CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2015 – Clevenger/Muyan

Assignment #2: Design Patterns and GUIs

Due Date: Tuesday, March 17th [2½ weeks]

Introduction

For this assignment you are to extend your game from Assignment #1 (A1) to incorporate several important *design patterns*, and a *Graphical User Interface* (GUI). The rest of the game will appear to the user to be similar to the one in A1, and most of the code from A1 will be reused, although it will require some modification and reorganization.

An important goal for this assignment will be to reorganize your code so that it follows the *Model-View-Controller (MVC) architecture*. If you followed the structure specified in A1, you should already have a "controller": the **`Game`** class containing the **`getCommand()`** method along with methods to process the various commands (most of which call methods in **`GameWorld`**). The **`GameWorld`** class becomes the "data model", containing the collection of game objects and other game state information. You are also required to add two classes acting as "*views*": a <u>score</u> view which will be graphical, and a <u>*map*</u> view which for now will retain the text-based form generated by the 'm' command in A1 (in A3 we will replace the text-based map with an interactive graphical map).

Most of the keyboard commands from A1 will be replaced by GUI components (menus, buttons, etc.). Each such component will have an attached "command" object, and the command objects will perform the same operations as previously performed by the keyboard commands.

The program must use appropriate interfaces for organizing the required design patterns. The following design patterns are to be implemented in this assignment:

- *Observer/Observable* – to coordinate changes in the model with the various views,
- *Iterator* – to walk through all the game objects when necessary,
- *Command* – to encapsulate the various commands the player can invoke,
- *Singleton* – to insure that only a single copy of each command object exists,
- *Strategy* – to control movement for additional (non-player) cars, and

Model Organization

The "game object" hierarchy will be the same as in A1. **`GameWorld`** is to be reorganized (if necessary) so that it contains a collection of *game objects* implemented in a single collection data structure. The game object collection is to implement the **`Collection`** interface and provide for obtaining an **`Iterator`**. All game objects are to be contained in this *single* collection data structure[1]. The iterator for the collection returns one game object for each call to its **`getNext()`** method.

[1] If you did not implement your game object collection this way in Asst #1 you must change it for this assignment.

The model is also to contain the same game state data as A1, plus a new state value: a flag indicating whether *Sound* is ON or OFF (described below).

Views

A1 contained two functions to output game information: the 'm' key for outputting a "map" of the game objects in the **`GameWorld`**, and the 'd' key for outputting current *game state data* (time, lives, etc.) Each of these two operations is to be implemented in this assignment as a ***view*** of the **`GameWorld`** model. To do that, you will need to implement two new classes: a ***`MapView`*** class containing code to output the map, and a ***`ScoreView`*** class containing code to output the current score and other state information.

To implement this, **`GameWorld`** should be defined as an *observable*, with two *observers*– **`MapView`** and **`ScoreView.`** Each of these should be "registered" as an observer of **`GameWorld`**. When the controller invokes a method in **`GameWorld`** that causes a change in the world (such as a game object moving, or a new object being added to the world) the **`GameWorld`** notifies its observers that the world has changed. Each observer then automatically produces a new output view of the data it is observing – the game world objects in the case of **`MapView`**, and a description of the game state values in the case of **`ScoreView`**. The **`MapView`** output for this assignment is unchanged from A1: text output on the console showing all game objects which exist in the world. However, the **`ScoreView`** is to present a *graphical* display of the game state values (described in more detail below).

In order for a view (observer) to produce the new output, it will need access to some of the data in the model. This access is provided by passing to the observer's **`update()`** method a parameter that is a reference back to the model. This has the undesirable side-effect that *the view has access to the model's mutators*. We will see later how to fix this problem.

Non-Player Cars (NPCs)

The game initialization code should create and add to the game world three additional instances of Car (so that there are now four cars – the player's car plus three "non-player cars". Each NPC is to have an initial location which is "near" the first pylon, but not exactly *at* the first pylon; instead, each NPC should be at least several car lengths away from the first pylon (this is to make sure that the NPCs are not colliding with the player's car to start with).[2]

When an NPC collides with the player's car, the player's car sustains *damage* just as described in A1 – including that if the car sustains so much damage that it can no longer move the player loses a life, and the world is reinitialized. NPCs also sustain damage when they collide, but you should initialize them with much higher values so that they can compete longer (consider NPCs to be "armored" so that they can sustain more damage).

NPCs will be controlled by separate pieces of code called *strategies* which can be altered using a new "*Change Strategies*" command; this is described under "Strategy Pattern", below.

[2] You may if you wish apply additional constraints to the NPC's initial locations. For example, you could compute a "starting line" intersecting the first pylon and perpendicular to the line from the first to the second pylon, and require that the initial position of each NPC be behind that line in relation to the position of the second pylon. This would make for a much fairer game. However, this is not a requirement; the only requirement is that initial NPC locations be such that the NPCs do not overlap the player's car.

GUI Operations

The program is to be modified so that the top-level **`Game`** class extends ("is-a") **`JFrame`** representing the GUI for the game. The **`JFrame`** should be divided into three areas: one for commands, one for "score" (game state) information, and one for the "map" (which will be an empty **`JPanel`** for now but in subsequent assignments will be used to display the map in graphical form). See the sample picture at the end. Note that since user input is via GUI components, flow of control will now be event-driven and there is no longer any need to invoke a "**`play()`**" method – once the **`Game`** is constructed it simply waits for user-generated input events. Note however that it is still a requirement to have a "**`Starter`**" class containing the **`main()`** method.

In A1 your program prompted the player for commands in the form of keyboard input characters. For this assignment the code which prompts for commands and reads keyboard command characters is to be discarded. In its place, commands will be input through three different mechanisms: on-screen buttons, key bindings, and menu items. Some commands will be invokable only via a single one of these mechanisms, while others will be able to be invoked in multiple ways. You are to create *singleton command objects* (see below) for each of the commands from A1 plus the new *change strategies* command, and attach the objects as commands to various combinations of invokers as shown in the following table. An 'x' in a column indicates that the command in that row is to be able to be invoked by the mechanism in the column header. Further details regarding this are given below.

Command	KeyBinding	MenuItem	Button
accelerate	x		
brake	x		
left turn	x		
right turn	x		
collide with NPC			x
collide with **p**ylon			x
fuel pickup		x	x
collide with Bird (**g**)			x
enter oil slick			x
e**x**it oil slick			x
add **o**il slick	x	x	
new colors		x	
tick	x		x
quit	x	x	x
Change Strategies	x		x

As shown in the table, some commands will be invoked by GUI buttons. You will need a class that extends **`JPanel`** and contains these buttons. You are to create the appropriately labeled buttons, add them to the panel, and add the panel as a component of the game (**`JFrame`**). Each button is to have an appropriate command object attached to it, so that when a button gets pushed it invokes the "action performed" method in the corresponding command object (the "execute" method in terms of the Command pattern), which contains the code to perform the command.

Most commands execute exactly the same code as was implemented in A1. One exception is the '**p**' (*player hit **p**ylon*) command. Previously this command consisted of three characters typed on the keyboard: ***pxx***, where ***xx*** is the number of the pylon the player car has driven over. Now, the '**p**' button command is to use the static method **`JOptionPane.showInputDialog`** to display a modal input dialog box which allows the user to enter the value ***xx***. When the user clicks "OK", the typed value is returned as a String; at this point it can be processed similarly to the way your code from A1 handled the input string. For example:

```
String inputString = JOptionPane.showInputDialog("Please input a pylon number");
int pylonNum = Integer.parseInt(inputString);
```

Note that **`showInputDialog()`** throws **`NumberFormatException`** if the user inputs a non-numeric value; your program should handle this gracefully. Note also that **`showInputDialog()`** has additional forms which allow you to manage the form of the displayed dialog in more detail; this would be nice but is not required.

The other command which must operate slightly different in this assignment is '**c**' (player's car **c**ollided with another car). In A1, this command just increased the damage level of the player's car. Now, the command is to also add damage to one of the non-player cars: the '**c**' command code should choose one NPC and increase that NPC's damage as well. To make the game more fair you should really alternate between NPCs when this command is entered, but it is acceptable to use any algorithm (including choosing an NPC randomly, or always choosing the same NPC). We will change how this works in the next assignment, so any approach in A2 is fine.

Some commands will be invoked using the Java *KeyBinding* mechanism. For example, your program is to arrange that the *up arrow*, *down arrow, left arrow,* and *right arrow* keys invoke command objects corresponding to the code previously executed when the 'a', 'b', 'l', and 'r' keys (for controlling the player's car) were entered, respectively. Thus, up-arrow performs acceleration, down-arrow performs braking ("slowing down"), left-arrow turns the steering wheel left, and right-arrow turns the steering wheel right. Note that using key bindings means that whenever a key is pressed, the program will *immediately* invoke the corresponding action (without the user pressing the Enter key).

In addition, the program should bind the "space" key to the new "*change strategies*" operation (see below), as well as binding the *add **o**il slick*, ***t**ick*, and ***q**uit* commands to the 'o', 't', and 'q' keys respectively. If you want, you may also use key bindings to map any of the other command keys from A1, but only the ones listed here are required.

Some commands will be invoked using menu items. Your GUI should contain at least two menus: "*File*" and *"Commands"*. The *Commands* menu should contain one item for each of the following commands from A1: '**o**' (add **o**il slick), '**f**' (pick up **f**uel can), and '**n**' (**n**ew colors). (Give each of these commands appropriate names on the menu).

The "File" menu should contain at least items "New", "Save", "Sound", "About", and "Quit". Only the *"Sound", "About"* and *"Quit"* items need to do anything for this assignment. (Menu items which do nothing else should at least display a message on the console indicating that they were invoked). The Sound menu item should include a **`JCheckBoxMenuItem`** showing the current state of the "sound" attribute (in addition to the attribute's state being shown on the

ScoreView GUI panel as described below). Selecting the Sound menu item check box should set a boolean "sound" attribute to "ON" or "OFF", accordingly.

The "About" menu item is to display a **JOptionPane** dialog box (see **JOptionPane.showMessageDialog()**) giving your name, the course name, and any other information (for example, the version number of the program) you want to display. "Quit" should invoke the "quit command" object (the *same* object invoked by the 'q' key binding and by the "Quit" GUI Button). It should prompt graphically for confirmation and then exit the program; **JOptionPane.showConfirmDialog()** can be used for this.

The **ScoreView** class should extend **JPanel** and contain **JLabel** components for each of the game state values from A1 (time, lives, etc.), plus the *new* attribute *"Sound"* with value either ON or OFF.[3] As described above, **ScoreView** must be registered as an observer of **GameWorld**. Whenever any change occurs in **GameWorld**, the **update()** method in its observers is called. In the case of the **ScoreView**, what **update()** does is update the contents of the **JLabels** displaying the game state values in the **JPanel** (use **JLabel** method **setText(String)** to update the label). Note that these are exactly the same game state values as were displayed previously (with the addition of the "Sound" attribute); the only difference now is that they are displayed *graphically* in a **JLabel**.

Although we cover most of the GUI building details in the lecture notes, there will most likely be some details that you will need to look up using the online Java documentation. It will become increasingly important that you familiarize yourself with and utilize this resource.

Command & Singleton Design Patterns

Invoking a command should execute a corresponding *command object*, whether the command is invoked from a button, a keystroke, or a menu item. Further, it is a requirement that there be only a *single instance* of each command object in the program. Said another way, it is a requirement that commands be implemented using the *Singleton* and *Command* design patterns.

The recommended approach for implementing command classes is to have each Command extend the Java **AbstractAction** class, as shown in the course notes. Code to perform the command operation then goes in the command's **actionPerformed()** method. Java **AbstractButton** subclasses (for example, **JButton** and **JMenuItem**), are automatically able to be "holders" for such command objects; **AbstractButtons** have a **setAction()** method which allows inserting a command (**Action**) object into the "command holder" (button, menu item, etc.). **AbstractActions** automatically become listeners when added to an **AbstractButton** via **setAction()**, and the specified **Action** is automatically invoked when the component is activated (for example, when a button is pressed), so if you use the Java facilities correctly then this particular observer/observable relationship is taken care of automatically.

The game initialization code should create a *single* instance of each command object (for example, an "*accelerate player's car*" command object, a "*player's car entered oil slick*" command object, etc.), then insert the command objects into the appropriate command holders (control panel buttons, menu items, and/or Key Binding **ActionMaps**) using methods like

[3] In this version of the game there will not actually be any sound; just the state value ON or OFF (a Boolean attribute that is *true* or *false*). We'll see how to actually add sound later.

`setAction()` (for GUI components) or **`put()`** (for **`ActionMaps`**). Note that some commands can be invoked in multiple ways (e.g. from a keystroke and from a button); it is a requirement that only *one* Command object be implemented for each command and that the *same* Command object be invoked from each different Command Holder. As a result, it is important to make sure that nothing in any command object contains knowledge of *how* the command was invoked.

As discussed in class, some command objects may need to be provided with *targets*. For example, a command might have the **`GameWorld`** as its target if it needs access to game object data. You could implement the specification of a command's target either by passing the target to the command constructor, or by including a "**`setTarget()`**" method in the command and then passing the **`GameWorld`** to the command object after it is created.

Iterator Design Pattern

The game object collection in the **`GameWorld`** must be accessed through an appropriate implementation of the Iterator/Collection design pattern. That is, any routine that needs to process the objects in the collection must not access the collection directly, but must instead acquire an iterator on the collection and use that iterator to access the objects. You may use the interfaces discussed in class for this, or the more complex Java versions, or you may develop your own. Note however the following implementation requirement: the Game Object collection must provide an iterator *completely implemented by you*. Even if the data structure you use has an iterator provided by Java, *you must implement an iterator yourself.*

Strategy Design Pattern

Non-Player Cars move around the world according to their current *strategy*. That is, the NPC *move()* method uses the current strategy of that car to determine how to move, and the strategy can be changed on the fly (i.e. while the program is running). You are to implement at least two different NPC movement strategies: the first strategy causes the NPC to move directly toward the next pylon (that is, the pylon with a sequence number one higher than the one it most recently encountered); the second strategy causes the NPC to update its heading every time it is told to move so that the heading points toward the location of the player's car (in other words, its strategy is to play "Destruction Derby" by trying to ram into the player's car). You may also implement additional strategies if you like, although this is not required (for example, you might have a strategy where an NPC simply moves in circles, or a strategy where it moves back-and-forth between two pylons).

The program must use the Strategy design pattern to define the strategy for each NPC. When an NPC is created it should be assigned a strategy chosen from among the available strategies. It is a requirement that NPCs may not all have the same initial strategy; other than that you may assign initial strategies however you choose. Like all GameObjects, NPCs should include a *toString()* method; the NPC *toString()* should return a string which includes an identification of that NPC's current strategy.

When the human player invokes the "*switch strategy*" command (which happens when either the space bar or the appropriate GUI button is pressed), the game should cause all NPCs to *switch to a different movement strategy*. Switching strategies is to be done by invoking the strategy Context's *setStrategy()* method. You may choose the algorithm which determines the new NPC strategy, as long as every NPC acquires a new strategy each time the "switch strategy" command is invoked. Additionally, you should arrange that as a side effect of “switching strategies”, each NPC gets a new “highest pylon reached” value (otherwise, NPCs will never move beyond Pylon #2 since we have no command analogous to “***pxx***” which applies to NPCs).

Additional Notes

- Recall that there are two approaches which can be used to implement the Observer pattern: defining your own **`Observable`** interface, or extending the Java **`Observable`** class. Because of things we will do later, it will be more complicated if you choose to extend the Java **`Observable`** class; it is recommended that you build your Observable objects by having them implement a user-defined **`Observable`** interface such as the one described in class, even though it requires implementing the code yourself to keep track of the list of Observers.
- Make your initial GUI frame reasonably large, but small enough to work on most screens (a size of something like 1000x800 is usually a decent starting point).
- Note that all game data manipulation by a command is accomplished by the command object invoking appropriate methods in the **`GameWorld`** (the *model*). Note also that *every* change to the **`GameWorld`** will invoke *both* the **`MapView`** and **`ScoreView`** observers – and hence generate the output formerly associated with the ‘d’ and ‘m’ commands. This means you do not need the ‘d’ or ‘m’ commands; the output formerly produced by those commands is now generated by the observer/observable process.
- Programs must contain appropriate documentation as described in A1 and in class.
- Since ‘tick’ causes movable objects to move, every ‘tick’ results in a new map view being output (because each tick changes the model). However, it is *not* the responsibility of the ‘tick’ command code to produce this output; it is an effect of the observable/observer pattern. Note this is not the only command that generates output as a side effect - verify that your program correctly produces updated views automatically whenever appropriate.
- The mechanism for using Java “Key Bindings” is to define a **`KeyStroke`** object for each key and insert that object into the **`WHEN_IN_FOCUSED_WINDOW`** input map for the game world map display panel while also inserting the corresponding **`Command`** object into the panel’s **`ActionMap`**. Java **`KeyStroke`** objects can be created by calling **`KeyStroke.getKeyStroke()`**, passing it either a single *character* or else a *String* defining the key (string key definitions are exactly the letters following “**`VK_`**” in the Virtual Key definitions in the **`KeyEvent`** class). For example:

  ```
  KeyStroke bKey = KeyStroke.getKeyStroke(“b”);
  KeyStroke downArrowKey = KeyStroke.getKeyStroke(“DOWN”);
  ```

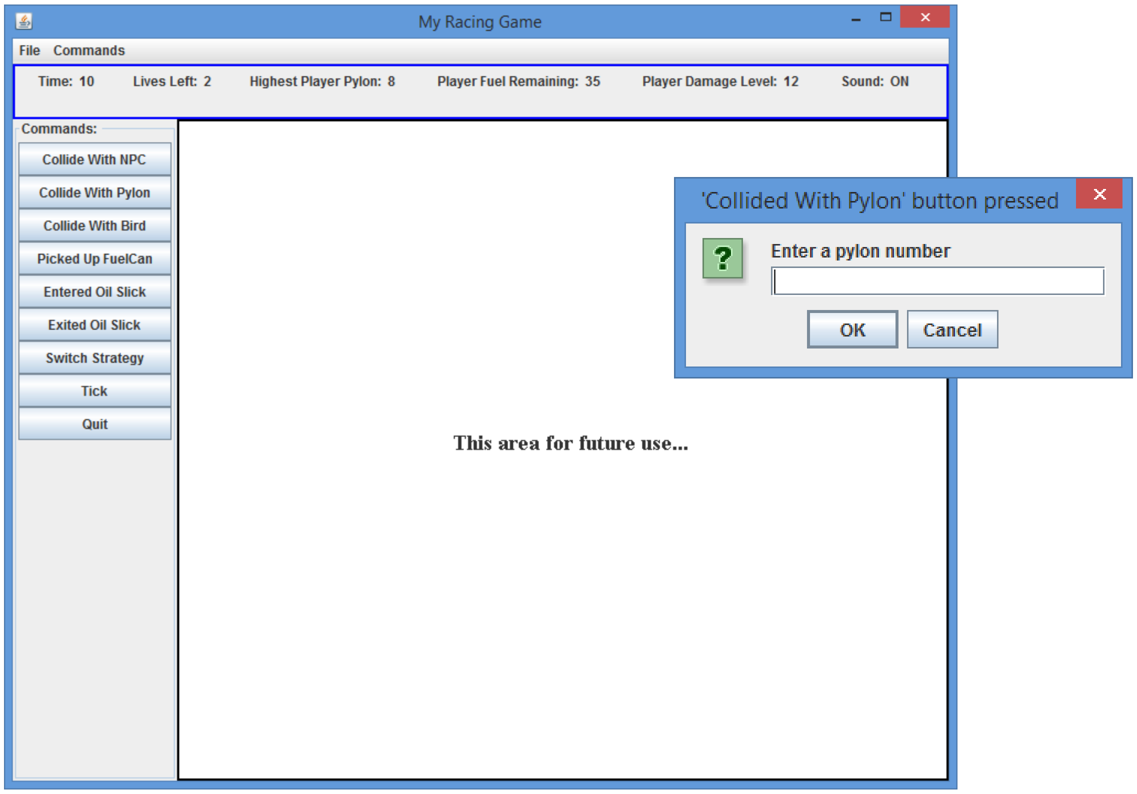
- **`JComponents`** such as **`JButton`**s automatically apply a key binding for the SPACE key when they are created. This is what causes the normal Java GUI application behavior where pressing SPACE automatically invokes the component which has focus. However, this behavior is detrimental in this kind of program: if you click on a button (to invoke its action), that button acquires focus; if you then hit the SPACE bar it will automatically invoke the button again (because the button has focus and contains a key binding for SPACE). To suppress this behavior, so that hitting SPACE invokes the Action you assign in the **`WHEN_IN_FOCUSED_WINDOW`** input map and *not* also the action in the button with focus, you need to *remove* the button’s default binding for SPACE. To do that, you assign the special action name “none” to the *button’s* input map. That is, for each **`JButton`** you create, you should execute the following statement:

```
    button.getInputMap().put(KeyStroke.getKeyStroke("SPACE"),"none");
```

- You may <u>not</u> use a “GUI Builder” tool for this assignment (if you don’t know what a “GUI Builder” is, don’t worry about it; we’ll be covering it later in the semester).
- As before, your program for this assignment must be contained in a single Java *package*. The requirements are the same as for the previous assignment, except that this time the name of the package must be exactly “**`a2`**”.
- All functionality from the previous assignment must be retained unless it is explicitly changed or deleted by the requirements for this assignment.
- You should develop a *UML diagram* including not only the major fields and methods required but also the interfaces and the relationships between classes (for example, Observer/Observable). This will be particularly useful in helping you understand what modifications you need to make to your code from A1.
- If you wish to use your own “Command” interface structure, rather than the built-in Java **`AbstractButton`** class, you may. However, if you do, you will have to implement the “Command Holder” facility yourself, and take care of setting up the listener and callback mechanisms. Using the built-in Java structures is likely to be considerably simpler.
- Although the **`MapView JPanel`** is empty for this assignment, you should put a border around it and install it in the frame, to at least make sure that it is there. You may if you wish also enhance your **`MapView`** to include the text output generated by the changes in the gameworld map; for example, you could add a **`JTextArea`** object to the **`MapView`** panel and use method **`JTextArea.append()`** to add text to it. However, *the map text must also still appear on the console*.

Sample GUI

The following shows an example of what your game GUI might look like. Notice that it has the required menus, a control panel on the left containing all the required buttons, a **`ScoreView`** panel near the top showing the current game state information, and an (empty) bordered MapView panel in the middle for future use. The game title displays at the top. The user has just hit the “Collide With Pylon” button, displaying the required input dialog.



Time: 10 Lives Left: 2 Highest Player Pylon: 8 Player Fuel Remaining: 35 Player Damage Level: 12 Sound: ON

'Collided With Pylon' button pressed

Enter a pylon number

This area for future use...

Deliverables

Submitting your program requires the same three steps as for A1: (1) create a single ZIP file containing your UML diagram in PDF form, your source code (.java) files, and your compiled code (.class) files; (2) *verify your submission* by unzipping your ZIP file to an empty folder and executing the program using the command **java a2.Starter** ; and then (3) upload your ZIP file to SacCT.

Also include in your ZIP a text file called “readme” in which you describe any changes or additions you made to the assignment specifications (for instance, if you bind any additional keys or add extra buttons, explain those in your readme file). Also include a description of any additional strategies in your readme file.

Be sure to keep a *backup copy* of all submitted work. All submitted work must be *strictly* your own.

Java Notes

Below is one possible organization for your MVC code for A2. Note that this organization is based on the assumption of using interfaces for both the Observer and the Observable. If you choose instead to use Java's Observable class, you will have to modify the code to account for that. Note also that this pseudocode shows one way of registering Observers with Observables: having the controller handle the registration. It is also possible to have each Observer handle its own registration in its constructor (examples are shown in the course notes). You may use either approach in your program.

```
public class Game extends JFrame {

      private GameWorld gw;
      private MapView mv;             // new in A2
      private ScoreView sv;           // new in A2

      public Game() {
            gw = new GameWorld();     // create "Observable" GameWorld
            gw.initLayout();          // initialize world
            mv = new MapView();       // create an "Observer" for the map
            sv = new ScoreView();     // create an "Observer" for the game state data
            gw.addObserver(mv);       // register the map Observer
            gw.addObserver(sv);       // register the score observer

            // code here to create menus, create Command objects for each command,
            // add commands to Command menu, create a control panel for the buttons,
            // add buttons to the control panel, add commands to the buttons, and
            // add control panel, MapView panel, and ScoreView panel to the frame

            setVisible(true);
      }
}
public interface IGameWorld {
     //specifications here for all GameWorld methods
}

public class GameWorld implements IObservable, IGameWorld {
     // code here to hold and manipulate world objects, handle
     // observer registration, invoke observer callbacks, etc.
}

public class GameWorldProxy implements IObservable, IGameWorld {
     // code here to accept and hold a GameWorld, provide implementations
     //  of all the public methods in a GameWorld, forward allowed
     //  calls to the actual GameWorld, and reject calls to methods
     //  which the outside should not be able to access in the GameWorld.
}
public class MapView extends JPanel implements IObserver {

     public void update (Observable o, Object arg) {
       // code here to output current map information (based on
       // the data in the Observable) to the console
     }
}

public class ScoreView extends JPanel implements IObserver {
     public void update (Observable o, Object arg) {
       // code here to update JLabels from data in the Observable
     }
}
```

JC:PM / jc
2/26/2015