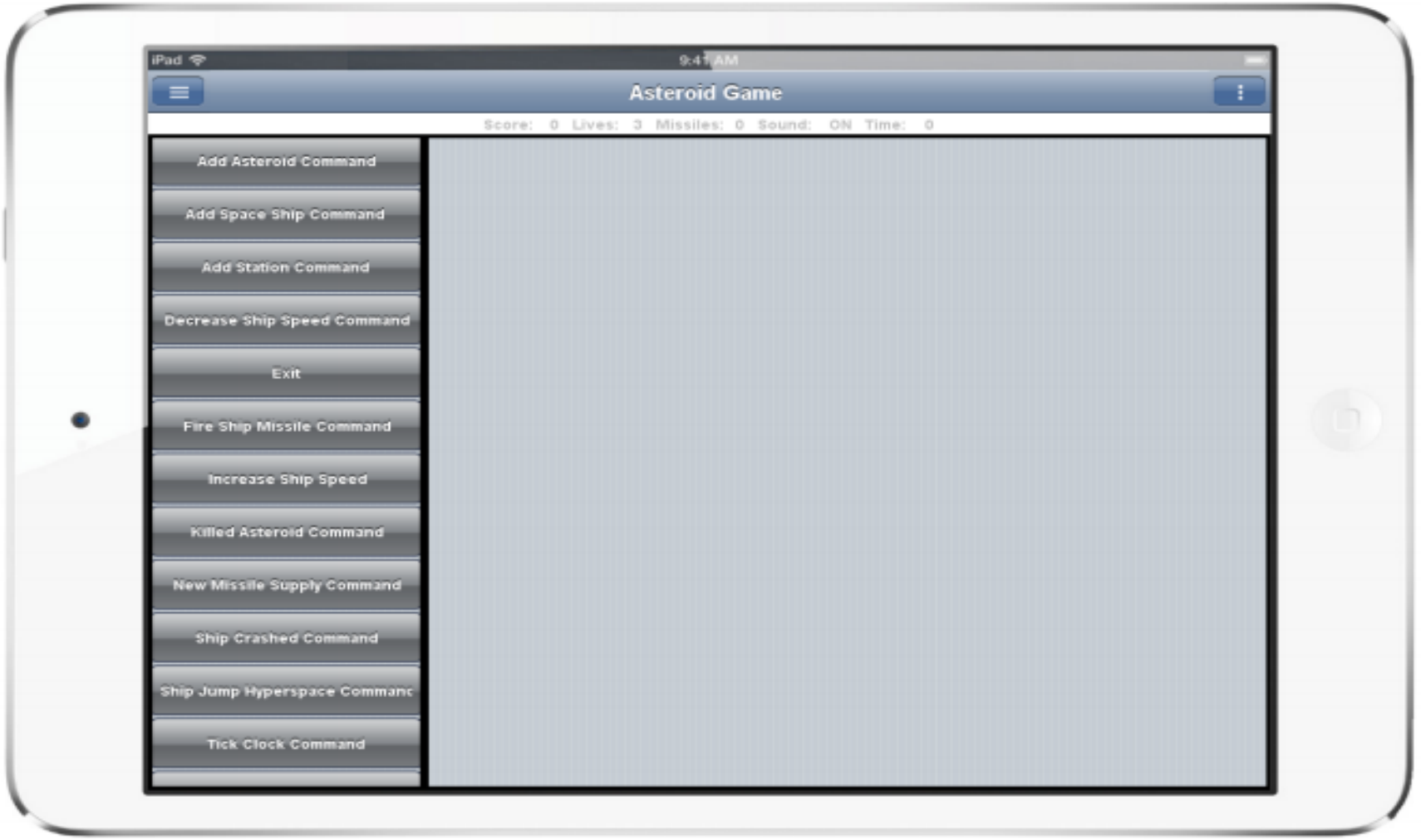
Assignment 2 Discussion

Assignment 2 GUI



(Note: This is a mockup screen only. Not all the commands are displayed here)

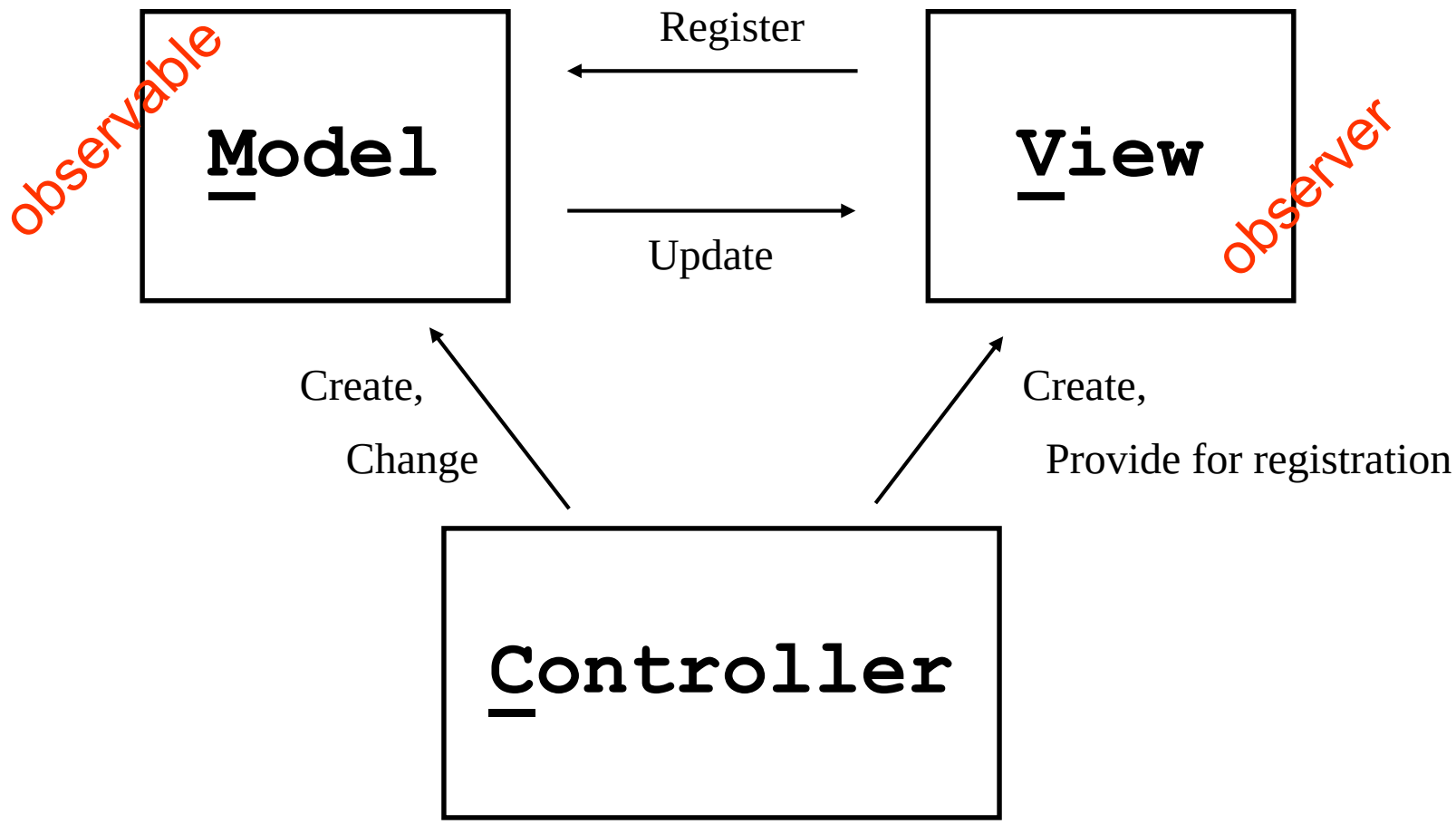
Note: Please see ContainerDemo in Canvas (SampleUIProgramSpring2018) for practice.

Assignment 2 – Design Patterns

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.

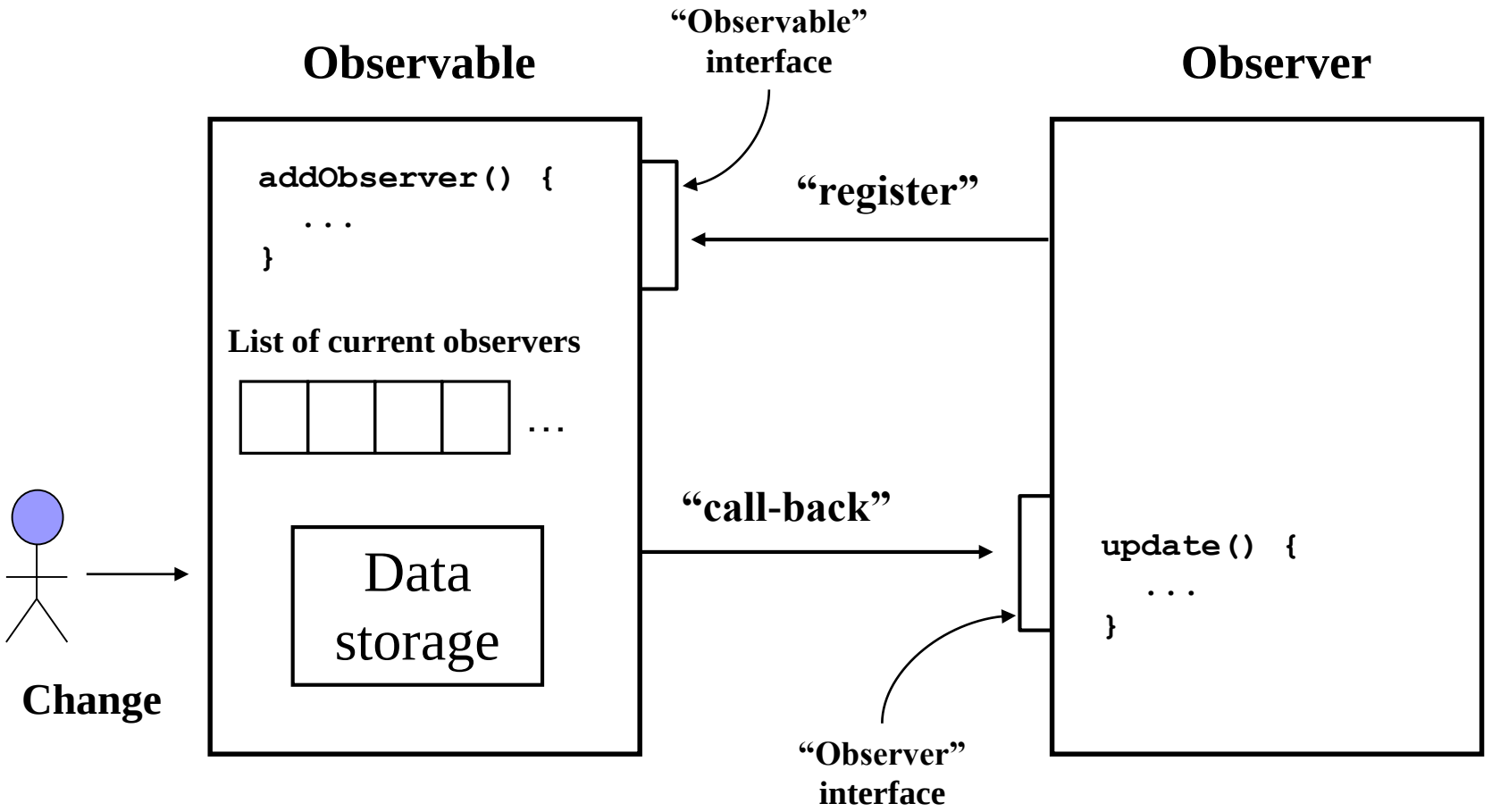
MVC Architecture



Model-View-Controller design pattern

Component	Responsibilities	Description
Model (GameWorld class)	Maintain data. Including (1) a GameWorld which **holds** a collection of **game objects** and other **state variables**	Business logic plus one or more data sources such as a database.
View (ScoreView and MapView Classes – New in assignment 2)	Display all or a portion of the data	The user interface that displays information about the model to the user.
Controller (Game Class)	Handle **events** that affect the model or view. Manages the flow of control in the game .This class accepts input in the form of keyboard **commands** from the human player and invokes appropriate methods in the game world to perform the requested **commands** – that is, to manipulate data in the game model.	The flow-control mechanism means by which the user interacts with the application.

Recall: The `Observer` Pattern (cont.)



Responsibilities

- Observables must
  - Provide a way for observers to “register”
  - Keep track of who is “observing” them
  - *Notify observers* when something changes
- Observers must
  - Tell observable it wants to be an observer *(“register”)*
  - Provide a method for the *callback*
  - Decide what to do when notified an observable has changed

Game Class High Level Steps

- Create "**Observable**" - gw = new GameWorld();
- Create an "**Observer**" for the map and create an "**Observer**" for the points
  - mv = new MapView() and pv = new PointsView(gw)
- Register observers: gw.addObserver(mv)
  gw.addObserver(pv);
- And more below:

```
// 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
```

(from assignment 2)

```
// Adding PointsView and MapView to the Form
add(BorderLayout.NORTH, pv);
add(BorderLayout.CENTER, mv);
```

IMPORTANT!

(Notes: (1) I will use PointView and ScoreView interchanged in this lecture
(2) Use BorderLayout for your containers)

GameWorld Class High Level Steps

(from assignment 2)

```
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.
}
```

Implements IGameWorld

(To be discussed in Proxy Design Pattern)

(To be discussed in Iterative Design Pattern)

```
public class GameWorld extends Observable {
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ F I E L D S ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
    //All state variables are stored here
    private GameCollection go;

    private boolean soundOn;

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ C O N S T R U C T O R S ~~~~~~~~~~~~~~~~~~~~~~//
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//

    /* This constructor sets all state values appropriately. This also creates the
     * collection necessary to hold all game objects.
     */
    public GameWorld(){
        go = new GameCollection();
        this.init();    //Then initialize the world
    }
```

(More on: Callbacks by passing GameWorld Proxy Later)

PointsView Class High Level Steps

(from assignment 2)

```
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)
     }
}
```

Showing only
One Attribute

```
public class PointsView extends Container implements Observer
{
    private Label pointsValueLabel;

    public PointsView()
    {
        // Instantiate text labels
        Label pointsTextLabel = new Label("Points:");
        // Instantiating value labels
        pointsValueLabel = new Label("XXX");
        // Set color
        pointsTextLabel.getAllStyles().setFgColor(ColorUtil.rgb(0,0,255));

        // Adding a container with a boxlayout
        Container myContainer = new Container();
        myContainer.setLayout(new BoxLayout(BoxLayout.X_AXIS));

        // Adding all labels in order
        myContainer.add(pointsTextLabel);
        this.add(myContainer);
    }
```

PointsView Class High Level Steps (Cont)

(from assignment 2)

```
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)
    }
}
```

Showing only One Attribute

```
// Updates the label text based on GameWorld state variables
// Update is called whenever the observable is updated
public void update(Observable o, Object arg)
{
    // Casting o as a GameWorld
    IGameWorld gw = (IGameWorld) arg;

    // Getting player score
    int score = gw.getPlayerScore();
    pointsValueLabel.setText("" + (score > 99 ? "" : "0") + (score > 9 ? "" : "0") + score);
    this.repaint();
}
```

Command Design Pattern

(from assignment 2)

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).

Example: Command Design Pattern

```
6  public class AddAsteroidCommand extends Command {
7      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
8      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ F I E L D S ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
9      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
10
11     private GameWorld gw;    //Reference to a Game World
12     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
13     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~ C O N S T R U C T O R S ~~~~~~~~~~~~~~~~~~~~~~~~//
14     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
15     /* There is only one constructor.
16      */
17     public AddAsteroidCommand( GameWorld gw ){
18         super( "Add Asteroid" );
19         this.gw = gw;
20     }
21     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
22     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ M E T H O D S ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
23     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//
24     //There is only one method to override the action performed
25     @Override
26     public void actionPerformed( ActionEvent e ){
27         gw.addAsteroid();
28         System.out.println("Add Asteroid.");
29     }
30
31 }
```

(Note: assignment 1)

Example: Command Design Pattern (Cont)

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.

Suggest:

In Game Constructor

```
//Add the new buttons that will be on the west border
Button addAsteroid = new Button ( "Add Asteroid" );

//Make all buttons look prettier
//Cyan 'Asteroid'
addAsteroid.getAllStyles().setBgTransparency( 255 );
addAsteroid.getUnselectedStyle().setBgColor( ColorUtil.rgb( 0, 150, 150 ) );
addAsteroid.getAllStyles().setFgColor( ColorUtil.rgb( 255, 255, 255 ) );

//Set some button padding
addAsteroid.getAllStyles().setPadding( TOP, 5 );
addAsteroid.getAllStyles().setPadding( BOTTOM, 5 );
```

Example: Command Design Pattern (Cont)

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

```
//~~~~~~~~~~~~~~~~~~~~~~~~~ ALL COMMANDS BELOW ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~//

//Declare all the needed commands for the buttons, keys, and side bar
AddAsteroidCommand myAddAsteroid = new AddAsteroidCommand(gw);

addAsteroid.setCommand( myAddAsteroid );

//Then the commands to the keys
addKeyListener( 'a', myAddAsteroid );
```

Example: Command Design Pattern Arrows and Space bar (Cont)

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.

```
//============ Binding play-mode specific key commands ===============
// Binding up arrow to increase ship speed (up arrow key code = -91)
addKeyListener(-91, myIncreaseShipSpeedCmd);
// Binding down arrow to decrease ship speed (down arrow key code = 92)
addKeyListener(-92, myDecreaseShipSpeedCmd);
// Binding left arrow to turn the ship left (left arrow key code = -93)
addKeyListener(-93, myTurnShipLeftCmd);
// Binding right arrow to turn the ship right (right arrow key code = -94)
addKeyListener(-94, myTurnShipRightCmd);
// Binding the space bar to fire a ship missile (space bar key code = -90) (this also binds the enter key)
addKeyListener(-90, myFireShipMissileCmd);
```

Iterative Design Pattern

(from assignment 2)

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**()).

Recall: Using An Iterator

```
/** This class implements a game containing a collection of SpaceObjects.
 *  The class assumes no knowledge of the underlying structure of the
 *  collection -- it uses an Iterator to access objects in the collection.
 */

public class SpaceGame {

    private SpaceCollection theSpaceCollection ;

    public SpaceGame() {
        //create the collection
        theSpaceCollection = new SpaceCollection();

        //add some objects to the collection
        theSpaceCollection.add (new SpaceObject("Obj1"));
        theSpaceCollection.add (new SpaceObject("Obj2"));
        ...
    }

    //display the objects in the collection
    public void displayCollection() {
        IIterator theElements = theSpaceCollection.getIterator() ;
        while ( theElements.hasNext() ) {
            SpaceObject spo = (SpaceObject) theElements.getNext() ;
            System.out.println ( spo ) ;
        }
    }
}
```

SpaceCollection With Iterator

```
/** This class implements a collection of SpaceObjects.
 *  It uses a Vector as the structure but does
 *  NOT expose the structure to other classes.
 *  It provides an iterator for accessing the
 *  objects in the collection.
 */
public class SpaceCollection implements ICollection {
    private Vector theCollection ;
    public SpaceCollection() {
        theCollection = new Vector ( );
    }
    public void add(Object newObject) {
        theCollection.addElement(newObject);
    }
    public IIterator getIterator() {
        return new SpaceVectorIterator ( ) ;
    }
    ...continued...
```

SpaceCollection With Iterator (cont.)

```
private class SpaceVectorIterator implements IIterator {
    private int currElementIndex;

    public SpaceVectorIterator() {
      currElementIndex = -1;
    }

    public boolean hasNext() {
        if (theCollection.size ( ) <= 0)  return false;
        if (currElementIndex == theCollection.size() - 1 )
          return false;
        return true;
    }

    public Object getNext ( ) {
        currElementIndex ++ ;
        return(theCollection.elementAt(currElementIndex));
    }
  } //end private iterator class
} //end SpaceCollection class
```

Example: Using An Iterative Design Pattern

```
//This will update the map eventually. For now it displays all the game objects
public void update( Observable o, Object arg ){
    System.out.println( "Map Width: " + Game.getMapHeight() + "  Map Height: "
            + Game.getMapWidth() );
    //Cast the Observable objects as the GameWorld first to access variables
    GameWorld gw = (GameWorld)o;
    GameCollection go = gw.getGameObjects();
    IIterator gameIterator = go.getIterator();
    while ( gameIterator.hasNext() ){
        System.out.println( gameIterator.getNext() );
    }
}
```

Note: This slide does not use Proxy Gameworld object – Use arg instead of o
See slides on Proxy Design Pattern next.

Proxy Design Pattern

(from assignment 2)

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).

Proxy Design Pattern (Cont)
IGameWorld Interface

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

```
5  public interface IGameWorld
6  {
7      // Returns the player score of the GameWorld
8      int getPlayerScore();
9      // Get other games attributes ........
```

Proxy Design Pattern (Cont) notifyObservers - Whenever you need to update the Views

```
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.
}
```

```
gameObjects.add(asteroid);

System.out.println("A new asteroid has been created.");

this.setChanged();
this.notifyObservers(new GameWorldProxy(this));
```

Proxy Design Pattern

(from assignment 2)

Remember Keyword: forward!

```
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.
}
```

```
 8 public class GameWorldProxy extends Observable implements IGameWorld
 9 {
10     private GameWorld gw;
11
12     public GameWorldProxy(GameWorld gw)
13     {
14         this.gw = gw;
15     }
16
17     // Returns the player score of the GameWorld
18     public int getPlayerScore()
19     {
20         return gw.getPlayerScore();
21     }
22
```

Proxy Design Pattern (not related to Assignment 2)

(Inside Update – Observer – PointView)

(Use Data passed over from Observeable)

Showing only One Attribute

```
// Updates the label text based on GameWorld state variables
// Update is called whenever the observable is updated
public void update(Observable o, Object arg)
{
    // Casting o as a GameWorld
    IGameWorld gw = (IGameWorld) arg;

    // Getting player score
    int score = gw.getPlayerScore();
    pointsValueLabel.setText("" + (score > 99 ? "" : "0") + (score > 9 ? "" : "0") + score);
    this.repaint();
}
```