CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2012
John Clevenger

# Assignment #2: Design Patterns and GUIs

**Due Date: Thursday March 29th [2 weeks + Spring Break]**

## 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 *score* 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 component 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 in the game world 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 balls left, 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 "d" key for outputting the current 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 ***ScoreView*** class containing code to output the points and other state information.

**GameWorld** should therefore be defined as an *observable*, with two *observers* – **MapView** and **ScoreView.** 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 brick being removed) 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 the current score and related values in the case of **ScoreView)**. The **MapView** output for this assignment is unchanged from A1: text output on the console. However, the **ScoreView** is to present a *graphical* display of the game state values (this is 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.

## 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 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 as before.

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 a class that extends **JPanel** and contains buttons – one button for each of the input commands from A1 except for the 'd' 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 (**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 (or "execute" method depending on your approach) in a command object which executes the corresponding code from A1.

The second input mechanism will use Java *KeyBindings* so that the *left arrow*, *right arrow,* and *up arrow* keys invoke command objects corresponding to the code previously executed when the "l", "r", and "i" keys (for moving the paddle left/right and for increasing the ball 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 "next ball" command and the "c" key to the "change paddle mode" 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 menus. 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", "s", "u", "b", "p", "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", "c", and "n"), or for the "d" 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 **`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 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 **`JOptionPane`** dialog box (see **`JOptionPane.showMessageDialog()`**) 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; **`JOptionPane.showConfirmDialog()`** can be used 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 **`ScoreView`** class should extend **`JPanel`** and contain **`JLabel`** components for each of the score elements from A1 (total points, number of balls, and elapsed time), plus one *new* attribute: *"Sound"* with value either ON or OFF.[2] 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 text of the **`JLabels`** displaying the score values in the **`JPanel`** (use **`JLabel`** method **`setText(String)`** to update the label). Note that these are exactly the

[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.

same score values as were displayed previously in A1 (with the addition of the "Sound" attribute); the only difference now is that they are displayed *graphically* in a **`JLabel`**.

Although most of the GUI building details are covered 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 Design Pattern

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 operation then goes in the command's **`actionPerformed()`** method. Use of the Java **`AbstractAction`** class has the added benefit of automatically providing a built-in "command holder" structure: Java **`AbstractButton`** subclasses (for example, **`JButton`** and **`JMenuItem`**), are automatically able to be "holders" for **`Action`** objects (that is because the **`AbstractAction`** class implements the **`Action`** interface). Nothing in any command object may contain knowledge of *how* the command was invoked.

The game initialization code should create a *single* instance of each command object (for example, a "next ball" command object, a "move paddle left" command object, etc.), then insert the command objects into the command holders (control panel buttons and menu items) using **`setAction()`**. Java **`AbstractActions`** automatically become listeners when added to an **`AbstractButton`** via **`setAction()`**, so if you use the Java facilities correctly then this particular observer/observable relationship is taken care of automatically.

Some command objects may need to be created 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.

## 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 may use the interfaces discussed in class for this, or the more complex Java version, 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*. For this assignment, your iterator will need to at least include **`hasNext()`**, **`getNext()`**, and **`remove()`**.

## 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 course notes and the attachment at the end for additional details.

Recall that there are two approaches which can be used to implement the **`Observable`** pattern: defining your own **`Observable`** interface, or extending the Java **`Observable`** class. If you choose to use Java's "**`Observable`**" class (e.g. declaring that your **`GameWorld`** class extends **`java.util.Observable`** instead of implementing your own **`Observable`** interface as discussed in class), you will find it is more complicated to deal with the proxy requirement: you will have to override not only the Java **`notifyObservers()`** method so that you can pass a proxy, but also the **`addObserver()`** method so that you can keep track of what observers to call back with a proxy. This means it's almost more work to extend the Java **`Observable`** class than to implement it via an interface yourself.

## 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.
- Make your initial GUI frame reasonably large, but small enough to work on most screens (a **`setSize()`** of something like 800x600 is 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 "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 of interest and to 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 leftArrowKey = KeyStroke.getKeyStroke("LEFT");
```

- **`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 instead of 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 not use a "GUI Builder" tool for this assignment. (If you don't know what a GUI Builder is, don't worry about it.)
- As before, your program for this assignment must be contained in a Java *package*. The name of the package must be exactly "**`a2`**". In other words, every class in your program must have the statement "**`package a2;`**" as its first statement, and it must be possible to execute the program from a command prompt by changing to the directory containing the "**`a2`**" package (subdirectory) and typing the single command "**`java a2.Starter`**".
- 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.
- 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 also have to implement the "Command Holder" facility yourself as well. You would also need to take care of setting up the listener and callback mechanisms yourself. 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. See the sample picture below.

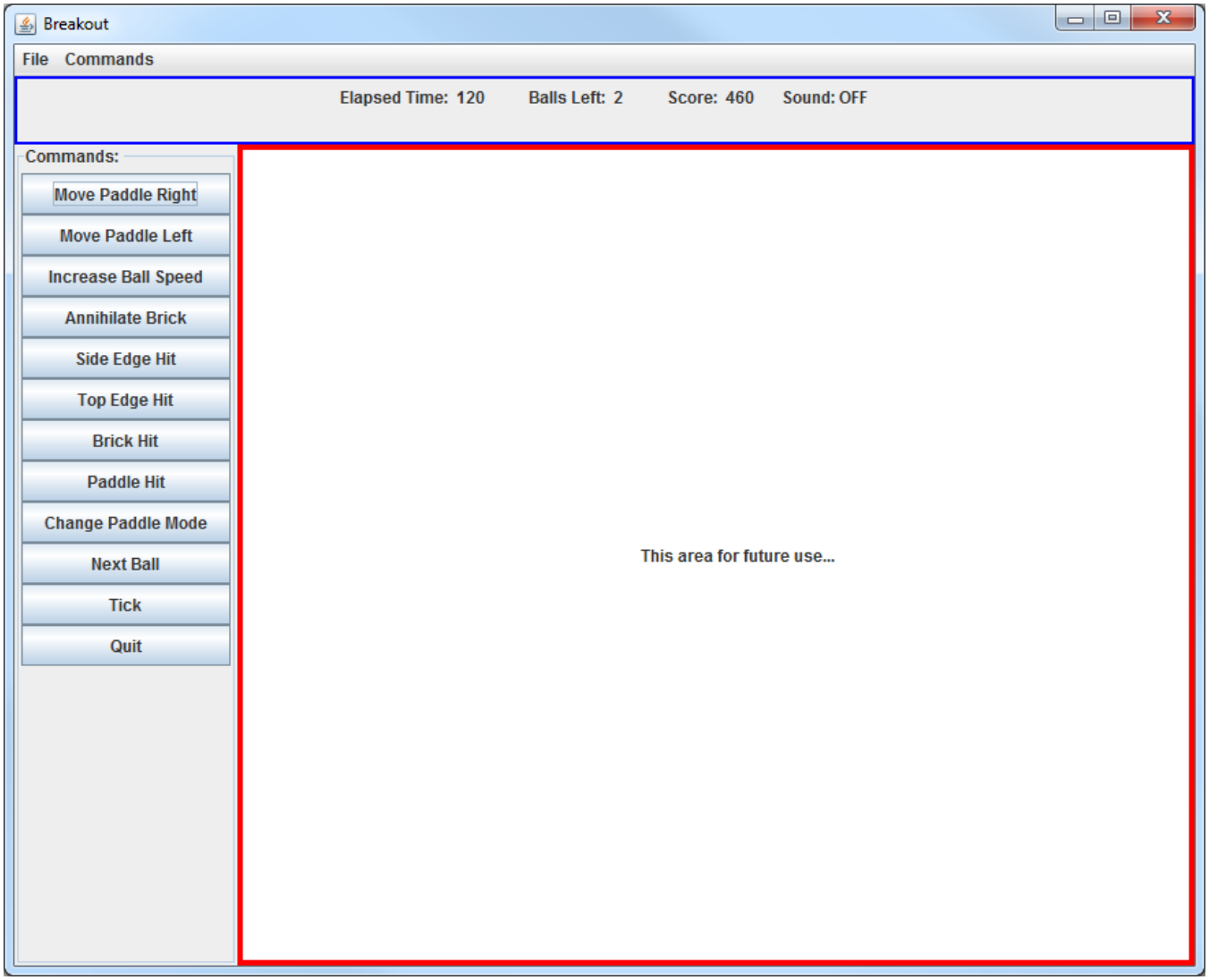
## Deliverables

Submission is done using ***SacCT***. Submit a single "ZIP" file containing both the *Java source code* for all the classes in your program and also *compiled (".class")* files. Be sure your ZIP file contains the proper subpackage hierarchy. Also, be sure to keep a *backup copy* of all submitted work. All submitted work must be *strictly* your own.

Include a 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.

## 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 ScoreView panel near the top showing the current game state information, and an (empty) bordered MapView panel in the middle for future use. The title "Breakout" displays at the top.




## Java Notes

Below is one possible organization for your code for A2. Note that this organization is based on the assumption of using INTERFACES for both Observer and Observable. If you choose instead to use Java's Observable class, you'll have to modify the code to account for that. 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 JFrame {

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

     public Game() {
          gw = new GameWorld();      // create "Observable"
          mv = new MapView();        // create an "Observer" for the map
          sv = new ScoreView();      // create an "Observer" for the score 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 by passing a GameWorld proxy, 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 (IObservable 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 ScoreView extends JPanel implements IObserver {
     public void update (IObservable o, Object arg) {
          // code here to update JLabels from data in the Observable (a GameWorldPROXY)
     }
}
```

JC:jc
3/9/2012