Project 1 - Refactoring - Summary

Due Sep 20 by 11:59pm **Points** 20 **Submitting** on paper

River Crossing Project Summary

Due date: Sunday, September 20
Total Points: 50 points
Deliverables:

1. Source code [submit to Web-CAT]
2. PDF of step-by-step instructions of first refactoring [see **step-by-step assignment**]
3. JPG of class diagram [see **class diagram assignment**]
4. ZIP archive of *.java files [see **refactoring code assignment**]

There will be one Canvas "assignment" each for deliverables (2), (3), and (4)]

Resources: **Dr. K's GitHub repository** **(https://github.com/gregwk/RiverCrossing)**

Overview

In this project, we will refactor and extend an existing RiverCrossing application. The application includes a GUI that works correctly and a test that currently passes. Although the test passes, it is incomplete. Ultimately, we want our application to be able to implement different games like those found here:

https://www.transum.org/software/River_Crossing/
(https://www.transum.org/software/River_Crossing/)

Right now it only implements "Level 1" of that game.

Our code has some problems. First, although the code works for a specific case, the code is not in very good shape. There is a lot of duplicate code in the game-engine class, and there is some unnecessary code in the game-object class. The GUI is also in need of some serious refactoring. Once we get the refactorings out of the way, we will need to generalize the code so that we can not only handle the farmer, goose, and beans, but also handle the other scenarios, like the big and small robots, and perhaps even the monsters and munchkins.

Part 1 – Get Familiar with the Current Code

Getting the GitHub Project

Go to the **RiverCrossing project (https://github.com/gregwk/RiverCrossing)** on my GitHub site and download the ZIP file of the current code. Move it to the workspace of your favorite IDE and create a project using that code. For example, in Eclipse you just create a new Java project and give it the name "RiverCrossing" and as long as the RiverCrossing folder is in your workspace, Eclipse will detect that and use it for the project.

Once you have the files working in your IDE, do the following (note: you do not need to turn this in):

- Sketch out a class diagram for the application that includes all the classes in the "river" package and the relationships between them. Once you have done this, compare your diagram with the class diagram near the end of this project description. They should look similar.
- Run the GUI and see if you can get to a winning state and a losing state. Get a feel for what the application will and will not allow.
- Look at Wikipedia's entry on **Model-View-Controller (https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)**. Can you identify each of the 3 components in the current application?

Part 2 – Refactor the Code

Finish GameEngineTest

Complete the GameEngineTest based on the TODO comments in the test code. Run your test case and make sure all of the tests pass. Feel free to write more tests if you think you need them. Use the behavior of the GUI to help you understand what some of the tests should do.

Refactor GameObject

- Get rid of the GameObject subclasses – we don't want to limit ourselves to the farmer-oriented river-crossing game. Therefore, it doesn't make much sense to have specific subclasses devoted to these objects. Furthermore, the subclasses don't serve a lot of purpose - they are essentially just there to initialize fields in the game-object.
- Change the getter method "getSound" in GameObject so that it is based on a field, just like the "getName" and "getLocation" getter methods.
- We are not going to modify the name field of a game object after it is created. Therefore, get rid of the setter method for it. Remember to check that there are no references to a method (or field or constructor) before deleting it.
- Clean up the class. Get rid of any unused constructors. Make all fields private. Make the fields that are never assigned to final.

Refactor GameEngine

- Put the Location class into its own file. Note that there is an Eclipse refactoring that will do this for you.

- Change all variables named top, mid, bottom, and player to wolf, goose, beans, and farmer. This includes the fields in GameEngine and the constants in Item. Make sure you keep the proper style (for example, constants are always capitalized). Note that this is going to be a temporary step to help us talk about how the application works and to be consistent with the GUI. Once we generalize the application, we are going to rename them again.
- Declare an instance variable of type Map that maps Items to GameObjects. Initialize the map in the constructor using a HashMap or an EnumMap (your choice) and put the four game-objects in it. Use your map to simplify the methods getName, getLocation, getItem, and any other methods that can be simplified.
- Once you have created the map and simplified the code, you should no longer need the fields wolf, goose, beans, and farmer (since you can easily access them using the map). Make sure there are no references to them and then delete them. Note that you will still have to create appropriate game objects in the constructor so that you can add them to the map, but fix things so that you won't need the fields anymore.
- Change the method names *getName*, *getLocation*, and *getSound* to *getItemName*, *getItemLocation*, and *getItemSound*. This will make it clear that these methods just call through to the game object getters.
- The "current location" is basically just the boat location. Rename the field and the getter to use boat-location rather than current-location.
- Clean up the class. Make sure all fields and helper methods have private access. And make sure all fields that can be declared final *are* declared final.

Refactor GameEngineTest

- The class under test is GameEngine, so declare a private field "engine" of type GameEngine and initialize it in the @Before method (setUp). Use it to replace the local instances of the game engine.
- Rename the testObject method to testObjectCallThroughs. Instead of creating game-objects, test the call through methods in the game-engine class. For example, instead of testing whether farmer.getName is "Farmer", test whether engine.getItemName(Item.FARMER) is "Farmer".
- Write a helper method called transport that takes an item. The method should transport the item from one side of the river to the other. Use it to simplify some of the test cases in which an item is transported.
- Write a helper method called goBackAlone that just calls engine.rowBoat, and use it whenever the farmer is going back alone in the test cases.

Remember that through all of the above refactorings, you are only changing the design of the code, not its functionality. Therefore, your test cases should continue to pass after each refactoring and -- if you do them carefully -- during each step of your refactoring.

Part 3 – Prepare to Create the GameEngine Interface

The refactorings in this section have to do with the fact that we don't want the GUI to depend on anything that has to do with this particular implementation of the game engine. Recall that the web site we looked at had three different river-crossing puzzles: one with the farmer (like the one we have here), one with robots, and one with monsters. Now take a look at the GUI. We see:

- Lots of direct references to the farmer, wolf, goose, and beans through the use of Item.FARMER, Item.WOLF, etc.
- Lots of indirect references to farmer, wolf, goose, and beans through the colors we choose for the rectangles and the letters we place in the rectangles.
- Indirect references to farmer, wolf, goose, and beans through the variables names we choose.

Unfortunately, all of this will have to change. Fortunately, we can do much of this through fairly straightforward refactorings.

- Get rid of references to farmer, wolf, goose, and beans game-objects. The only place these are used anymore are in the constructor and in the gameIsWon and gameIsLost methods. For the constructor, you can use local variables to help you create the game objects, or just inline them. For the gameIsLost/Won methods, replace goose.getLocation() with getItemLocation(Item.GOOSE) and do the same for the other game objects. Run your tests to make sure they work. Check that there are no other references to wolf, goose, beans, and farmer, and then delete those fields.
- Move the Item class out of GameEngine and refactor the names as follows:
  - BEANS => ITEM_0
  - GOOSE => ITEM_1
  - WOLF => ITEM_2
  - FARMER => ITEM_3
- After you do this, the constants in the Item class will not be in numerical order. Move them around so that they are – this will become important when refactoring the GUI. In the game-engine class, **extract constants** for Item.ITEM_0, Item.ITEM_1, etc. Give them their former names of BEANS, GOOSE, etc. Do the same for the GameEngineTest. Run you tests and the app to make sure everything still works.
- Replace properties name and sound in GameObject (which are not used except in tests) with label and color. The label property is a string and the color property is a color. These properties will be used by the GUI to draw the rectangles. In the GameEngine, make the labels and colors the same colors that currently appear in the GUI. Much of this can be done with the renaming of fields and methods, but at some point you are going to have to change a String type to a Color and this will break things. Make sure you have an idea of what it will break before you do it. Then fix those items. Run the test and the GUI again to see if they still work.

As always, tidy up your code and make sure it is well-formatted.

Part 4 – Modify RiverGUI

The RiverGUI has a lot of redundant code. Somehow we've got to clean it up.

The view paints the rectangles on screen. The screen rectangle represent either items (things that can go into the boat) and the boat itself. If there are only 4 items, then there are only 5 rectangles on the screen. Therefore, we should be able to paint the screen with code that looks something like this.

```
@Override
public void paintComponent(Graphics g) {
    g.setColor(Color.GRAY);
    g.fillRect(0, 0, this.getWidth(), this.getHeight());
    paintItem(g, Item.ITEM_0);
    paintItem(g, Item.ITEM_1);
    paintItem(g, Item.ITEM_2);
    paintItem(g, Item.ITEM_3);
    paintBoat(g);
}

private void paintItem(Graphics g, Item item) { ... }
private void paintBoat(Graphics g) { ... }
private Rectangle getItemRectangle(Item item) { ... }
private Rectangle getBoatRectangle() { ... }

private void paintRectangle(Graphics g, Color color, String label, Rectangle rect) {
    // similar to paintStringInRectangle but now it creates
    // the rectangle with the specified color first
    // use rect.x, rect.y, rect.width, rect.height to get those values if needed
}
```

Notice that we have getters for the rectangles. Do we need setters?

For the controller, we should have something similar in that the code focuses on those 5 rectangles.

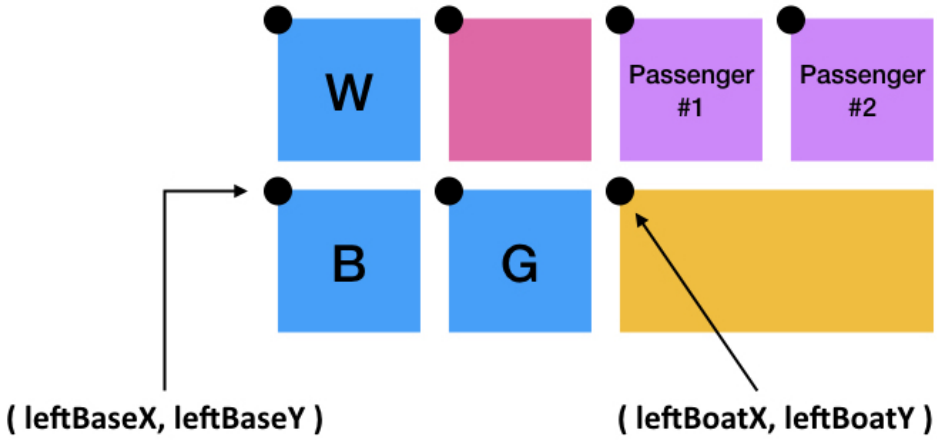
```
@Override
public void mouseClicked(MouseEvent e) {
    if (item0Rectangle.contains(e.getPoint())) {
        /* respond to click of Item.ITEM_0 */
    } else if (item1Rectangle.contains(e.getPoint())) {
        /* respond to click of Item.ITEM_1 */
    } else if (item2Rectangle.contains(e.getPoint())) {
        /* respond to click of Item.ITEM_2 */
    } else if (item3Rectangle.contains(e.getPoint())) {
        /* respond to click of Item.ITEM_3 */
    } else if (boatRectangle.contains(e.getPoint())) {
        /* respond to click of boat */
    }
    repaint();
}
```

Part 4 Suggestions

In the Q&A from February 6th, we implemented paintRectangle, and talked about how we could use that to refactor and simplify the existing code. But even if we replace filling rectangles and drawing labels

with "paintRectangle", the code is still a bit of a mess because we have way too many conditions. Remember that the goal is to just paint 5 rectangles – one for each item and one for the boat.

To do this, there will need to be some logic in paintItem. Our rectangle will depend on where the item is (left bank, boat, or right bank). It may help to use offsets. For example:



From the 2 (x, y) coordinates in the picture, I can derive any rectangle on the left side of the river.

```
// beans on left shore
rect = new Rectangle(leftBaseX, leftBaseY, 50, 50);
// farmer on left shore
rect = new Rectangle(leftBaseX + 60, leftBaseY - 60, 50, 50);
// 1st passenger on boat
rect = new Rectangle(leftBoatX + 60, leftBaseY, 50, 50);
```

To further simplify the items on the shore, you could declare dx and dy arrays with offsets for items 0 through 3:

```
int[] dx = { 0, 60, 0, 60 };
int[] dy = { 0, 0, -60, -60 };
```

Then, since ITEM is an enum type, you could use the getValue to return the index for the appropriate item:

```
int index = item.ordinal();
```

If item is ITEM_0, this will return 0; if item is ITEM_1, this will return 1, etc. Therefore, if you knew an item was on the left shore, it's rectangle would be:

```
rect = new Rectangle(leftBaseX + dx[item.ordinal()],
                     leftBaseY + dy[item.ordinal()], 50, 50);
```

To obtain all possible rectangles you would also need (x, y) coordinates for the boat and a base rectangle on the right shore.

Part 5 – The GameEngine Interface

- Rename getCurrentLocation to getBoatLocation.
- Rename GameEngine to FarmerGameEngine.
- Extract a GameEngine interface from FarmerGameEngine. Select all public methods *except* the no-argument unloadBoat.
- Rename your GameEngineTest to FarmerGameEngineTest.
- Make sure your GameEngineTest and your RiverGUI classes *declare* GameEngines but initialize using FarmerGameEngines.
- Make any changes necessary to get your tests and your application to work (like replacing occurrences of unloadBoat() with unloadBoat(Item)).
- Take a look at the GameEngine interface below. Specifically, look at the Javadocs and ensure that your FarmerGameEngine class behaves as specified in the Javadocs.

```
package river;

import java.awt.Color;

public interface GameEngine {

	/**
	 * Returns the label of the specified item. This method may be used by a GUI
	 * (for example) to put the label string inside of a rectangle. A label is
	 * typically one or two characters long.
	 *
	 * @param item the item with the desired label
	 * @return the label of the specified item
	 */
	String getItemLabel(Item item);

	/**
	 * Returns the color of the specified item. This method may be used by a GUI
	 * (for example) to color a rectangle that represents the item.
	 *
	 * @param item the item with the desired color
	 * @return the color of the specified item
	 */
	Color getItemColor(Item item);

	/**
	 * Returns the location of the specified item. The location may be START,
	 * FINISH, or BOAT.
	 *
	 * @param item the item with the desired location
	 * @return the location of the specified item
	 */
	Location getItemLocation(Item item);

	/**
	 * Returns the location of the boat.
```

```
     *
     * @return the location of the boat
     */
    Location getBoatLocation();

    /**
     * Loads the specified item onto the boat. Assuming that all the
     * required conditions are met, this method will change the location
     * of the specified item to BOAT. Typically, the following conditions
     * must be met: (1) the item's location and the boat's location
     * must be the same, and (2) there must be room on the boat for the
     * item. If any condition is not met, this method does nothing.
     *
     * @param item the item to load onto the boat
     */
    void loadBoat(Item item);

    /**
     * Unloads the specified item from the boat. If the item is on the boat
     * (the item's location is BOAT), then the item's location is changed to
     * the boat's location. If the item is not on the boat, then this method
     * does nothing.
     *
     * @param item the item to be unloaded
     */
    void unloadBoat(Item item);

    /**
     * Rows the boat to the other shore. This method will only change the
     * location of the boat if the boat has a passenger that can drive the boat.
     */
    void rowBoat();

    /**
     * True when the location of all the game items is FINISH.
     *
     * @return true if all game items of a location of FINISH, false otherwise
     */
    boolean gameIsWon();

    /**
     * True when one or more implementation-specific conditions are met.
     * The conditions have to do with which items are on which side of the
     * river. If an item is in the boat, it is typically still considered
     * to be on the same side of the river as the boat.
     *
     * @return true when one or more game-specific conditions are met, false
     * otherwise
     */
    boolean gameIsLost();

    /**
     * Resets the game.
     */
```

```
        void resetGame();
}
```

The requirements of this interface may change some functionality. For example, now you can load 2 non-farmer items into the boat (but you can't row it since only the farmer can drive). Therefore, if you created a passenger field from a previous refactoring, remove it. Instead, create a method that checks how many items are in the boat. When you load an item into the boat, see if there are already two items in the boat. If there are, don't load the item.

Part 6 - The RobotGameEngine

Implement a RobotGameEngine class that makes the game behave like the robot game from the river-crossing web page given above. So that I can test the RobotGameEngine also, please include the following constants (similar to FarmerGameEngine)

- SMALLBOT_1;
- SMALLBOT_2;
- TALLBOT_1;
- TALLBOT_2;

Part 7 - Deliverables

- (**Web-CAT submission**) Add more tests in FarmerGameEngineTest so that they cover all statements in FarmerGameEngine and submit your project to Web-CAT.
- (**ZIP file upload to Canvas**) Copy all your *.java file (only source files, do not include *.class files) to a folder. ZIP the folder and upload it to Canvas.
- (**PDF upload to Canvas**) Consider the second refactoring for GameObject (change getSound so it's based on a field). Below is an example of a step-by-step description of this refactoring. Notice that each step involves only one field, method, or statement, and that **after each step, you should be able to successfully compile the project and run the unit tests**. Write a step-by-step description of the first refactoring of GameObject (get rid of GameObject subclasses) and submit it to Canvas as a PDF file.
  1. Add a field named sound of type String.
  2. Add a GameObject constructor that takes all fields as arguments. It should be like the existing one, but it should include an argument for the sound field.
  3. Find all occurrences of the original GameObject constructor. There should be four in GameEngine and four in GameEngineTest.
  4. In GameEngine, change the GameObject constructor for wolf so that it uses the new constructor, with "Howl" passed in as the sound argument.
  5. Do step 4 for the wolf game-object in the GameEngineTest.
  6. Do steps 3 and 4 for the goose, beans, and farmer game-objects using "Honk", empty-string, and empty-string for sound arguments respectively.

7. Ensure that the old GameObject constructor is no longer used and get rid of it.
8. Comment out the existing code in getSound and replace it with “return sound;”
9. Assuming everything still works, delete the code you just commented out.

- (**JPG upload to Canvas**) See **class-diagram assignment**.