
</thead>
</table>
```
**Time to Light the Lights**

Create a new Test class in a file called Test.java, and program it to run the following code.

```java
LightSystem system = new LightSystem();
LightDisplay d1 = new LightDisplay(new LightPanel());
LightDisplay d2 = new LightDisplay(new LightPanel());
```

Try turning on and off the lights.

Now, go through the following files and add good documentation to them. You need to add Javadoc comments above each class definition and method definition. You’ll want to add comments to any code in the classes that isn’t completely obvious.

```
LightDisplay.java
LightSystem.java
LightPanel.java
```

**A Photonic Relationship**

Find a partner (or several partners), and run a LightSystem (with a LightPanel or two) on one of your computers. Look up the IP address of that computer.

Do not run a LightSystem on the other computer. Instead, create and display a couple of LightPanels, passing the IP address and LightSystem.DEFAULT_PORT to the LightPanel constructor. If you are successful, you should each be able to turn on/off all the lights.

**In a Bit**

We wish to communicate binary data by turning on/off the lights. A LightPanel by itself won't let us broadcast and receive bits – all we are doing is turning on the voltage on this shared wire, which makes the lights come on. To send bits, we'll now create a new class called BitHandler. Each BitHandler will have a reference to its own LightPanel in an instance variable, and will use it to allow us to perform higher-level tasks. The BitHandler will be able to send bits on the wire using Manchester Encoding, as (briefly) discussed in class (section 6.16 of the text).

BitHandler's constructor should take no arguments, and should initialize its internal LightPanel to point to a new instance of the LightPanel class. (It would also be a good idea to create a second constructor and have it use an IP address and port to initialize the LightPanel.)
Since we'll rely on timing to distinguish one bit from another, it will be helpful to define the following method, which allows us to pause for a given number of milliseconds.

```java
public static void pause(int milliseconds)
{
    try {
        Thread.sleep(milliseconds);
    } catch(InterruptedException e) {
        // should never get here, but
        // let's print something just in case
        e.printStackTrace();
    }
}
```

Our BitHandler class should also keep track of a half delay—a number of milliseconds corresponding to half the time required to broadcast a single bit. We'll initialize this value to be 500 milliseconds. Later in the lab, we will see if we can lower this value and still communicate reliably.

We'll represent the binary values "0" and "1" by setting the light as follows over a full delay period (two half delays).

```
  on
 off

  on
 off

Binary "0"  Binary "1"
```

Write the methods broadcastZero and broadcastOne, which should use the LightPanel to turn the light on/off as depicted above. Now modify your Test class to create a BitHandler and have it broadcast the sequence "100". Draw a diagram to determine how long the light should be on/off, and test that the light's behavior matches your prediction.
**A Burst of Bits**

Next add a method called `broadcast` to your `BitHandler` class, which should take in a `String bits` consisting only of the characters "0" and "1", and should use your `broadcastZero` and `broadcastOne` methods to broadcast this sequence of digits. Once the final digits have been broadcast, be sure to switch the light off. When you've finished, test that your `broadcast` method works correctly.

**Seeing Things**

The `LightPanel's toString` method tells us its ID #. Create a `BitHandler toString` method to call the `LightPanel's toString` method and return that value.

Now look at `BitDisplay.java` — a simple GUI for showing and controlling your `BitHandler`.

Update your `Test` so that you create a new `BitDisplay` and use it to broadcast bits through your `BitHandler`.

Now, go through all the `Bit*.java` files and add good Javadoc documentation to these, as you did before with the `Light*.java` files.

**Light Pollution**

Next we would like to detect collisions—when two `BitHandlers` are broadcasting at the same time. Whenever a collision has been detected, we'll throw a `CollisionException`. Go ahead and define the very simple `CollisionException` class as follows, in `CollisionException.java`.

```java
public class CollisionException extends Exception {
    
}
```

Now modify your `BitHandler` class methods `broadcastZero()` and `broadcastOne()` so that they test if anyone has changed the light after each pause, and throw a `CollisionException` at such times.
You will now get an error message when you attempt to compile. Unlike
NullPointerException, ClassCastException, and
ArrayIndexOutOfBoundsException, our CollisionException class is not
a runtime exception (does not extend the RuntimeException class). We will
therefore need to handle it at compile time, by declaring or catching it. Go ahead and
declare that your broadcastZero and broadcastOne methods throw a
CollisionException. When you compile now, you will discover that you need to
declare that broadcast also throws this exception.

Compiling once more will reveal that the BitDisplay class now needs to handle this
exception. Add code to catch the exception and indicate that there was a collision by
setting the receiveField’s text to “Collision!”

Now test that you can broadcast a string of bits without the display reporting a collision.
Then try broadcasting two strings at once, and test that the display correctly identifies the
collision.

Now, go through the files again and make sure everything is documented well. All the
Javadoc explanations should be up-to-date, etc.

Revenge of the Finite State Machines

There is code in BitHandler’s run() method that repeatedly checks the “wire” to see
if the light has changed on or off, etc., and then interprets the results into bits. As it
examines the light, our handler may find itself in one of the following states.

<table>
<thead>
<tr>
<th>State Name</th>
<th>Light Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SILENT</td>
<td>off</td>
<td>No bits are being broadcast.</td>
</tr>
<tr>
<td>EXPECT_ZERO</td>
<td>on</td>
<td>We expect that we're about to read a &quot;0&quot;.</td>
</tr>
<tr>
<td>EXPECT_ONE</td>
<td>off</td>
<td>We expect that we're about to read a &quot;1&quot;</td>
</tr>
<tr>
<td>HALF_ZERO</td>
<td>off</td>
<td>We've seen the first half of a &quot;0&quot;, and we're waiting for the second half.</td>
</tr>
<tr>
<td>HALF_ONE</td>
<td>on</td>
<td>We've seen the first half of a &quot;1&quot;, and we're waiting for the second half.</td>
</tr>
<tr>
<td>GARBAGE</td>
<td>on</td>
<td>The light has been on for too long.</td>
</tr>
</tbody>
</table>
There are three situations that will cause the handler to change to a new state:

- The light just changed after a half delay.
- The light just changed after a full delay.
- The light has not changed for a timeout period of 3 half delays.

The code in `run()` implements the finite state machine, but does not call `notifyReceived()`. All you have to do is uncomment those lines to get it to work. Make that change now.

Here is a picture of the relationship between classes that now exists.

```
BitListener
  |      |      |      |      |
  v      v      v      v
BitHandler
  |      |      |      |      |
  |      |      |      |      |
  v      v      v      v
LightPanel
  |      |      |      |      |
  |      |      |      |      |
  v      v      v      v
LightSystem
```

Change your `Test` class to create a `LightSystem`, a `ListDisplay` (which takes a `LightPanel` as an argument) and a `BitDisplay` (which takes a `BitHandler` as an argument). Have a partner do the same (except the partner does not start up a `LightSystem`, but shares yours). Test that you can broadcast bits from one handler to the other. **NOTE NOTE NOTE: In order for this to work, all bit strings sent over the wire must start with a “0” bit.**

**The Speed of Light**

Now test how low you can set the `BitHandler`'s half delay value without sacrificing reliability. All machines using one `LightSystem` must have the same half delay for this to work.

**Those doing this class for Honors**
Implement Binary Exponential Backoff.

Submit Your Code

You need to submit your files to /home/cs/332/current/<yourid>/lab1/. Many of these files have only had documentation added to them. Submit two Test files – TestServer.java and TestClient.java. TestServer.java contains code to start up a LightSystem and a LightDisplay and BitDisplay. TestClient.java does not start up a LightSystem, but instead connects to a LightSystem on another machine (you may leave the IP address in your code that you used when you tested the code), and then starts up a BitDisplay as well.

Also, submit a README.txt file that tells me who you worked/partnered with, so that I can see if your code and that person’s code works together – i.e., because they use the same value for the half delay.

Grading Rubric:

20 pts total
  • 15 points for correctness
  • 5 points for thorough and correct documentation