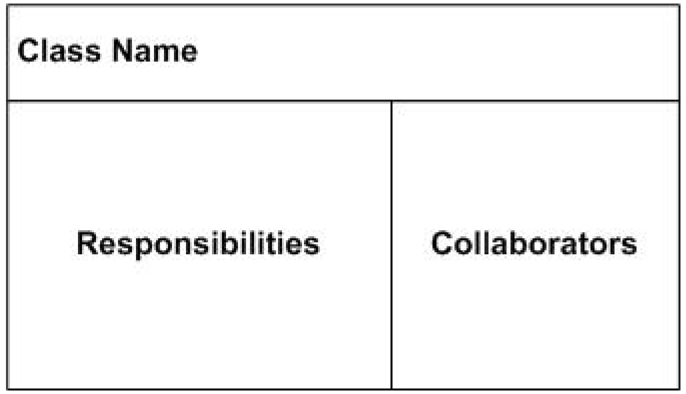
CSc 133 Lecture Notes

Assignment 1 Discussion (Spring 2019)

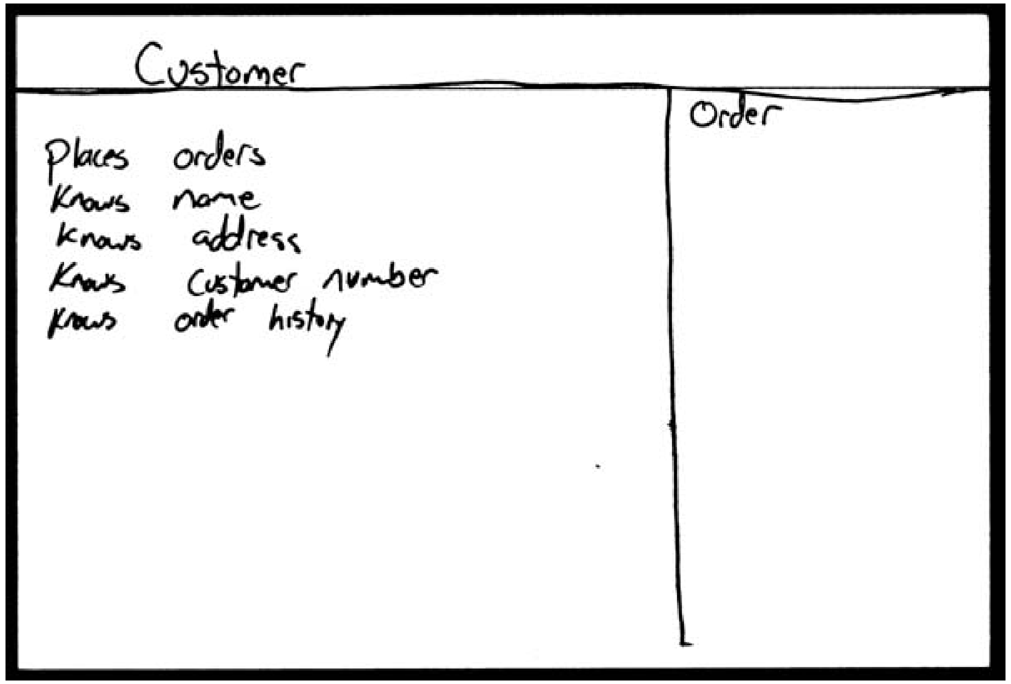
Key Points

- **Read the assignment MANY times.**
  - **See appendix at the end (please not skip it)**
  - **Sample code provided there.**
- **Analysis**
  - **Identify Suitable Classes**
    - **Use hints from assignment**
    - **Use CRC form: Class Responsibility Collaboration**

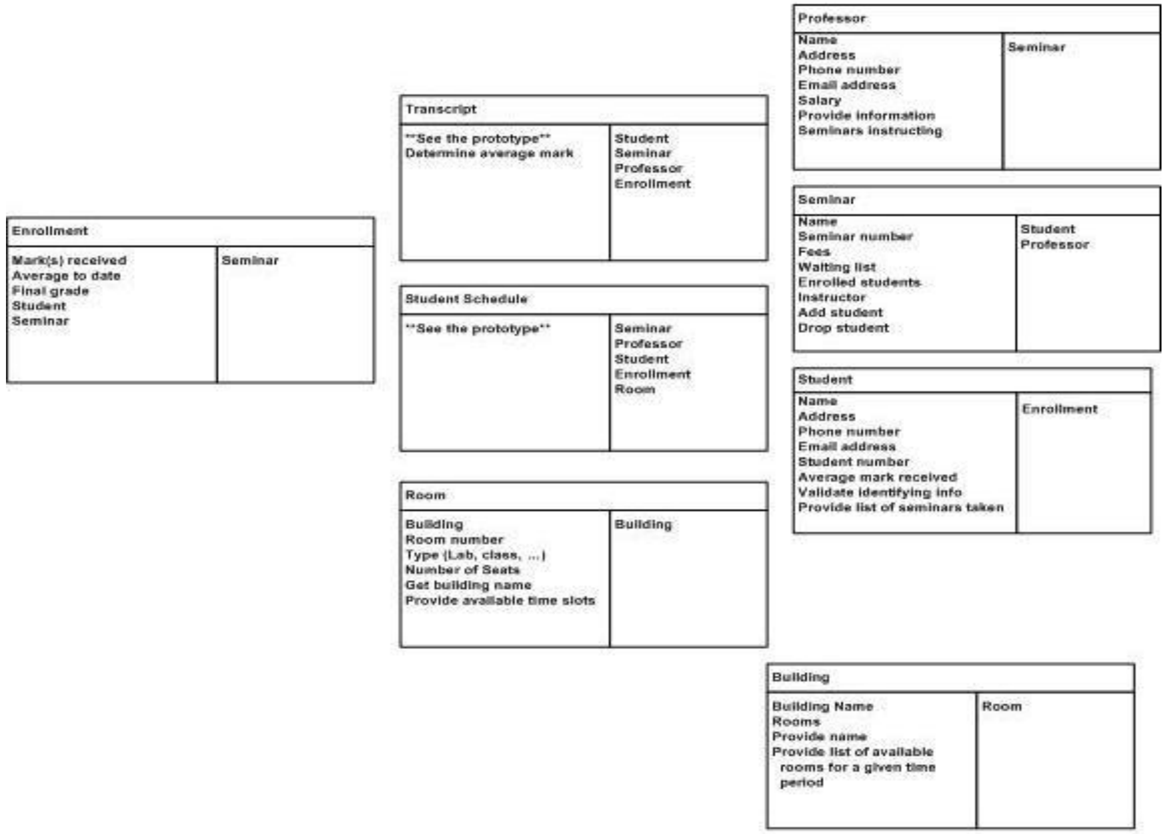


- **Check your work again your requirements**

Example of CRC card



Customer	
Places orders	Order
Knows name	
knows address	
Knows customer number	
Knows order history	



Professor	
Name	Seminar
Address	
Phone number	
Email address	
Salary	
Provide information	
Seminars instructing	

Transcript	
See the prototype	Student
Determine average mark	Seminar
	Professor
	Enrollment

Seminar	
Name	Student
Seminar number	Professor
Fees	
Waiting list	
Enrolled students	
Instructor	
Add student	
Drop student	

Enrollment	
Mark(s) received	Seminar
Average to date	
Final grade	
Student	
Seminar	

Student Schedule	
See the prototype	Seminar
	Professor
	Student
	Enrollment
	Room

Student	
Name	Enrollment
Address	
Phone number	
Email address	
Student number	
Average mark received	
Validate identifying info	
Provide list of seminars taken	

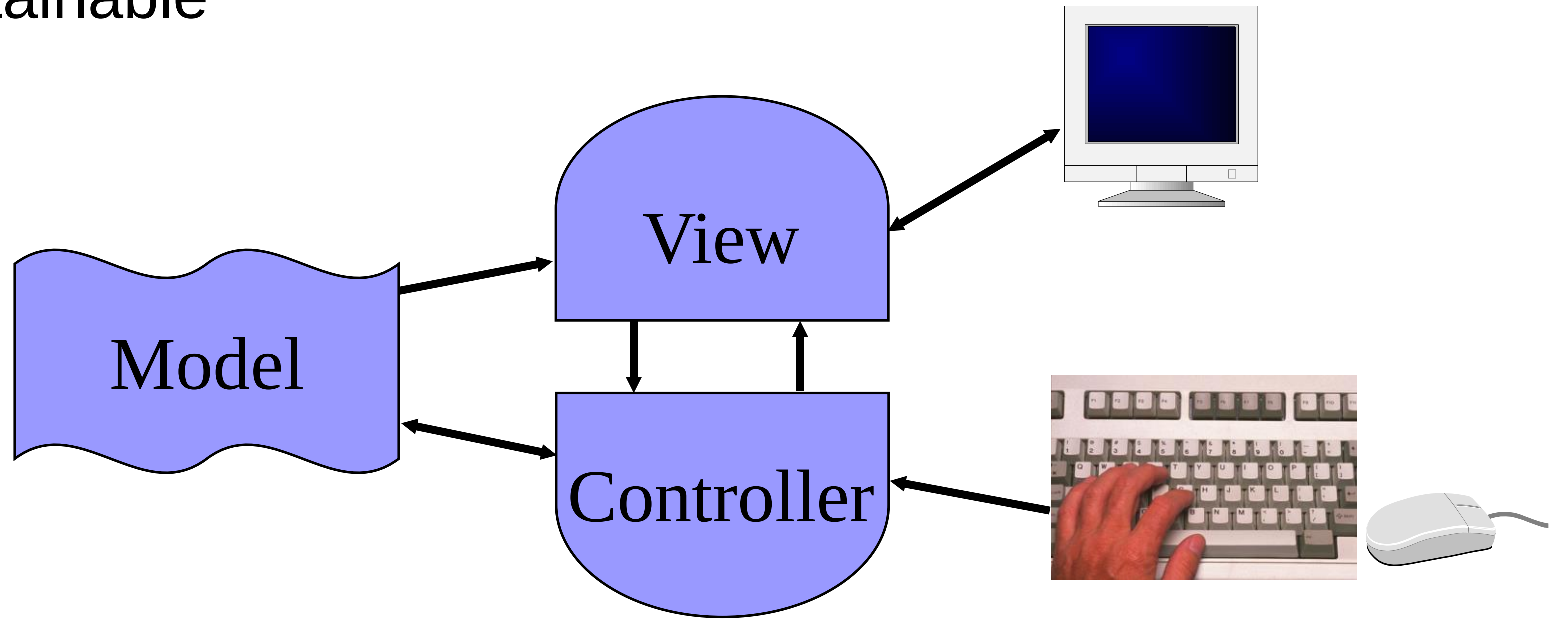
Room	
Building	Building
Room number	
Type (Lab, class, ...)	
Number of Seats	
Get building name	
Provide available time slots	

Building	
Building Name	Room
Rooms	
Provide name	
Provide list of available rooms for a given time period	

Source: http://agilemodeling.com/artifacts/crcModel.htm

Model-View-Controller
(Assignment 1 Game Structure)

- Partitions application into **Model, View, Controller** so that it is
  - scalable
  - maintainable



Model-View-Controller design pattern (Cont)

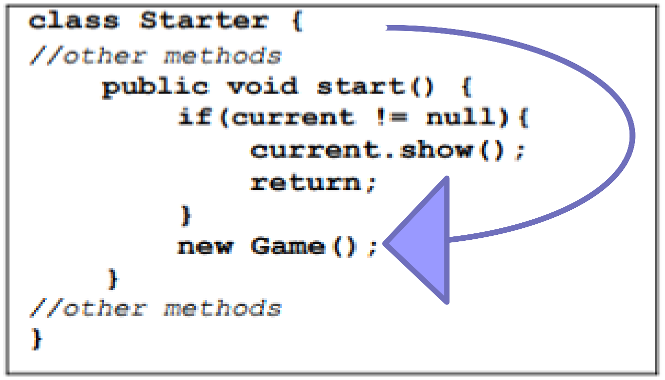
Component	Purpose	Description
Model	Maintain data	Business logic plus one or more data sources such as a database.
View	Display all or a portion of the data	The user interface that displays information about the model to the user.
Controller	Handle events that affect the model or view	The flow-control mechanism means by which the user interacts with the application.

Model-View-Controller design pattern (Cont)

Component	In our assignment # 1 context (Spring 2019)
Model (GameWorld)	A game in turn contains several components, including (1) a GameWorld which **holds** a collection of **game objects** and other **state variables**. Later, we will learn that a component such as **GameWorld** that holds the program's data is often called a model.
View (Future: **Map** and **Score** Views)	In this first version of the program the top-level Game class will also be responsible for **displaying** information about the state of the game. In future assignments we will learn about a separate kind of component called a **view** which will assume that responsibility.
Controller (Game)	The **top-level Game class** also manages the **flow of control** in the game (such a class is therefore sometimes called a controller). The controller enforces rules such as what actions a player may take and what happens as a result. 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.

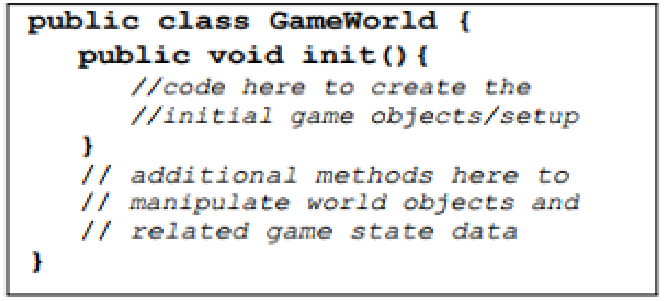
Model-View-Controller design pattern (Cont)

CN1 Starter Class – **Just add new Game()**



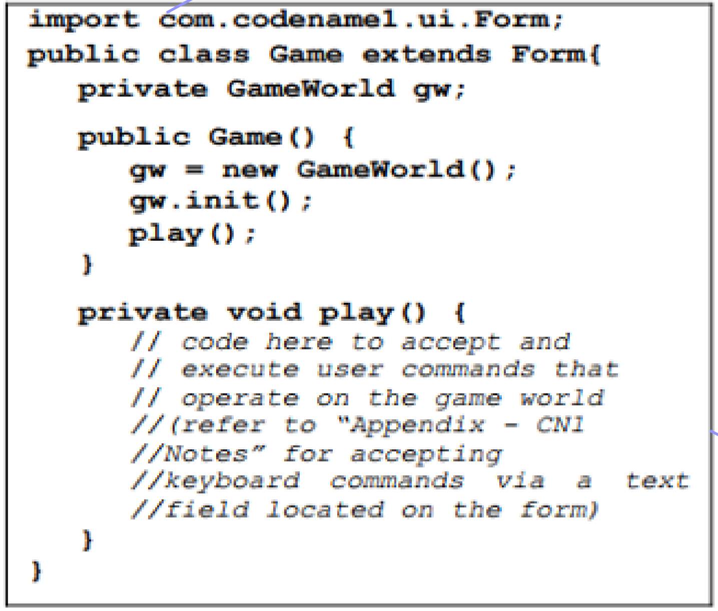
```
class Starter {
//other methods
    public void start() {
        if(current != null){
            current.show();
            return;
        }
        new Game();
    }
//other methods
}
```

Model
(GameWorld)



```
public class GameWorld {
   public void init(){
       //code here to create the
       //initial game objects/setup
   }
   // additional methods here to
   // manipulate world objects and
   // related game state data
}
```

Controller
(Game)



```
import com.codename1.ui.Form;
public class Game extends Form{
    private GameWorld gw;

    public Game() {
        gw = new GameWorld();
        gw.init();
        play();
    }

    private void play() {
        // code here to accept and
        // execute user commands that
        // operate on the game world
        //(refer to "Appendix - CN1
        //Notes" for accepting
        //keyboard commands via a text
        //field located on the form)
    }
}
```

View (in A1 only)
(Game – Will
be having a separate
Components in A2)

Controller: Process input commands (from A1 Appendix)

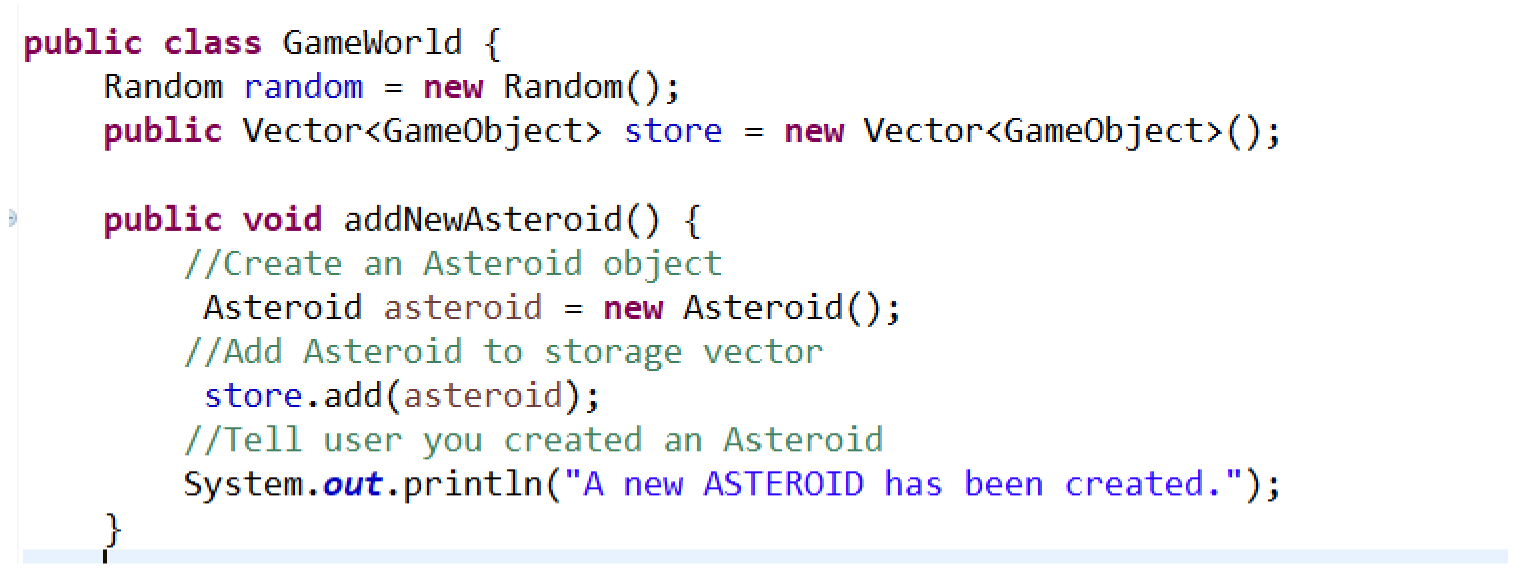
```
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.Label;
import com.codename1.ui.TextField;
import com.codename1.ui.events.ActionEvent;
import java.lang.String;
private void play()
      {
      Label myLabel=new Label("Enter a
      Command:"); this.addComponent(myLabel);
      final TextField myTextField=new TextField();
      this.addComponent(myTextField);
      this.show();
      myTextField.addActionListener(new ActionListener(){

            public void actionPerformed(ActionEvent evt) {

            String sCommand=myTextField.getText().toString();
            myTextField.clear();
            switch (sCommand.charAt(0)){
                  case 'e':
                        gw.eliminate();
                        break;
                  //add code to handle rest of the commands
            } //switch
      } //actionPerformed
      } //new ActionListener()
      ); //addActionListener
      } //play
```

```
switch (sCommand.charAt(0)) {
      case 'a':
            gw.addAsteroid();
            break;
```

End of Phase 1: Do Design Work & Drawing a Sketch, Code & Test

- Do a short design work
- Draw a UML sketch of the current classes (Starter, Game, GameWorld)
- Code it
- Run Test Case # 1 (Can you pass this test ?)
- Refactor

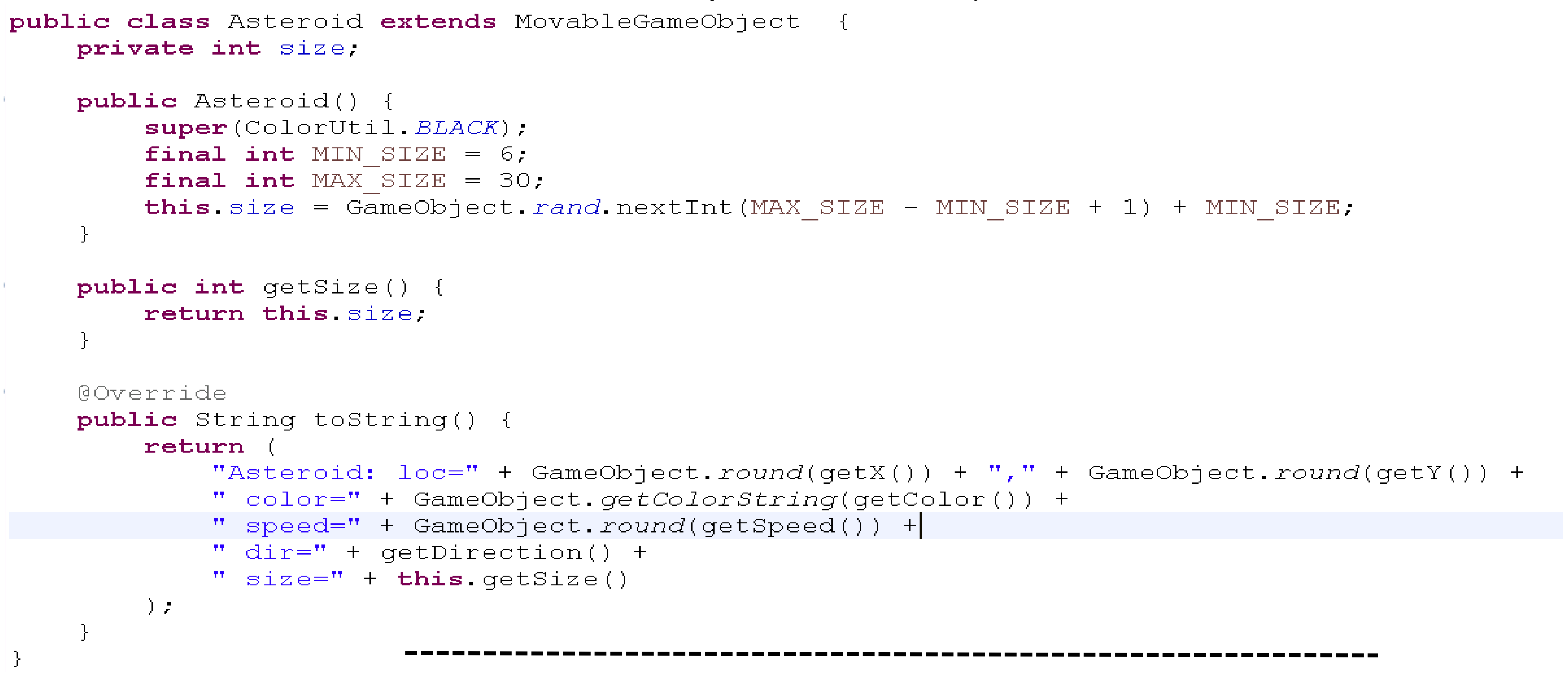
Model: GameWorld Process Add Asteroid Command



```java
public class GameWorld {
    Random random = new Random();
    public Vector<GameObject> store = new Vector<GameObject>();

    public void addNewAsteroid() {
        //Create an Asteroid object
         Asteroid asteroid = new Asteroid();
        //Add Asteroid to storage vector
         store.add(asteroid);
        //Tell user you created an Asteroid
        System.out.println("A new ASTEROID has been created.");
    }
```

And others command

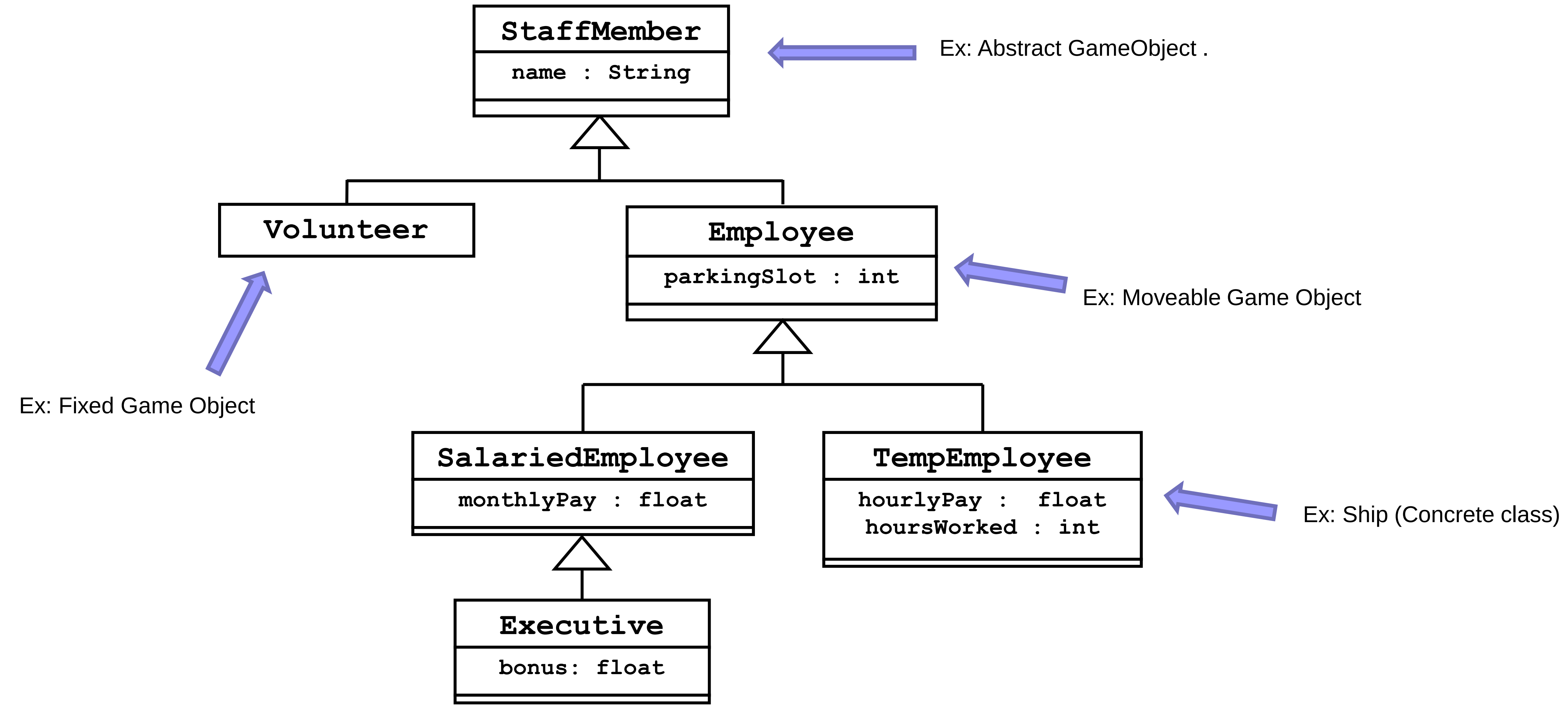
Asteroid Concrete Class (Sample Only)



```java
public class Asteroid extends MovableGameObject  {
    private int size;

    public Asteroid() {
        super(ColorUtil.BLACK);
        final int MIN_SIZE = 6;
        final int MAX_SIZE = 30;
        this.size = GameObject.rand.nextInt(MAX_SIZE - MIN_SIZE + 1) + MIN_SIZE;
    }

    public int getSize() {
        return this.size;
    }

    @Override
    public String toString() {
        return (
            "Asteroid: loc=" + GameObject.round(getX()) + "," + GameObject.round(getY()) +
            " color=" + GameObject.getColorString(getColor()) +
            " speed=" + GameObject.round(getSpeed()) +
            " dir=" + getDirection() +
            " size=" + this.getSize()
        );
    }
}
```

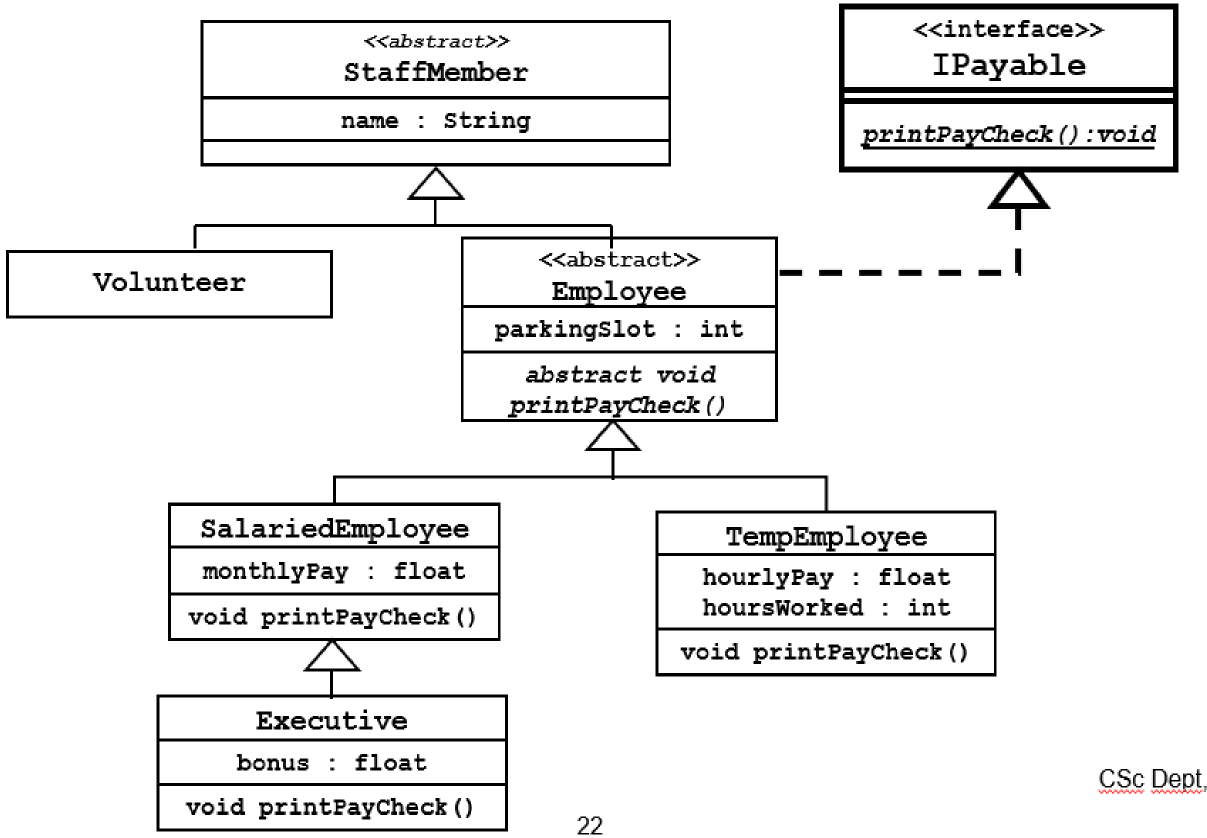
Now, add others commands and their objects

Now, think sub-classes (Connecting to our Lecture Materials)



Think sub-classes (Cont)

- StaffMember hierarchy using Interfaces:



22

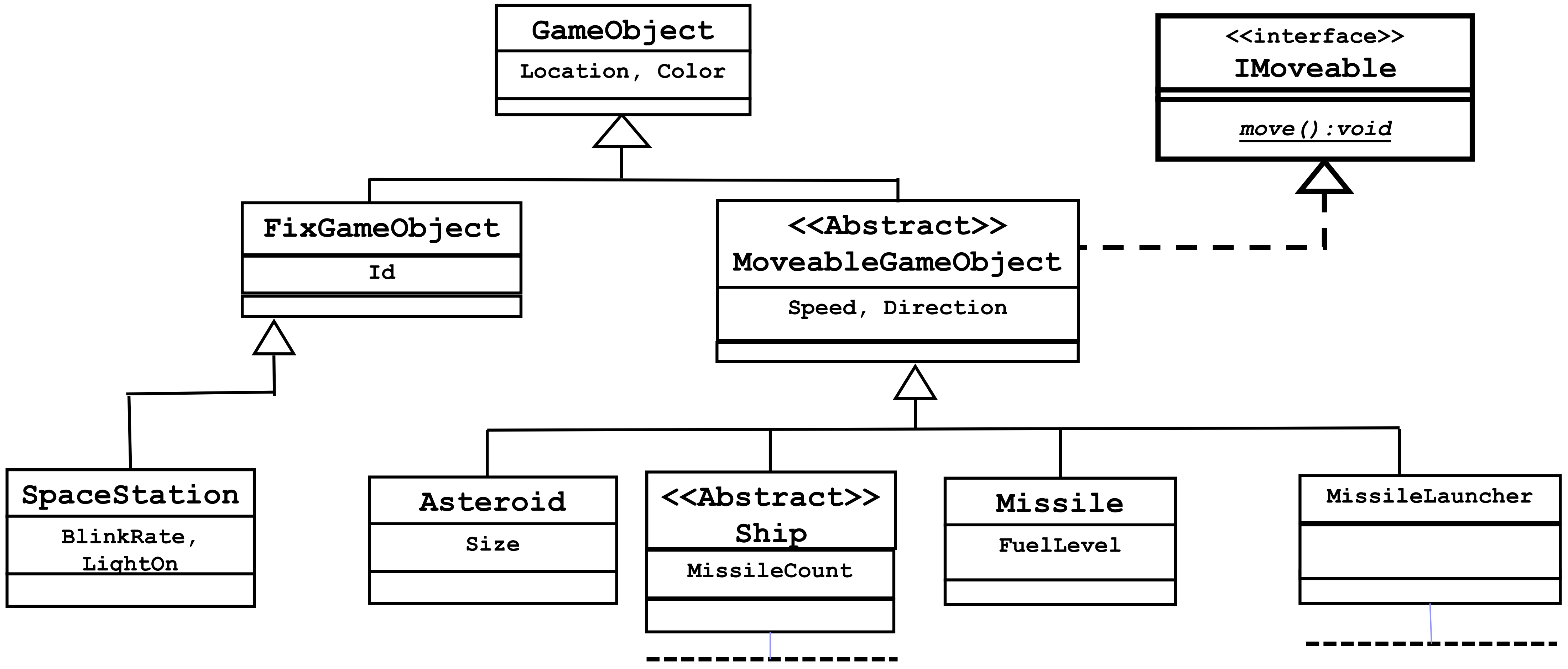
CSc Dept, CSUS

Can your work support Runtime Polymorphism safely?

```
for (int i=0; i<theWorldVector.size(); i++) {
    if (theWorldVector.elementAt(i) instanceof IMovable) {
        IMovable mObj = (IMovable)theWorldVector.elementAt(i);
        mObj.move();
    }
}
```

Note: See lecture on Polymorphism
And   "Additional details" note in Assignment 1.

Think sub-classes – Example Only (Not Complete)

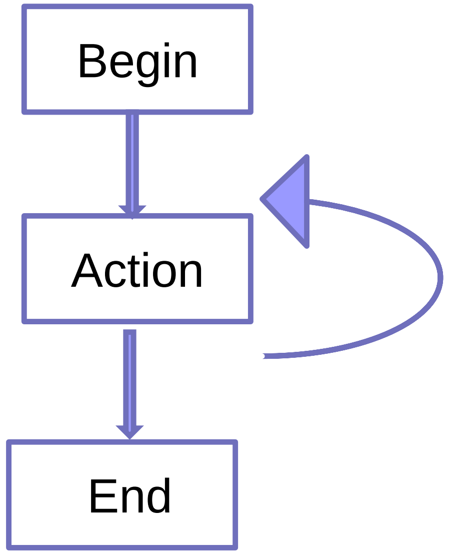


(More classes down here)

End of Phase 2: Do Design Work & Drawing a Sketch, Code & Test

- Do a short design work
- Draw a UML sketch of the current classes (Asteroid, Missile, Station, PS, NPS, Launchers..)
- Code each class
- Run Test Case # 2 (Can you pass this test ?)
  - Check output format and correct state for each object
- Refactor

Start Phase 3 and 4: Do Design Work & Drawing a Sketch, Code & Test



- Do a short design work
- Implement **animation functions** (i.e. ‘t’, ‘b’, ‘r’, ‘>’)
- Implement **collision functions** (i.e. ‘k’, ‘e’, ‘E’, ‘c’ …)
- Code each function
- Run Test Case # 3 (Can you pass this test ?)
  - Check output format and correct state for each object by running both ‘m’ and ‘p’ commands.
- Refactor

- Do a short design work
- Implement game end **function** (i.e. ‘q’, or player ran out of lives)
- Code the function
- Run test case # 4 (Can you pass this test ?)
- Refactor

Additional information (Do not forget these in your UML Diagram!)

- Including Interfaces (2)
- Declare Methods, Attributes, Modifier (i.e. Private)
- Include external packages and class names (reference in UML diagram)
  - com.codename1.ui.Form ……
- Other classes: Starter, Game, GameWorld (Do not forget – See slide 7)
- Other relationships: Composition, Dependency, Association.
- Concrete classes have to toString method (see Asteroid Slide 11)
- Use Codename one provided Point **or Point2D to hold x,y** instead of int x,int y – For object location (See assignment 1 note)

Coding after completion of UML Diagram

- Expect to return to UML diagram to make changes once identified issues identifying through coding or testing
- Including comments and correct use of variables, class names, package names
- Proper use of inheritance, encapsulation, polymorphism, interface

Other Information

- Can I code the entire A1 without UML diagram? Answer is no.
- Can I automatically generate UML diagram from my Code (code/fix) ? Answer is no.
- Prototyping some key concepts ? Yes, I recommend it.
  - I.e. Input processing ?
  - Calling the game object?

Other Information

- Validate the complete set of commands
- Check for input errors and boundary conditions
  - Your program should not be crashed under these conditions (software quality). Output error handling text when required.
    - Can you program handle this as input: !@$%^Aa ?
  - Conform to expected output format (i.e. decimal points)

Turning the A1 assignment (Suggestions)

- Check the deliverable section of the assignment
- Refresh the dist folder ← (Watch out for old code: No “Hello World” or A0)
- Run the launch command to ensure the program can launch correctly
- Turn in your work before Feb **25th before 11:59 PM**
- Have fun ☺ but not procrastinated