CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

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

Assignment #1: Class Associations & Interfaces

Due Date: Thursday, March 1st [2+ weeks]

Introduction

This semester we will be studying object-oriented computer graphics programming by developing a program which is a simplified version of a classic Arcade and home computer game called *Breakout.* In this game the human player controls a paddle which bounces a moving ball into a wall of “bricks”. Hitting a brick removes it and scores points; additional bonus points are obtained by removing all the bricks and by other means.

If you have never seen the original Breakout game, visit Dr. Clevenger’s webpage at http://gaia.ecs.csus.edu/~clevengr/133 and click on the “Breakout” link. The version posted there is very close to the original; as the semester progresses you will be creating a similar game but with additional features. It is not necessary to be familiar with the original game to do any of the assignments during the semester (but seeing how the original game plays might help you understand the general idea of what we’re going to be doing).

The initial version of your game will be *text-based*. As the semester progresses we will add graphics, animation, and sound. The goal of this first assignment is to develop a good initial class hierarchy and control structure, and to implement it in Java. This version uses keyboard input commands to control and display the contents of a “game world” containing a set of objects in the game. In future assignments many of the keyboard commands will be replaced by interactive GUI operations, but for now we will simply simulate the game in “text mode” with user input coming from the keyboard and “output” being lines of text on the screen.

Program Structure

Because you will be working on the same project all semester, it is extremely important to organize it correctly from the beginning. Pay careful attention to the class structure described below and make sure your program follows this structure accurately.

The primary class encapsulates the notion of a ***Game***. A game in turn contains several components: (1) a ***GameWorld*** which holds a collection of *game objects* and other state variables, and (2) a set of methods to accept and execute user commands. Later, we will learn that a component such as *GameWorld* (that holds the program’s data) is often called a *model*.

The top-level *Game* class also encapsulates 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.

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.

The program also has a class named ***Starter*** which has the **`main(String[]args)`** method. **`main()`** does one thing – construct an instance of the ***Game*** class. The game constructor then instantiates the game components, and starts the game by calling a method named **`play()`**. The **`play()`**method then accepts keyboard commands from the player and invokes appropriate methods to manipulate the game world and display the game state.

The following shows the pseudo-code implied by the above description.

```
class Starter {
   main() {
      Game g = new Game();
   }
}
```

```
class GameWorld {
   // code here to hold and
   // manipulate world objects
   // and related game state data
}
```

```
class Game {
   private GameWorld gw;

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

   private void play() {
      // code here to accept and
      // execute user commands that
      // operate on the GameWorld
   }
}
```

Game World Objects

The game world can contain two abstract kinds of game objects: *Ball* and *Obstacle*. There are several types of obstacles: *Bricks*, which comprise the wall against which the ball is bounced; *Edges*, which form the boundaries of the playing field; and *Paddle*, which the player uses to bounce the ball into the wall of bricks. There are two types of bricks: *Regular Bricks*, which simply cause the ball to rebound, and *Speed Bricks*, which cause the ball to rebound as well as increase speed. There are three types of edges: *Top Edges, Side Edges, and Bottom Edges* (which lie at the top, left/right, and bottom edges of the playing field respectively).

The various game objects have attributes (fields) and behaviors ("functions" or "methods") as defined below. These definitions are requirements which must be properly implemented in your program.

- All game objects have a *location*, defined by floating point values X and Y. The point (X,Y) is the center of the object. The origin of the "world" is the lower left hand corner of the screen (although we will change this later; think of it this way for now). All game objects provide the ability for other objects to obtain their location. Some game objects have changeable locations (that is, they are moveable), while others have locations that are not allowed to change. The program must properly implement this requirement.

- All game objects have a *color*, defined by a value of Java type java.awt.Color. All game objects provide the ability for other objects to obtain their color. Some types of game objects have color which is changeable (meaning that they provide an interface which allows other objects to modify their color), while other objects have a color which cannot be changed once the object is created. Objects which are not explicitly stated as having the ability to change their color must not support that ability.

- Some game objects are *moveable*, meaning that they provide an interface that allows other objects to control their movement. Telling a moveable object to *move()* causes the object to update its location based on its current speed and direction. Moveable objects can also be told to do three other operations: *bounceHorizontal()*, which instructs the object to reverse its horizontal movement direction; *bounceVertical()*, which instructs the object to reverse its vertical movement direction; and *alterCourse()*, which instructs the object to make a small random change in its movement direction.

- Some game objects are *positionable*, meaning that they provide an interface that allows other objects to change their location after they have been created. Note that the difference between *positionable* and *moveable* is that other objects can *set the location* of *positionable* objects whereas other objects can only request that a *movable* object update its <u>own</u> location according to its current speed and heading.

- All bricks have attributes *width* and *height*, along with *point value* (the number of points scored when a ball hits them). Speed bricks also have an attribute *speed factor*, the factor by which the speed of the ball increases when the brick is hit. Bricks can have their color changed but not their location. Speed bricks have a different color than regular bricks.

- Balls have an attribute *diameter*. Balls are *movable* and do not allow their color to be changed.

- Top Edges and Bottom Edges are horizontal lines at the position defined by their Y location; Side Edges are vertical lines at the position defined by their X location. Edges can neither change location nor color.

- Paddles have attributes *width* and *height*, along with values *minX* and *maxX* representing the leftmost and rightmost allowable positions for the paddle. Paddles also have a *mode* attribute: they are either in *normal* mode or *random* mode. Paddles are *positionable* and have changeable color.

The preceding paragraphs imply a certain set of *associations* between classes – an <u>inheritance</u> class hierarchy among game objects, <u>interfaces</u> presented by certain classes (e.g. for *changeable color, moveable,* and *positionable* objects), and <u>aggregation</u> associations between objects and where they are held. You must develop a UML diagram for the relationships, and then implement it in a Java program. Appropriate use of encapsulation, inheritance, aggregation, abstract classes, and interfaces are important grading criteria.

<u>Game Play</u>

The playing field is a rectangular area bounded by four edges (top, left and right sides, and bottom). Inside the playing field near (but not touching) the top is a wall made up of three rows of bricks. Each row of bricks initially stretches unbroken across the entire width of the playing field, and the rows of bricks lie next to each other. There is a space between the uppermost row of bricks and the top edge of the playing field, and that space is at *least* as high as the height of one row of bricks. Each row of bricks contains a *random assortment* of regular and speed bricks. For this assignment all bricks are to be the same size (this may change later).

The paddle is in the center near the bottom when the game starts. The player gets three balls, played one at a time. Each ball starts at a random location between the paddle and the brick wall, moving downward. The player can move the paddle to the left or right. If the ball hits the paddle then the ball changes direction and moves upward; if the paddle is in “random” mode at the time the ball hits it then the ball also acquires a small additional random change to its direction. If the ball hits the bottom edge then the ball is lost; if there are no more balls left then the game is over. If the ball hits a brick then the ball bounces (changes direction), the brick is removed, and the player's score is incremented by the point value of the brick. If the ball hits a side or top edge then it bounces (changes direction).

The game keeps track of three "game state" values: elapsed time, balls remaining, and current score. The objective is to obtain the highest score in the least amount of elapsed time.

Commands

Once the game world has been created and initialized, the game constructor should call method **`play()`** to actually begin the game. **`play()`** repeatedly calls a method named **`getCommand()`**, which prompts the user for single-character *commands*. Commands should be input using the Java InputStreamReader, BufferedReader, or Java Scanner class (see the "Java Coding Notes" Appendix at the end, and also page 23 in the text).

The allowable input commands and their meanings are defined below. Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. Each command returned by **`getCommand()`** should invoke a corresponding function in the game world, as follows:

'**r**', '**l**' – move (change the position of) the paddle to the **r**ight (or **l**eft) by one unit.

'**i**' – **i**ncrease the speed of the ball by a factor of 20% (later we will make the increase dependent on a particular brick; for now we'll assume a constant increase).

'**a**' – **a**nnihilate (remove) a brick from the wall. The brick to be removed should be chosen randomly from among the remaining bricks in the wall. Eventually we will require the ball to hit a brick to remove it, but for now we'll allow a brick to be removed at any time regardless of the ball's position. Removing a brick increases the player's score by the point value of the brick.

'**s**' – **s**ide edge hit (i.e., pretend that the ball has hit a side of the playing field; as above, and similarly for the following commands, we will allow this for now regardless of where the ball really is). The effect of hitting a side is that the ball starts moving in the opposite horizontal direction.

'**u**' – **u**pper edge hit (i.e., pretend that the ball has hit the top edge of the playing field). The effect of hitting the top edge is that the ball starts moving in the opposite vertical direction.

'**b**' – **b**rick hit (i.e., pretend that the ball has hit a brick in the wall). The effect of hitting a brick is that the ball starts moving in the opposite vertical direction.

'**c**' – change paddle mode (i.e., if the paddle is in normal mode then it switches to random mode, and vice versa).

'**p**' – **p**addle hit (i.e., pretend that the ball has hit the paddle). The effect of the ball hitting the paddle is that the ball starts moving in the opposite vertical direction. If the paddle is in random mode when the ball hits it, then an additional effect occurs: a small random change is added to the ball's horizontal movement direction.

'**n**' – **n**ext ball (i.e., pretend that the previous ball has hit the bottom edge and been lost). If there are balls remaining then a new ball is put into play; otherwise, the game is over.

'**t**' – tell the game world that the "game clock" has ticked. A clock tick in the game world has the following effects: (1) all moveable objects are told to update their positions according to their current heading and speed, and (2) the "elapsed game time" is incremented by one.

'**d**' – print a **d**isplay (output lines of text) giving the current game state values, including: (1) the current game clock value, (2) the current score, and (3) the number of balls left in the game. Output should be well labeled in easy-to-read format.

'**m**' – print a "**m**ap" showing the current world state (see below).

'**q**'– **q**uit, by calling the method **`System.exit(0)`** to terminate the program. Your program should first confirm the user's intent to quit before actually exiting.

Additional Details

- Each command must be encapsulated in a separate method in the Game class (later we will move those methods to other locations). Most of the game class command actions also invoke related actions in *GameWorld*. When the *Game* gets a command, it invokes one or more methods in the *GameWorld*, rather than itself manipulating game world objects.
- All classes must be designed and implemented following the guidelines discussed in class:
  - *All data fields must be private,*
  - *Accessors / mutators must be provided, but only where the design requires them,*
  - *Game world objects may only be manipulated by calling methods in the game world. It is never appropriate for the controller to directly manipulate game world data attributes.*
- Moveable objects need a way to keep track not only of their current location, but how that location is supposed to change when their *move(), bounceVertical(), bounceHorizontal(),* and *alterCourse()* methods are invoked. A simple way to do this is to have the object maintain its velocity (speed and direction) in the form of two values: speed in the X direction, and speed in the Y direction. When told to move, the object can then compute its new location by adding these to the X and Y components of its location respectively. (For example, if the object is moving at a rate of xSpeed=2 and ySpeed=3 and its current location is (5,5), telling the object to *move()* would cause it to update its location to be (7,8)). Telling an object to *bounce* (start moving in the opposite direction either horizontally or vertically) can be implemented simply by negating the speed in that direction. Similarly, *alterCourse()* can be accomplished by adding a small randomly-generated value to either xSpeed or ySpeed.
- For this assignment all output will be in text on the console; no "graphical" output is required. The "map" (generated by the '**m**' command) will simply be a set of lines describing the objects currently in the world, as follows. First, for each of the three rows of bricks, print a row containing one 'r' character for each position occupied by a regular brick, an 's' character for each position occupied by a speed brick, and a blank for each empty position. Then, print one line each for the other objects in the world. Thus the output for a single 'm' command might appear similar to:

```
rrrs ssrs s  ssrsr s
ss rrr  rssrr   rrss
rrrrrsssss  ssrrss r

Ball: x=130,y=65 Color=[0,255,0]...
Paddle: x=40,y=20 Color=[128,128,255]...
TopEdge: x=100,y=300 Color=[0,0,0]
BottomEdge: x=100,y=0 Color=[0,0,0]
LeftEdge: x=0,y=150 Color=[0,0,0]
RightEdge: x=200,y=150 Color=[0,0,0]
```

Note that this example assumes that it takes 20 bricks to span the playing field (e.g. that the field is 200 units wide by 300 units high and each brick is 10 units wide). The actual choice of world units is up to you. The output for the Ball and Paddle must include all attributes defined for those objects. Note that the appropriate mechanism for implementing this output is to override the **`toString()`** method in each concrete game object class so that it returns a String describing itself. See the coding notes Appendix for additional details.

- The program must handle any situation where the player enters an illegal command – for example, a command to remove (annihilate) a brick when there are no bricks in the world – by printing an appropriate condition-specific error message on the console (for example, "Cannot execute 'annihilate' – no bricks exist in the world") and otherwise simply waiting for a valid command.

- The program is not required to have any code that actually checks for collisions between objects; that's something we'll be adding later. For now, the program simply relies on the user to say when such events have occurred, using for example the '**`b`**' and '**`p`**' commands.

- You must follow standard Java coding conventions:
  - *class names always start with an <u>upper case</u> letter,*
  - *variable names always start with a <u>lower case</u> letter,*
  - *compound parts of compound names are capitalized (e.g., `myExampleVariable`),*
  - *Java interface names should start with the letter "I" (e.g., `ISteerable`).*

- Your program must be contained in a Java *package* named "**`a1`**" ("a-one", lower-case). Specifically, every class in your program must be defined in a separate **`.java`** file which has the statement "**`package a1;`**" as its first statement. Further, it must be possible to execute the program from a command prompt by changing to the directory containing the **`a1`** package and typing the command: "**`java a1.Starter`**". *<u>Verify for yourself that this works correctly</u>* from a command prompt before submitting your program. Note that the appropriate use of *sub-packages* within package "**`a1`**" is both acceptable and encouraged. See pages 16-18 and 29-31 in the text for more on the use of Java packages.

- Single keystrokes don't invoke action -- the human hits "enter" after each key command.

- You are not required to use any particular data structure to store the game world objects, but *all your game world objects must be stored in a <u>single</u> structure*. In addition, your program must be able to handle changeable numbers of objects at runtime – so you can't use a fixed-size array, and you can't use individual variables.

Consider either the Java *ArrayList* or *Vector* class for implementing this storage. Note that Java Vectors (and ArrayLists) hold elements of type "Object", but you will need to be able to treat the Objects differently depending on the type of object. You can use the "**`instanceof`**" operator to determine the type of a given Object, but be sure to use it in a polymorphically-safe way. For example, you can write a loop which runs through all the elements of a world Vector and processes each "movable" object with code like:

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

- Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged, but not required.

- Students are encouraged to ask questions or solicit advice from the instructor outside of class. But have your UML diagram ready – it is the first thing the instructor will ask to see.

Deliverables

Submission is to be done using the ***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 submitted ZIP file contains the proper subpackage hierarchy. Also, be sure to take note of the requirement (stated in the course syllabus) for keeping a *backup copy* of all submitted work. If you are not familiar with submitting assignments using SacCT, see the paper in the Handouts section of my 133 web page.

All submitted work must be strictly your own!

Appendix – Java Coding Notes

Input Commands

In Java 5.0 and higher, the **`Scanner`** class will get a line of text from the keyboard:

```
     Scanner in = new Scanner (System.in);
     System.out.print (“Input some text:”);
     String line = in.nextLine();
  or int aValue = in.nextInt();
```

Random Number Generation

The class used to create random numbers in Java is **`java.util.Random`**. This class contains methods **`nextInt()`**, which returns a random integer from the entire range of integers, **`nextInt(int)`**, which returns a random number between zero (inclusive) and the specified integer parameter (exclusive), and **`nextBoolean()`**, which returns a random boolean value (either true or false). The Random class is discussed on pg. 28 of the text.

Output Strings

The Java routine **`System.out.println()`** can be used to display text. It accepts a parameter of type String, which can be concatenated from several strings using the “+” operator. If you include a variable which is not a String, it will convert it to a String by invoking its *toString()* method. For example, the following statements print out "The value of I is 3":

```
int i = 3 ;
System.out.println ("The value of I is " + i);
```

Every Java class provides a *toString()* method. Sometimes the result is descriptive; for example, an object of type **`java.awt.Color`** returns a description of the color. However, if the *toString()* method is the default one inherited from Object, it isn't very descriptive. Your own classes should override *toString()* and provide their own String descriptions – including the *toString()* output provided by the parent class if that class was also implemented by you.

For example, suppose there is a class **`Ball`** with attribute *radius*, and a subclass of **`Ball`** named **`ColoredBall`** with attribute *myColor* of type java.awt.Color. An appropriate *toSring()* method in **`ColoredBall`** might return a description of a colored ball as follows:

```
public String toString() {
     String parentDesc = super.toString();
     String myDesc = "ColoredBall: " + myColor.toString();
     return parentDesc + myDesc ;
}
```

A program containing a **`ColoredBall`** called “**`myBall`**” could then display it as follows:

```
System.out.println ("myBall = " + myBall.toString());
```

or simply:

```
System.out.println ("myBall = " + myBall);
```

JC:jc
2/15/12