CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2017 (Credits to Prof. Clevenger)

Assignment #2: Design Patterns and GUIs

Due Date: Friday October 20th [3 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 (some of which call methods in **`GameWorld`** when access to the game objects is needed). The **`GameWorld`** class becomes the "data model", containing a collection of game objects and other game state values (more details below). You are also required to add two classes acting as "*views*": a *points* view which will be graphical, and a *map* view which 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.) which generate events. Each such event will have an associated "command" object, and the command objects will perform the same operations as previously performed by the keyboard commands in A1.

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

- *Observer/Observable* – to coordinate changes in the data with the various views,
- *Iterator* – to walk through all the game objects when necessary,
- *Command* – to encapsulate the various commands the player can invoke,
- *Proxy* – to insure that *views* cannot *modify* the game world.

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 (moveable and fixed) 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 also contains a few other important pieces of game state data: the current score, number of missiles, elapsed time, and a flag indicating whether Sound is ON or OFF (described later).

Views

A1 contained two functions to output game data: the "m" key for outputting a "map" of the objects in the **GameWorld**, and the "p" key for outputting the "points" (score) information. Each of these operations is to be implemented 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 ***PointsView*** class containing code to output the points and other state information.

GameWorld should therefore be defined as an *observable*, with two *observers* – **MapView** and **PointsView.** Each view 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 missile being added) 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; that is, the game world objects in the case of **MapView** and a description of the point values and related data in the case of **PointsView**. The **MapView** output for this assignment is unchanged from A1: text output on the console. However, the **PointsView** is to present a *graphical* display of the game state values (described in more detail below).

When a change occurs to the model's data, the model (observable) notifies its views (observers) and the views then produce a new set of output values. In order for a view 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. The view uses that reference to get the data it needs to produce the new output.

Note that providing a view with a reference to the model has the undesirable side-effect that the view has access to the model's *mutators*, and hence could *modify* model data; see the discussion below on the Proxy design pattern.

Recall that there are two approaches which can be used to implement the Observer pattern: defining your own IObservable interface, or extending the build-in CN1 Observable class. You are required to use the latter approach (where your GameWorld class extends java.util.Observable). Note that you are also required to use the build-in CN1 Observer interface (which also resides in java.util package).

GUI Operations

Game class extends Form (as in A1) representing the top-level container of the GUI. The **Form** should be divided into three areas: one for commands, one for "points" information, and one for the "map" (which will be an empty **container** 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 as described in A1.

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. First, you will need on-screen buttons – one button for each of the input commands from A1 except for the 'p' and 'm' commands. The program should create the appropriately labeled buttons, add them to the panel, and add the panel as a component of the game. 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 (or "execute" method depending on your approach) in a command object which executes the corresponding code from A1.

The second input mechanism will use CN1 *Key Binding concepts* so that the *left arrow*, *right arrow, up arrow,* and *down arrow* keys invoke command objects corresponding to the code previously executed when the "l", "r", "i", and "d" keys (for changing the ship direction and speed) were entered, respectively. Note that this means that whenever an arrow key is pressed, the program will *immediately* invoke the corresponding action (no "Enter" key press is required). The program is also to use key bindings to bind the SPACE bar to the "fire missile" command and the "j" key to the "jump through hyperspace" command. If you want, you may also use key bindings to map any of the other command keys from A1, but only the ones listed above are required.

The third input mechanism will use a side menu (see class notes for tips on how to create a side menu using CN1 build-in **`Toolbar`** class). Your GUI should contain at least two menus: "*File*" containing at least *"New", "Save", "Undo", "Sound", "About"*, and *"Quit"* items; and *"Commands"* containing one item for each of the following user commands from A1: "a", "b", "s", "n", "k", "c", "x", "t" and "q" (give them appropriate names on the menu). Note that you do not need to create menu items for the KeyBinding commands listed earlier ("r", "l", "i", "d", "j", and "space"), or for the "p" and "m" commands.
On the "File" menu, only the *"Sound", "About"* and *"Quit"* items need to do anything for this assignment (although *all* menu items are required to provide confirmation that they were invoked). The Sound menu item should include a **`check box`** showing the current state of the "sound" attribute (in addition to the attribute's state being shown on the **`PointsView`** GUI panel as described above). 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 dialog box (use CN1 build-in Dialog class) giving your name, the course name, and any other information you want to display (for example, the version number of the program). "Quit" should prompt graphically for confirmation and then exit the program; you should also use a dialog box for this.

Selecting a Command menu item should invoke the corresponding command, just as if the button of the same name had been pushed. Recall that there is a requirement that commands be implemented using the *Command* design pattern. This means that there must be *only one* of each type of command object, which in turn means that the items on the Command menu must share their command objects with the corresponding control panel buttons. (We could *enforce* this rule using the *Singleton* design pattern, but that is not a requirement in A2; just don't create more than one of any given type of command object). Each of the commands is to perform exactly the same operations as they did in A1.

The **`PointsView`** class should display game state data for each of the points elements from A1 (total points, number of missiles, and elapsed time), plus one *new* attribute: *"Sound"*

with value either ON or OFF.[2] As described above, **`PointsView`** 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 **`PointsView`**, what **`update()`** does is update the contents of the labels displaying the game state (use Label method setText(String) to update the label). Note that these are exactly the same point values as were displayed previously (with the addition of the "Sound" attribute); the only difference now is that they are displayed *graphically* in **`PointsView`**.

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 CN1 documentation (i.e., CN1 JavaDocs and developer guide). It will become increasingly important that you familiarize yourself with and utilize these resources.

Command Design Pattern

The approach you must use for implementing command classes is to have each command extend the CN1 build-in Command class (which implements the ActionListener interface), as shown in the course notes. Code to perform the command operation then goes in the command's actionPerformed() method. Hence, actionPerformed() method of each command class that performs an operation invoked by a single-character command in A1, should call the appropriate method in the GameWorld that you have implemented in A1 when related single-character command is entered from the text field (e.g., accelerate command's actionPerformed() would call accelerate() method in GameWorld).

Hence, most command objects need to have the GameWorld as its target since they need to access the methods defined in this class. You could implement the specification of a command's target either by passing the target (reference to GameWorld) to the command constructor, or by including a "setTarget()" method in the command.

Button class is automatically able to be a "holder" for command objects; Button have a setCommand() method which allows inserting a command object into the button. Command automatically becomes a listener when added to a Button via setCommand() (you do not need to also call addActionListener()), and the specified Command is automatically invoked when the button is pressed, so if you use the CN1 facilities correctly then this particular observer/observable relationship is taken care of automatically.

The Game constructor should create a single instance of each command object (for example, a "Accelerate" command object, etc.), then insert the command objects into the command holders (buttons, side menu items, title bar area items) using methods like setCommand() (for buttons), addCommandToSideMenu() (for side menu items), and addCommandToRightBar() (for title bar area items). You should also bind these command objects to the keys using addKeyListener() method of Form. You must call super("command_name") in the constructors of command objects by providing appropriate command names. These command names will be used to override the labels of buttons and/or to provide the labels of side menu items or title bar area item(s).

[2] 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.

Note that commands can be invoked using multiple mechanisms (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 (e.g., through which mechanism) the command was invoked.

Iterator Design Pattern

The game object collection must be accessed through an appropriate implementation of the Iterator 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 should develop your own interfaces to implement this design pattern, instead of using the related build-in CN1 interfaces. The game object collection will implement an interface called **ICollection** which defines at least two methods: for adding an object to the collection (i.e., **add**()) and for obtaining an iterator over the collection (i.e., **getIterator**()). The iterator should exist as a private inner class inside game object collection class and should implement an interface called **IIterator** which defines at least two methods: for checking whether there are more elements to be processed in the collection (i.e., **hasNext**()) and returning the next element to be processed from the collection (i.e., **getNext**()).

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 CN1, you must implement an iterator yourself: The iterator should keep track of the current index and it cannot make use of the built-in iterator defined for the data structure (e.g., you cannot call **hasNext**() method of the built-in iterator defined for **Vector/ArrayList** from the **hasNext**() method of your iterator class). However, the iterator can use following build-in methods of the data structure: **size**() and **elementAt**()/**get**().

Proxy Design Pattern

To prevent views from being able to modify the **`GameWorld`** object received by their **`update()`** methods, the model (**`GameWorld`**) should pass a GameWorld *proxy* to the views; this proxy should allow each view to obtain all the required data from the model while prohibiting any attempt by the view to *change* the model. The simplest way to do this is to define an interface **`IGameWorld`** listing the methods provided by a **`GameWorld`**, and to provide a new class **`GameWorldProxy`** which implements this same interface. The **`GameWorld`** model then passes to each observer's **`update()`** method a **`GameWorldProxy`** object instead of the actual **`GameWorld`** object. See the attachment at the end for additional details.

Recall that there are two approaches which can be used to implement the Observer pattern: defining your own **IObservable** interface, or extending the build-in CN1 Observable class. You are required to use the latter approach (where your GameWorld class extends java.util.Observable). Note that you are also required to use the build-in CN1 Observer interface (which also resides in java.util package).

Additional Notes

- All menu items, buttons, and other GUI components must display a message on the console indicating they were selected. You may add additional GUI items if you like, but if you do, they too must at least indicate on the console that they were selected.
- You must use BorderLayout as the layout manager of your form and use FlowLayout or BoxLayout for the containers you have added to different areas of your form. These layout managers automatically place and size the containers/components that you have added to your form/containers.
- You can change the size of your buttons/labels using setPadding() method of Style. Each label in PointView should be divided into two parts, text and value, and padding should be added to the value part so that the labels look stable when value changes as discussed in the class notes. You can also use setPadding() on left and right control containers to start adding buttons at positions which are certain pixels below their upper borders.
- In A2, you must assign the size of your game world by querying the size of your MapView container (instead of assigning your width and height to 1024x768 as in A1, assign them to width and height of MapView) using getWidth() and getHeight() method of Component as discussed in the class notes.
- You must change the style of your buttons using methods like **`setBgTransparency()`**, **`setBgColor()`**, **`setFgColor()`** of **`Sytle`** class so that they look different from labels as discussed in the class notes. You can extend from the built-in **`Button`** class and do the styling in this new class. Then, while you construct your GUI, instead of creating instances of the build-in **`Button`** class, you can create objects out of this new user-defined button class.
- 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 **`PointsView`** observers – and hence generate the output formerly associated with the “m” and “d” 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.
- Note that since the ‘tick’ command causes moveable objects to move, every ‘tick’ will result in a new map view being output (because each tick changes the model). Note however that it is *not* the responsibility of the ‘tick’ command code to produce this output; it is a side effect of the observable/observer pattern. Note also that this is not the only command which causes generation of output as a side effect. You should verify that your program correctly produces updated views automatically whenever it should.
- 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");
```

- You may not use a "GUI Builder" tool for this assignment. (If you don't know what a GUI Builder is, don't worry about it.)
- All functionality from the previous assignment must be retained unless it is explicitly changed or deleted by the requirements for this assignment.
- As before, you should develop a *UML diagram* showing the relationships between your classes, including not only the major fields and methods required but also the interfaces and the relationships between classes using those interfaces (for example, Observer/Observable). This will be particularly useful in helping you understand what modifications you need to make to your code from A1. Feel free to see me if you are not sure your design is correct. Note that if you come to see me for help with the program – which is encouraged – the first thing I am likely to ask is to see is your UML diagram.
- Although the **MapView** Container 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. See the sample picture below.

Deliverables

Submitting your program requires the same three steps as for A1:

1. Be sure to verify that your program works from the command prompt as explained above.

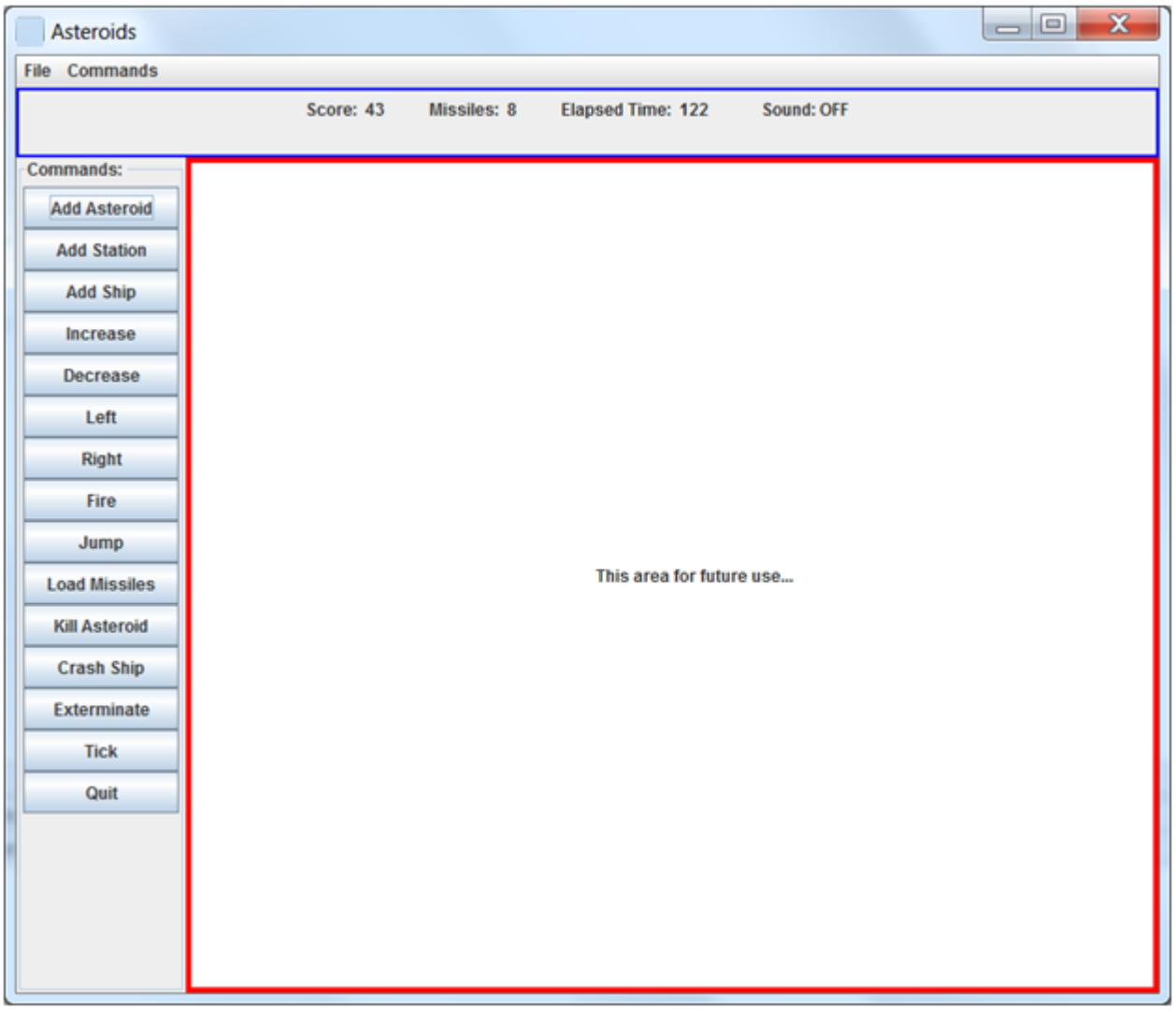
2. Create a *single* file in "ZIP" format containing (1) your UML diagram in PDF format, (2) the entire "src" directory under your CN1 project directory (called A2Prj) which includes source code (".java") for all the classes in your program, and (3) the A2Prj.jar located under the "A2Prj/dist" directory which includes the compiled (".class") files for your program in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a2.zip. Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications (you do not need to include the readme file, if you have not made any changes or additions).

3. Login to ***SacCT***, select “Assignment 2”, and upload your *verified* ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a *backup copy* of all submitted work (save a copy of your ZIP file).

All submitted work must be strictly your own!

Sample GUI

The following shows an example of what your game GUI MIGHT look like. Notice that it has a control panel on the left containing all the required buttons, a menu bar containing the required menus, a PointsView panel near the top showing the current game state information, and an (empty) MapView panel in the middle for future use. The title “Asteroids” displays at the top.



Java Notes

Below is the organization for your MVC code for A2. Note that in this organization, we are using the CN1 “Observable” class and CN1 “Observer” interface to derive your GameWorld and views. Doing it this way has the benefit of CN1 handling the “list of observers” for you. Note also that this pseudo-code 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 Form {

     private GameWorld gw;
     private MapView mv;              // new in A2
     private PointsView pv;           // new in A2

     public Game() {
          gw = new GameWorld();       // create “Observable”
          mv = new MapView();         // create an “Observer” for the map
          pv = new PointsView(gw);    // create an “Observer” for the points
          gw.addObserver(mv);         // register the map Observer
          gw.addObserver(pv);         // register the points 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 PointsView panel to the form

          this.show();
     }
}

public interface IGameWorld {
     //specifications here for all GameWorld methods
}

public class GameWorld extends Observable implements IGameWorld {
     // code here to hold and manipulate world objects, handle observer registration,
     // invoke observer callbacks by passing a GameWorld proxy, etc.
}

public class GameWorldProxy extends Observable implements 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 Container implements Observer {
     public void update (Observable o, Object arg) {
          // code here to output current map information (based on the data in the Observable)
          // to the console.  Note that the received “Observable” is a GameWorld PROXY and can
          // be cast to type IGameWorld in order to access the GameWorld methods in it.
     }
}

public class PointsView extends Container implements Observer {
     public void update (Observable o, Object arg) {
          // code here to update labels from data in the Observable (a GameWorldPROXY)
     }
}
```