COMS W1007 Homework 4

Submission instructions

All programs must compile and run on CUNIX to receive credit. Submit your electronic files via `http://courseworks.columbia.edu`. Post your archived submission directory to the CLASS FILES section in the appropriate homework directory. We prefer electronic submission of written portions, though you will not be penalized for paper submissions. (Do **not** print out your programs.) Include a separate README text file that explains exactly what each file in your submission is. Your submission should contain your code (.java files), written problems, and your README file. Do not include your compiled .class files or your compiled javadoc, since we will generate those from your code. Place all the files you want to submit into a submission directory with the following naming scheme.

`<your_uni>_hw<number>`

So if my UNI is `uni1234` am submitting homework 5, my directory would be `uni1234_hw5`. Archive your submission directory using zip, tar, or gzip format. In CUNIX, you can tar and gzip with the following command:

```
tar -czvf uni1234_hw5.tar.gz uni1234_hw5
```

and upload `uni1234_hw5.tar.gz` to courseworks. (You will probably need to first download the file to a local directory using an FTP program. See CUNIX tutorial for more info.)

Multiple Submissions: You can submit multiple times, but we will only consider the latest submission based on the timestamp in courseworks. Please give at least 1-2 minutes between two submissions so we can tell which is the newest submission.

Keep a pristine copy of your submission folder in case there is any submission error.

Written Problems

1. (6 points) **Based on Horstmann 7.24** Consider a class `Pair<F>`, defined as follows.

```
public class Pair<F>
{
    public Pair(F first, F second)
    {
        this.first  = first;
        this.second = second;
    }
```

```
    /**
        Set the first value of a Pair.
        @param value the value to be set
    */
    public void setFirst(F aValue)
    {
        first = aValue;
    }

    /**
        get the first value of a Pair.
        @return the first value.
    */
    public F getFirst()
    {
        return first;
    }

    /**
        Set the second value of a Pair.
        @param aValue the value to be set
    */
    public void setSecond(F aValue)
    {
        second = aValue;
    }

    /**
        get the second value of a Pair.
        @return the second value.
    */
    public F getSecond()
    {
        return second;
    }

    private F first;
    private F second;
}
```

Describe specifically how to make the `Pair` class cloneable and serializable. What needs to be added? Make sure to discuss the required type bounds.

2. (6 points) **Horstmann 7.26** Supply a method

```
public static < . . . > void putFirstLast(ArrayList< . . . > a, Pair< . . . > p)
```

in the `Utils` class that places the first and last element of `a` into `p`. Supply the appropriate type bounds.

3. (6 points) **Horstmann 8.10** The `RandomAccess` interface type has no methods. The `Set` interface type adds no methods to its superinterface. What are the similarities and differences between the functionality that they are designed to provide?

4. (6 points) Examine the `AbstractEdge` class from Horstmann's Graph Editor framework. Describe how the method `GetConnectionPoints` is an example of the TEMPLATE-METHOD pattern. Using the Simple Graph Editor as an example, what represents the Abstract class, the template method, the primitive operations and the concrete class(es)?

5. (6 points) **Horstmann 9.2** In the program in Section 9.1, is it possible that both threads are sleeping at the same time? That neither of the two threads is sleeping at a particular time? Explain. (This question is about `GreetingProducer` and `ThreadTester`.)

Programming Assignment

(70 points) For this homework assignment, you will use Horstmann's Graph Editor framework to build a simple physics simulation. You will create an editor where the nodes of a graph represent objects and the edges of the graph represent rubber bands connecting the objects.

Description

Make sure to also read the notes and hints below. Download Horstmann's graph editor framework. This will include the following files.

```
AbstractEdge.java, Edge.java, Graph.java, GraphFrame.java, GraphPanel.java,
Node.java, PointNode.java, ToolBar.java
```

You will also want to look at the Simple Graph Editor example we looked at in class. Create an instance of the Graph Editor Framework that allows the following four types of nodes:

- A rigid node that cannot move,
- A light node that has very small mass (and thus requires little force to accelerate),
- A medium node that has medium mass (and thus requires medium force to accelerate),
- A heavy node that has large mass (and thus requires heavy force to accelerate),

and one type of edge: a rubber band that applies elastic pulling force on its two end points. The differences between the nodes should be visually represented using either shading, drawn patterns, size, or combinations of these.

The simulated mechanics should be somewhat realistic. Each node must have a mass and a velocity (in both the x and y axes). Each rubber edge will exert equal force on its connected nodes, and that force will accelerate the nodes according to the standard $F = MA$ formula (force = mass $\times$ acceleration). See the notes below or look up Hooke's Law to compute how much force a rubber edge should apply. At each time step, you will need to compute the force and acceleration for both

x and y dimensions. Then apply the acceleration to change the velocity, and apply the velocity to change the position of each node. Allow the nodes to pass through each other so you do not need to compute collisions. Finally, you should add a friction force that slows the velocity by some fraction at each time step. You are free to choose this constant, but multiplying each velocity by 0.99 yielded realistic-looking slowdown.

You should add a button to the editor window that toggles between "Start Animation" and "Stop Animation". When the user clicks on "Start Animation", the nodes should start moving realistically as if the edges are elastic cords. Using the framework to make this work with a single graph will be worth 65 points out of the 70 available on the programming portion of this homework.

Save and Load

The remaining 5 points are designated for handling save and load functionality. The architecture of Horstmann's framework does not easily allow animation in conjunction with the save and load functionality (serialization). This is caused by the fact that the `Graph` object, which stores the data, is completely encapsulated inside the `GraphFrame`. Thus, when you load a new graph object, any `Timer` you had attached to the `Graph` decouples from the actual `Graph` being displayed.

To fix this, you have two options:

1. Disable saving and loading. This can be done by extending `GraphFrame` and overriding the save and load methods to tell the user that those operations are disabled.

2. To actually allow saving and loading, you will need to edit the framework. **Providing functioning save-load features should be the only reason you edit the framework.** You must clearly document any changes you make, and this step should feel very very wrong. Any undocumented changes to the framework will be penalized, so make sure you explain what you changed and why you changed it to resolve this save-load issue.

   I was able to do this by adding an accessor for the `Graph` in the `GraphFrame` class. Try to keep whatever changes you need to make as minimal as possible, and perhaps you will find a more elegant way to do this than I was able to.

Neither solution feels completely satisfying from a design point-of-view, but each is marginally better than having the animation break unexpectedly upon loading a new graph. This is why this fix is only worth 5 points. You must include a brief writeup of how you handled the save-load problem in your README.

Notes

- To approximate realistic physics, Hooke's Law of elasticity says that the force exerted by a spring is proportional to the displacement of the spring from its equilibrium position. In other words, if your rubber band's equilibrium position is 0 length, the force applied by the rubber band is proportional to its length. It is up to you to decide a constant for the springiness of the rubber band.

- Assume the edges have no mass. Assume angular momentum does not exist.

- A standard frame rate for computer animation is 60 frames per second, so your animation timer should update about once every 16 milliseconds.

- I recommend sticking to circular shapes for the node outlines. You can draw patterns inside the circles to indicate the rigid nodes, but circular shapes mean you can reuse the simple `getConnectionPoint` logic from the `CircleNode` class.
- Note that animation should not stop while the user is editing. If the user wants to connect nodes that are moving, your program should work fine. It can help to set velocity to zero whenever a node is dragged by the user.
- Since Horstmann's framework does not use the MVC pattern, and I have asked you not to modify the framework if possible, you obviously do not need to use the MVC pattern this time.
- Make sure to spend some time thinking about how the animation will work from an OOP point of view before you start coding. Think about what class will be responsible for computing the movements, storing velocity, storing the `Timer`, telling the view to `repaint()`, etc. Think about how to do this using the framework. Our standard tools (CRC, UML) will be helpful for organizing this.
- As always, finish the assignment before you get distracted by adding other features you might want, such as collisions, gravity, etc. Those will not earn you more class credit, so make sure you finish what is required first.
- A few sanity-check test-cases:
  - Connect a node to two rigid nodes. The node should eventually settle down to rest exactly in the middle of the two nodes.
  - Connect a chain of nodes with rigid nodes at the ends of the chain. Moving the rigid nodes should produce waves along the chain.
  - Connecting a heavy node to a light node should cause both nodes to move, with the light node whipping frantically around the heavy node. But the heavy node should be moving somewhat.

Process and Deliverables

For this programming assignment, you are only required to submit code that compiles via javac and javadoc. Also submit the code from Horstmann's framework. At least all public methods must have javadoc comments for any parameters and return values. You should include your code and a "README" file that briefly describes each included file, and can be used to indicate to us any notes that would be helpful for grading (such as any extra functionality you have added or were unable to finish in time).

You are encouraged to use the design tools we practiced on previous homework, but you do not need to submit them.

As always, to get full credit, your code must not crash for any reasonable user behavior. You may exit with an informative error message, but uncaught exceptions or crashes will lose some points.