CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCEScoreView
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2015 / Dr. Muyan

Assignment #2: Design Patterns and GUIs

Due Date: Tuesday, Oct 20th

Introduction

In this assignment you will 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`**). 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.). 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.

The program must use appropriate interfaces for organizing the required design patterns. In all, the following design patterns will 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.

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 (e.g. have a class named **`GameObjectCollection`**). Any routine that needs to process the objects in the collection must access the collection via an iterator (described below). All game objects are to be contained in this *single* collection data structure[1].

The model also contains the game state data as A1 (the total score, number of cats/dogs captured, number of cats/dogs remaining), plus a new state value: a flag indicating whether Sound is ON or OFF (described below).

[1] If you did not implement your game object collection this way in A1 you must change it for this assignment.

Views

A1 contained two functions to output game data: the "m" key for outputing 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 have 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.

GameWorld should therefore 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 kitten being born) 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 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. Note this has the undesirable side-effect that *the view has access to the model's mutators*, and hence could *modify* model data (later in the lectures, we will discuss how to address this issue with the Proxy pattern).

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. You may use either approach. The latter is described in the course notes, and an example of the former (where your **GameWorld** class extends **java.util.Observable**) is shown at the end of the handout. Note that you will also need to have the **Observer** interface (you can use **java.util.Observer**).

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.

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 *command objects* (see below) for each of the commands from A1 (except "p" and "m" which are implemented as views in A2) 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):

Command	KeyBinding	MenuItem	Button
expand the net		x	x
contract the net		x	x
scoop animals	x		x
move the net **r**ight	x		x
move the net **l**eft	x		x
move the net **u**p	x		x
move the net **d**own	x		x
produce a **k**itten		x	x
a cat-dog **f**ight		x	x
clock **t**icks			x
quit game	x	x	

For the "on-screen buttons" input mechanism, you will need to have a **`JPanel`** (a control panel) that contains buttons for all commands except the "quit" command listed in the above table. You will create the 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.

The "key bindings" input mechanism will use the Java *KeyBinding* concept so that the *right arrow*, *left arrow, up arrow, down arrow, "s", and "q"* keys invoke command objects corresponding to the code previously executed when the "r", "l", "u", "d", "s", and "q" keys (for moving and scooping with the net and quitting the game) were entered, respectively. 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). If you want, you may also use key bindings to map any of the other command keys, but only the ones mentioned above are required.

The "menu items" input mechanism will use menus. Your GUI should contain at least two menus: "*File*" containing at least *"New", "Save", "Undo", "Sound", and "About"* items; and *"Commands"* containing one item for each of the following user commands from A1: "e", "c", "k", "f", and "q". Also note that the "quit command" object (which is invoked via "Quit" menu item and "q" key) should prompt graphically for confirmation and then exit the program; **`JOptionPane.showConfirmDialog()`** can be used for this.

On the "File" menu, only the *"Sound",* and *"About"* 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.

Selecting a Command menu item (like hitting a mapped key) 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 and key bindings must share their command objects with the corresponding buttons. (We could *enforce* the 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 are 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 points elements from A1 (total points, cats/dogs captured, cats/dogs remaining), 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 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 points values in the **`JPanel`** (use **`JLabel`** 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 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 Design Pattern

The recommended approach for implementing command classes is to have each Command extend the Java **`AbstractAction`** class (which implements the **`Action`** interface), 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`** constructor should create a *single* instance of each command object (for example, a "Scoop" command object, a "Fight" command object, etc.), then insert the command objects into the command holders (buttons, menu items, and/or Key Binding **`ActionMaps`**) using methods like **`setAction()`** (for buttons and menu items) and **`put()`** (for **`ActionMaps`**). Make sure that you call **`super("command_name")`** in the constructors of

[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.

command objects by providing appropriate names since these names will be used to override the labels of command holders of these command objects.

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.

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.

Iterators

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. The game object collection will implement an interface which defines at least two methods: for adding an object to the collection (e.g. **`add()`**) and for obtaining an iterator over the collection (e.g. **`getIterator()`**). The iterator will implement an interface which defines at least three methods: for checking whether there are more elements to be processed in the collection (e.g. **`hasNext())`**, returning the next element to be processed from the collection (e.g. **`getNext())`**, and removing from the collection the last element returned by the iterator (e.g. **`remove())`**.

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.*

Additional Notes

- 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 "m" and "p" commands. This means you do not need the "p" or "m" commands; the output formerly produced by those commands is now generated by the observer/observable process.
- Since the 'tick' command causes animals 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.

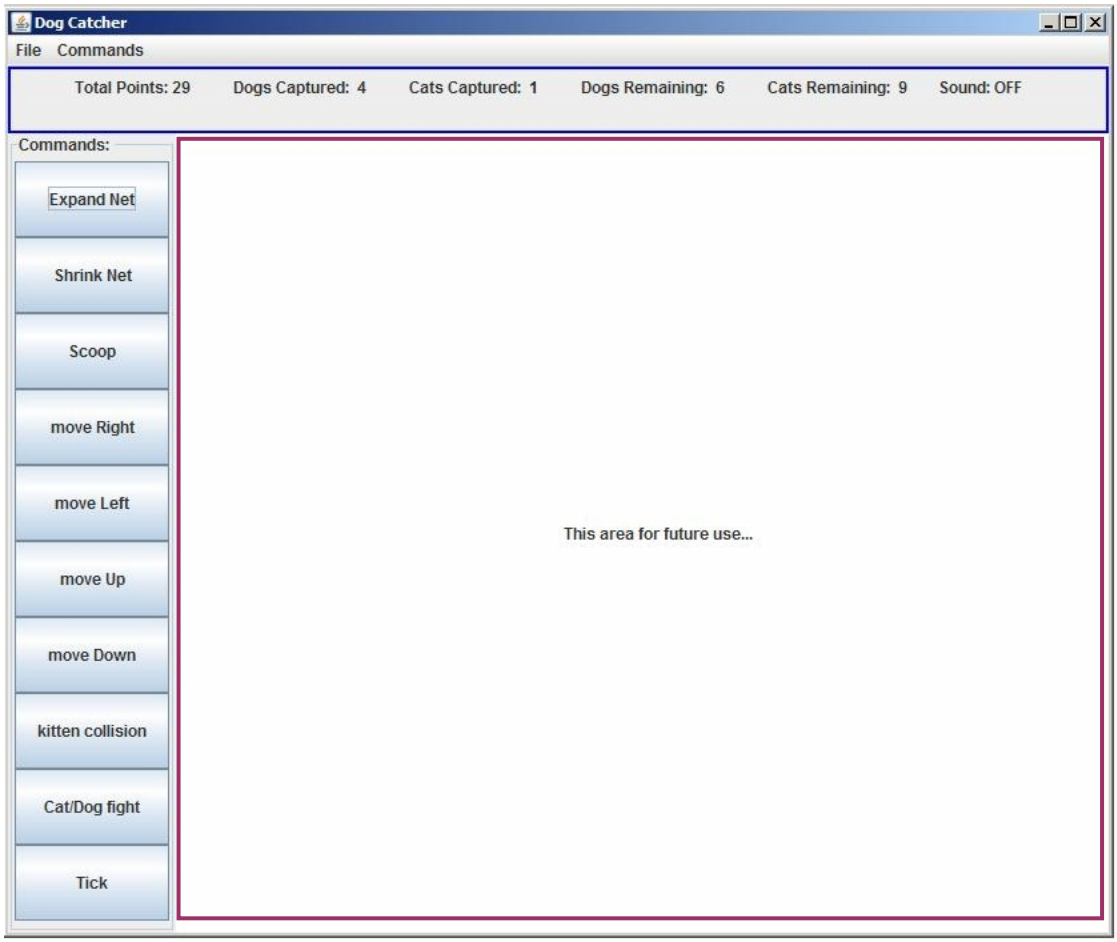
- Programs must contain appropriate documentation as described in A1.
- The mechanism for using Java "Key Bindings" is to define a **`KeyStroke`** object for each key and insert that object into the **`WHEN_IN_FOCUSED_WINDOW`** input map of **`MapView`** panel while also inserting the corresponding command object into this 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 <u>not</u> use a "GUI Builder" tool for this assignment.
- 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.
- 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.
- Although the **`MapView`** panel 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.

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



Deliverables

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

1. Create a *single* file in "ZIP" format containing (1) your UML diagram in .PDF format, (2) the *Java source code (".java")* for all the classes in your program, and (3) the compiled (".class") files for your program. Be sure your ZIP file contains the proper subpackage hierarchy. Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications.

2. *Verify your submission file.* To do this, copy your ZIP file to an empty folder on your machine, unzip it, open a command prompt window, change to the folder where you unzipped the file, and type the command "**java a2.Starter"**. If this does not properly execute your program, your ZIP file is not properly structured; go back to step (1) and fix it. Substantial penalties will be applied to submissions which have not been properly verified according to the above steps.

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!

Java Notes

Below is one possible organization for your MVC code for A2. Note that this organization is based on the assumption of using the Java "Observable" class and Java "Observer" interface to derive your GameWorld and views. Doing it this way has the benefit of Java 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 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);

		// some Swing methods will only function when the frame is visible!
	}
}

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

public class MapView extends JPanel 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
	}
}

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