CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2020 – Dr. Muyan

Assignment #2: Design Patterns and GUIs

Due Date: Sunday, March 15th 11:59 PM

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 **`play()`** method. 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 *score* view which will be graphical, and a *map* 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).

Single-character commands entered via a text field in A1 will be replaced by GUI components (side menu items, buttons, key bindings, etc.). Each such component will have an associated "command" object, and the command objects will perform the same operations as previously performed by the single-character commands.

The program must use appropriate interfaces and built-in classes 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,
- *Strategy* – to control movement for additional (non-player) cyborgs.
- *Singleton* – to ensure that only a single instance of player cyborg can exist.

Model Organization

`GameWorld` is to be reorganized so that it has a **`GameObjectCollection`** which contains a collection of *game objects*. All game objects are to be contained in this *single* collection. Any routine that needs to process the objects in the collection must access the collection via an iterator (described below).

The model is also to contain the same game state data as A1 (current clock time and lives remaining), 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/player-cyborg state data* (i.e., the number of lives left, the current clock value, last base number the player's cyborg has reached sequentially so far, the player cyborg's current energy level, and player cyborg's current damage level). 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 game/player-cyborg 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 energy station 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/player-cyborg 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/player-cyborg state values (see below).

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

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 (which is done automatically by the **notifyObservers()** built-in method when built-in **Observable** class is used). 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).

Non-Player Cyborgs (NPCs)

The "game object" hierarchy will be the same as in A1 except that you will add a new concrete kind of movable object called **NonPlayerCyborg,** which extends from **Cyborg.** But **Cyborg** will now be an abstract type and we will add another concrete type called **PlayerCyborg** which also extends from **Cyborg**. The game initialization code should create and add to the game world three instances of **NonPlayerCyborg** (so that there are now four cyborgs – the player's cyborg which is an instance of **PlayerCyborg** plus three "Non-Player Cyborgs (NPCs)". Each NPC is to have an initial location which is "near" the first base, but not exactly *at* the first base; instead, each NPC should be at least several cyborg lengths away from the first base (this is to make sure that the NPCs are not colliding with the player's cyborg to start with).

When an NPC collides with the player's cyborg, the player's cyborg sustains *damage* just as described in A1 – including that if the player's cyborg 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.

Having NPCs in this version of the game introduces an additional case for ending the game such that if a NPC reaches the last base before the player does, the game will end with the following message displayed on the console: "Game over, a non-player cyborg wins!".

GUI Operations

`Game` class extends **`Form`** (as in A1) representing the top-level container of the GUI. The form should be divided into five areas: one for "score" information, three for commands, 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 the **`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.

Hence, in this assignment, the **`play()`** method used in A1 which prompts for single-character commands and reads them from a text field on the form is to be discarded. In its place, commands will be input through three different mechanisms: on-screen buttons, key bindings, and side menu items (we will also utilize a tile bar area item to provide help as explained below). Each command will be invokable via a combination of these mechanisms. You are to create *command objects* (see below) for each of the commands from A1 (except "d" and "m" which are implemented as views in A2, and "y" and "n" which are replaced by an exit dialog box – see below) and for new commands introduced in A2 (i.e., change strategies, sound, about, and help). You are to attach the objects as commands to various combinations of invokers as shown in the following table (a '+' 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	Side Menu (or Title Bar Area) Item	Button
accelerate	+	+	+
brake	+		+
left turn	+		+
right turn	+		+
collide with NPC			+
collide with base			+
collide with **e**nergy station	+		+
collide with drone (**g**)	+		+
tick	+		+
exit		+	
change strategies			+
turn the sound on or off		+	
give about information		+	
give help information		+	

For west, east, and south regions on the form (**Game**), you should have a different **Container** (another built-in CN1 class) object created. For north and center regions, you should have a **ScoreView** and a **MapView** object created, respectively. **ScoreView** and **MapView** classes should extend from **Container**. **MapView** will be an empty container whereas you should add appropriate labels (use CN1 built-in **Label** class) to **ScoreView** (see below for details). Adding the labels to the **ScoreView** should be done in the constructor of the **ScoreView**. For regions with buttons (i.e., west, east, and south), after you create their **Container** objects (which we will call as "control containers"), you should add related buttons to these containers.

Each button is to have an appropriate command object attached to it, so that when a button gets pushed it invokes the **actionPerformed()** method in the corresponding command object (the **execute()** method in terms of the Command pattern), which in turn calls appropriate command method in **GameWorld**.

Most commands execute exactly the same code as was implemented in A1. One exception is the "collide with base" command. Previously this command consisted of a number (between 1-9) entered from the text field located on the form which we eliminate in A2. Now, this command is to use the static method **Dialog.show()** to display a dialog box that allows the user to enter the number on a text field located on the dialog box (please see the "Titled Form in CN1" slide of "GUI Basics" chapter in lecture notes for more information). Note that if the user inputs an invalid value; your program should handle this gracefully.

The other command which must operate slightly different in this assignment is '**c**' (player's cyborg **c**ollided with NPC). In A1, this command just increased the damage level of the player's cyborg. Now, the command is to also add damage to one of the NPCs: 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.

The "key binding" input mechanism will use the CN1 *key binding* concept so that the *"a", "b", "l", "r", "e", "g",* and *"t"* keys invoke command objects corresponding to the code previously executed when the *"a", "b", "l", "r", "e", "g",* and *"t"* single-character commands 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 from A1, but only the ones listed here are required.

The "side menu item" input mechanism will use a side menu (see class notes for tips on how to create a side menu using CN1 built-in **Toolbar** class). Your GUI will contain a side menu that contains the following items: "Accelerate", "Sound", "About", and "Exit". "Accelerate" menu item should invoke "a" command. "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 **ScoreView** container as described below). Selecting the Sound menu item check box (use CN1 built-in **CheckBox** class) should set a boolean "sound" attribute to "ON" or "OFF", accordingly. The "About" menu item is to display a dialog box (use CN1 built-in **Dialog** class) giving your name, the course name, and any other information (for example, the version number of the program) you want to display. "Exit" side menu item should invoke "x"

command. Note that in A2 the "x" command should prompt graphically for confirmation and then exit the program; you should also use a dialog box for this.

Side menu will be placed on the left side of the title bar area. You are also required to add a "Help" item to the right side of the title bar area. When "Help" is invoked, you are required to display a dialog box listing the keys user can use while playing the game.

Selecting a side menu item that performs a game command (e.g., "Accelerate" menu item) should invoke the same code, when the button of the same name had been pushed (e.g., "Accelerate" button) and/or its related key has been hit (e.g., "a" key). 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 menu items, key bindings, and buttons must share their command objects. (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).

The **`ScoreView`** class should display *game/player-cyborg state data* (i.e., clock value, lives left, the player cyborg's last base reached, the player cyborg's energy level, and the player cyborg's damage level), plus one *new* attribute: *"Sound"* with value either ON or OFF.[1] You should add several labels to the **`ScoreView`** to create this display. As described above, **`ScoreView`** must be registered as an observer of **`GameWorld`**. Whenever any change occurs in the world, **`GameWorld`** calls **`notifyObsersevers()`** (do not forget to call **`setChanged()`** first) which in turn calls the **`update()`** method in its observers (you do not need to call **`update()`** yourself, it is automatically called). In the case of the **`ScoreView`**, what **`update()`** does is updating the contents of the labels displaying the game/player-cyborg state (use **`Label`** method **`setText()`** to update the label; if you cannot see a change on labels when you call **`setText()`**, try calling **`repaint()`** on the **`ScoreView`**). Note that these are exactly the same game/player-cyborg state values as were displayed previously (with the addition of the "Sound" attribute); the only difference now is that they are displayed *graphically* in **`ScoreView`**.

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 built-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

[1] 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'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, a "Brake" 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).

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 built-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 built-in methods of the data structure: **size()** and **elementAt()/get()**.

Strategy Design Pattern

NPCs move around the world according to their current *strategy*. That is, the NPC uses the current strategy of that cyborg to determine the steering direction and speed, 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 base (that is, the base 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 cyborg (in other words, its strategy is to play "Attack" by trying to collide into the player's cyborg). 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 bases).

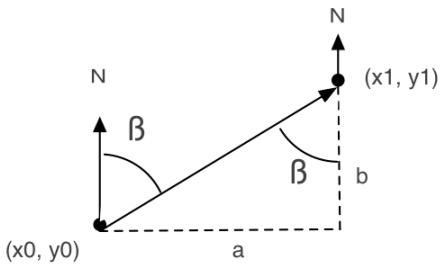
The program must use the Strategy design pattern to define the strategy for each NPC. You must use **`IStrategy`** interface instead of utilizing an abstract strategy super class. You should add two methods called **`setStrategy()`** and **`invokeStrategy()`** and a field for saving the current strategy to the NPC class. 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 way you choose. Like all game objects, NPCs should include a **`toString()`** method; the NPC **`toString()`** should return a string which includes the NPC's information provided for a player cyborg plus an identification of that NPC's current strategy.

When the human player invokes the "*switch strategy*" command (by pressing the appropriate GUI button), the game should cause all NPCs to *switch to a different movement strategy*. Switching strategies is to be done by invoking the NPC'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 the next "last base reached" value (otherwise, NPCs will never move beyond Base #2 since we have no command analogous to "collide with base" which applies to NPCs). Also, you should arrange so that NPCs never run out of energy (e.g., NPCs do not need to collide with energy, if the NPC's energy level is getting low, just set it to a reasonable value).

Additional Notes

- It is a requirement that the program contain only a *single* instance of player cyborg. This requirement must be enforced via the *Singleton* design pattern.
- 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`** class. Each label in **`ScoreView`** can be divided into two parts, text and value, and padding can 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 1000x1000 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 they look different from labels as shown 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 built-in **`Button`** class, you can create objects out of this new user-defined button class.
- Since almost all commands change the **`GameWorld`** (i.e., all command but exit, about, and help), they produce updated views as side effects. For instance, since the 'tick' command causes opponents 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 (i.e., **`tick()`** should not call **`update()`** on the observers); it is a side effect of the observable/observer pattern. You should verify that your program correctly produces updated views automatically whenever it should.
- Note that all game data manipulation by a command is accomplished by the command object invoking appropriate methods in the **`GameWorld`** (i.e., **`actionPerformed()`** of each command class would call the appropriate method in **`GameWorld`**). 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 as explained above. Please note that even though some changes to the **`GameWorld`** may not relate to the data displayed in all observers (e.g., changes introduced by the 'accelerate' command only relate to the data displayed in **`MapView`** not in **`ScoreView`**), every time a change happens, it is okay to invoke all observers.
- You may found the following formula useful while implementing the NPC strategies:



ß = arc tan (a , b)

(image from a tutorial at http://www.invasivecode.com/)

Here, (x0,y0) is the current location of the NPC, (x1,y1) is where it wants to go (target), then B is the ideal compass angle (90 – ideal_heading) for the NPC so it can go towards its target. Based on this information NPC should change its steering direction accordingly in order to approach this ideal heading. To calculate *arc tan* you can use **`atan()`** method of CN1 **`MathUtil`** class which generates angle in "radians". If you need, you can use **`MathUtil.toDegrees()`** to convert radians to degrees.

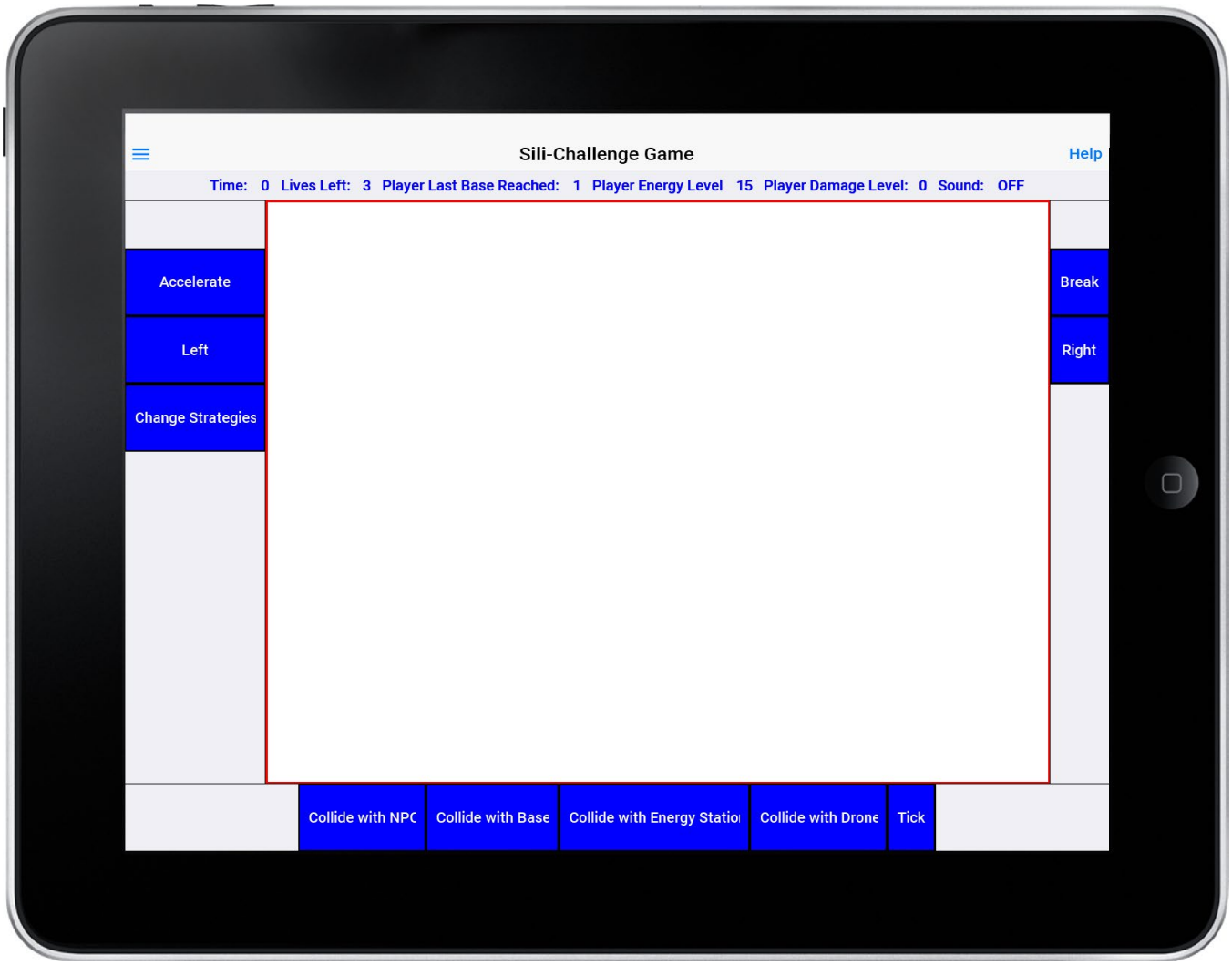
- You may not use a CN1's "GUI Builder" tool for this assignment.
- You must use IPad III OS7 skin in landscape orientation to develop and test your application.
- Programs must contain appropriate documentation as described in A1 and in class.
- All functionality from the previous assignment must be 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/interfaces (and built-in classes/interfaces that your entities extend/implement). This will be particularly useful in helping you understand what modifications you need to make to your code from A1.
- You must use a tool to draw your UML (e.g., Violet or any other UML drawing tool) and output your UML as a single pdf file (e.g., print your UML to a single pdf file). You are **not** allowed to use tools that automatically generate a UML from your code. For your entities you must use three-box notation (for built-in entities use single-box notation - you are only required to include built-in entities that you extend/implement) and show all the important fields and methods. You must indicate the visibility modifiers of your fields and methods, but you are not required to show parameters in methods and return types of the methods.
- Although the **`MapView`** container is empty for this assignment, you should put a red border around it and install it in the form, to at least make sure that it is there (use **`setBorder()`** method of **`Style`** class as described in the class notes).
- Your program must be contained in a CN1 project called A2Prj. You must create your project following the instructions given at "2 – Introduction to Mobile App Development and CN1" lecture notes posted at Canvas (Steps for Eclipse: 1) File → New → Project → Codename One Project. 2) Give a project name "A2Prj" and uncheck "Java 8 project" 3) Hit "Next". 4) Give a main class name "Starter", package name "com.mycompany.a2", and select a "native" theme, and "Hello World(Bare Bones)" template (for manual GUI building). 5) Hit "Finish".). Further, ***you must verify that your program works properly from the command prompt*** before submitting it to Canvas (Get into the A2Prj directory and type:

  java -cp dist\A2Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a2.Starter

  for Windows machines. For the command line arguments of Mac OS/Linux machines please refer to the class notes.). Substantial penalties will be applied to submissions which do not work properly from the command prompt.

Sample GUI

The following shows how your game GUI should look like. Notice that it has three control containers containing all the required buttons (on the left, right, and bottom), a side menu and "Help" item on title bar area, a **`ScoreView`** container near the top showing the current game/player-cyborg state information, and an (empty) bordered **`MapView`** container in the middle for future use (notice that there is a red border around the **`MapView`**). The title "Sili-Challenge Game" also displays on title bar area.



Deliverables

Submitting your program requires the similar steps as for A1:

1. Create a "TEXT" (i.e., not a pdf, doc etc.) file called "readme-a2.txt" that includes the lab number and the name of the specific machine you have used in that lab to build/test your program (or if you have used Hyrda Windows Terminal Server, just say it was tested on Hydra). Be sure to **verify that your program works from the command prompt on the lab machine** as explained above. For hints on how to connect to Hydra and how to build/test your assignment on a lab machines, please see the syllabus. In addition, **be sure that you include the *src* folder and *jar* file generated/tested on the lab machine in the below-mentioned zip file**. You may also include additional information you want to share with the grader in this text file. You will receive the grader comments on your text file as annotations when grades are posted (click on "View Feedback" link next to readme-a2.txt on Canvas to see the comments).

2. Create a "ZIP" file 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 is automatically generated by CN1 and includes the compiled (".class") files

for your program in a zipped format. Do **NOT** include other directories/files under the CN1 project directory. Be sure to name your ZIP file as YourLastName-YourFirstName-a2.zip.

3. Login to ***Canvas***, select "Assignment#2", and upload your ZIP file and TEXT file separately (**do NOT place this TEXT file inside the 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 and TEXT files).

All submitted work must be strictly your own!

CN1 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 create 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 ScoreView sv;          // new in A2

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

            // code here to create Command objects for each command,
            // add commands to side menu and title bar area, bind commands to keys, create
            // control containers for the buttons, add buttons to the control containers,
            // add commands to the buttons, and add control containers, MapView, and
            // ScoreView to the form

            this.show();
            ... // code here to query MapView's width and height and set them as world's
                // width and height
            gw.init();                // initialize world
      }
}
public class GameWorld extends Observable {
   public void init(){
      //code here to create the initial game objects
   }
// additional methods here to manipulate game objects and related game state data
}


public class MapView extends Container implements Observer {
      public void update (Observable o, Object arg) {
            // code here to call the method in GameWorld (Observable) that output the
            // game object information to the console

      }
}
public class ScoreView extends Container implements Observer {
      public void update (Observable o, Object arg) {
            // code here to update labels from the game/player-cyborg state data
      }
}
```