CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2014
John Clevenger

Assignment #1: Class Associations

Due Date: Thursday, March 6th [2 weeks]

Introduction

This semester we will study object-oriented graphics programming by developing a program which is a simplified version of a classic Arcade game named *Xonix* (also called Aironix or AirXonix), which itself is a variation of several earlier games such as *JezzBall*. In this game you will be controlling a "car" moving over a rectangular playing field, trying to avoid collisions with other objects such as "monster balls" rolling around on the field.

The rectangular field is made up of a grid of adjacent squares. Successfully driving your car from one edge of the field to any other edge allows you to claim ownership of all the field squares over which you traveled. In addition, if the path taken divides the field such that there is an enclosed area containing no monster balls, you also gain ownership of all the squares in that enclosed area. There are also various other objects on the field, some of which can move while others are fixed, and some of which can hurt you while others can help you. The object of each game level is to gain ownership of a certain percentage of the field; the required percentage increases at each level (as do the number of monster balls and other bad things).

If you have never seen the Xonix game, there is a nice 3D version you can play online at http://www.silvergames.com/xonix-3d to see the basic idea of the game (note however that this is a fancy 3D version; we'll be restricting ourselves to working in 2D, at least to start with). In any case, it is not necessary to be familiar with the original game (or the one on the web) to do any of the assignments during the semester (although looking at the above site might give you a better idea of what we will be working toward this semester).

The goal of this first assignment is to develop a good initial class hierarchy and control structure, designing the program in UML and then implementing it in Java. This version will use 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, and we will add graphics, animation, and sound. 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 such as **`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 contains a collection of two kinds of game objects: “fixed objects” (which are fixed in place) and “moveable objects” (which can move or be moved about the world). For this first version of the game there are two kinds of fixed objects: *field squares* and *time tickets*, and there are two kinds of moveable objects: *cars* and *monster balls*. Later we will see that there are other kinds of game objects (both fixed and moveable) as well.

The various game objects have attributes (fields) and behaviors (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 non-negative 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. All game objects provide the ability to obtain their location. Some kinds of game objects have changeable positions (that is, they are moveable), while others have positions that are not allowed to change.

- All game objects have a *color*, defined by a value of Java type *java.awt.Color*. All game objects provide the ability to obtain their color. Some objects provide the ability to have their color changed, while others have a color which cannot be changed once the object is created.

- Moveable game objects have integer attributes *heading* and *speed*. Telling a moveable object to *move()* causes the object to update its location based on its current heading and speed. *Heading* is specified by a compass angle in degrees: 0 means heading north (upwards on the screen), 90 means heading east (rightward on the screen), etc.

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

- *MonsterBalls* are concrete movable game objects with changeable color. They have a fixed integer attribute *radius.* They start with a random location, heading, speed, and color. Their movement is constrained such that they always lie within the rectangular field.

- *Cars* are concrete steerable objects with unchangeable color. Cars have integer attributes *width* and *height*. There is a rule that says there can only be ONE such object in the game (we'll see later how to enforce this rule in software). Cars are constrained such that the only headings they are allowed to take are 0, 90, 180, or 270 (that is, north, east, south, or west). The width and height of a car must be no greater than the *size* attribute of a *FieldSquare* (that is, cars must fit onto FieldSquares). However, cars may move anywhere, including off the field entirely.

- *FieldSquares* are concrete fixed (non-movable) game objects with changeable color. Field squares have a fixed integer attribute *size* which specifies the dimensions of a square. All field squares have exactly the same size.

- *TimeTickets* are concrete fixed game objects with unchangeable color and with fixed integer attributes *width* and *height*. TimeTickets also have an integer attribute *time* that represents the amount of time (in seconds) the player gains when the car is driven over (into) the TimeTicket.

The preceding paragraphs imply several *associations* between classes: an <u>inheritance</u> hierarchy, <u>interfaces</u> such as for *steerable* objects, and <u>aggregation</u> associations between objects and where they are held. You are to 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 field contains a grid of 100 x 100 squares. Each square is 5 x 5 units. At the start of each level in the game, the squares around the outside edge are all owned by the player; the rest of the squares are not owned. Owned squares are a different color from those that are not owned (you may choose the colors).

The car starts in the center of the square immediately to the left of the middle of the bottom of the field (that is, if the squares along the bottom are numbered 1..100 left to right, the car starts on square 50). The "location" of the car is considered to be the location of the exact center of the car; therefore, the initial location of the car is (247.5, 2.5). (You should do the math to make sure you understand the derivation of this initial location.) The car's initial heading is 0 degrees (north).

When the game starts the player has three "lives" (chances to complete the current level). The game has a *clock* which counts down; when the clock reaches zero the player loses a life. The player also loses a life if the car runs into a monster ball (or vice versa). When the player loses all three lives without completing a level, the game ends.

The objective of each level is to achieve a certain minimum score associated with that level. The score is computed as the percentage of field squares owned by the player (for example, if the player currently owns 2500 squares then the current score is 25%, since there are 10,000 squares on the field). The player retains all owned squares over the three lifetimes of a given level, but starts over again (owning just the outside squares) at the start of a new level. For Level One the minimum score to successfully complete the level is 50%; for each subsequent level the minimum score increases by 10%.

If the car runs into a Time Ticket, the *time* value of the Time Ticket is added to the current clock value (meaning the clock *increases*, giving the player more time to complete the level). For Level One the initial value of the clock is 10 and the value of Time Tickets is 5; for subsequent levels the starting value of the clock goes down by 2 for each level and the value of Time Tickets goes down by 1 for each level. (Note: these numbers will change in future assignments; write your code with that in mind).

The game keeps track of five "game state" values: current level, current clock time, lives remaining for the level, current score (percentage of squares owned by the player), and minimum score required to complete the level.

Commands

Once the game world has been created and initialized, the Game constructor is to call a method name **`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 Appendix on "Java Coding Notes", and also page 23 in the text).

The allowable input commands and their meanings are defined below. Each command returned by **`getCommand()`** should invoke a corresponding function in the game world, as follows (note that commands are case sensitive):

- 'n', 's', 'e', 'w': tell the game world that the car has changed heading and is now pointed either **n**orth, **s**outh, **e**ast, or **w**est respectively.
- 'i': **i**ncrease the car's speed by one unit.
- 'l' (the letter ell): **l**ower the car's speed by one unit as long as doing so doesn't reduce speed below zero.
- 'b': add a new Monster **b**all to the world at a random location and with random color, heading and speed (the speed should be constrained to be a small number).
- 'k': add a new Time Tic**k**et to the game world at a random location and with a time value corresponding to the current level as defined above.

- ‘g’: tell the game world that the car surrounded a **g**roup of squares and now owns them all. The effect of this command is to generate a random number (between 1 and the maximum number of non-owned squares) representing the number of new owned squares, display the number on the console, and add that number to the count of owned squares. Then if the new total of owned squares produces a score equal to or higher than the minimum score defined for the level, the player advances to the next level, which in turn causes the world to be reinitialized as described under “Game Play”.

- ‘1’: tell the game world that the car has collided with a monster ball. Colliding with a monster ball has the effect of costing the player one life and repositioning the car back at the initial starting location. It also causes the monster ball’s color to change.

- ‘2’: tell the game world that the car has driven over a new square and now owns another square (in other words, increase the number of owned squares by one). Later we will have to keep track of precisely *which* squares are owned; for now we won’t worry about that. As with the ‘g’ command, if the new total of owned squares produces a score equal to or higher than the minimum score defined for the level, the player advances to the next level, which in turn causes the world to be reinitialized as described under “Game Play”.

- ‘3’: tell the game world that the car has driven over (collided with) a Time Ticket. The effect of this is to increase the countdown clock by the amount of time contained in the Time Ticket and to remove the Time Ticket from the world.

- ‘t’: tell the game world that the “game clock” has **t**icked. 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 time “game clock” is decremented by one (the game clock for this assignment is simply a variable which decrements with each tick). If the clock has now reached zero, the player loses a life and is relocated to the start position. If the player has no lives left, the game ends.

- ‘d’: generate a **d**isplay by outputting lines of text on the console describing the current game state values. The display should include (1) the current level, (2) the number of lives left in the level, (3) the current countdown clock value (remaining time), (4) the current score (the percent of squares owned), and (5) the minimum required score for the level. All output should be appropriately labeled in easily readable format.

- ‘m’: tell the game world to output a “**m**ap” showing the current world (see below).

- ‘q’: call the method `System.exit(0)` to **q**uit the program. Your program should confirm the user’s intent to quit before actually exiting.

Additional Details

- The code to perform each command must be encapsulated in a separate method (this is because we will be moving it in later assignments). When the *Game* gets a command which requires manipulating the *GameWorld*, the Game must invoke a method in the GameWorld to perform the manipulation (in other words, it is not appropriate for the Game class to be directly manipulating objects in the GameWorld; it must do so by calling an appropriate GameWorld method).

- Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. For example, input lines with more than a single character are illegal, as are commands to perform impossible operations (such as "collide with a Time Ticket" when there are no Time Tickets in the world).

- All classes must be designed and implemented following the guidelines discussed in class, including:
  - o *All data fields must be private.*
  - o *Accessors / mutators must be provided, but only where the design requires them.*

- Moving objects need to determine their new location when their *move()* method is invoked, at each time tick. The new location can be computed as follows:

  `newLocation(x,y) = oldLocation(x,y) + (deltaX, deltaY),` where
  `deltaX = cos(θ)*speed,`
  `deltaY = sin(θ)*speed,`

  and where **θ** = 90-heading. There is a diagram in course notes (p. 232) showing the derivation of these calculations for an arbitrary amount of time; in this assignment we are assuming "time" is fixed at one unit per "tick", so "elapsed time" is 1.

- For this assignment <u>all</u> 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, similar to the following:

```
Car: loc=120,110 color=[255,0,0] speed=10 heading=90 width=5 height=10
Ball: loc=130,110 color=[0,255,0] speed=5 heading=23 radius = 8
Ball: loc=70,250 color=[0,0,255] speed=2 heading=235 radius = 4
Ball: loc=20,300 color=[0,0,0] speed=10 heading=0 radius = 10
TimeTicket: loc=50,50 color=[64,64,64] width=10 height=100 time=30
```

  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 attached "Java Coding Notes" for additional details.

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

- It is a requirement to follow standard Java coding conventions:
  - o *class names always start with an <u>upper case</u> letter,*
  - o *variable names always start with a <u>lower case</u> letter,*
  - o *compound parts of compound names are capitalized (e.g., `myExampleVariable`),*
  - o *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 a "package" statement (for example, "**`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. See pages 16-18 and 29-31 in the text for more on the use of Java packages.

- Use of subpackages below "package a1" is encouraged. For example, you might create a package "a1.gameObjects" holding the classes comprising the GameObject hierarchy.

- You are not required to use any particular data structure to store the game world objects. However, your program must be able to handle changeable numbers of objects at runtime; this means 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 Movable) {
          Movable mObj = (Movable)theWorldVector.elementAt(i);
          mObj.move();
     }
}
```

- Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged.

- 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 ***SacCT***. Submit a single "ZIP" file containing: (1) your UML diagram in .PDF format, (2) the *Java* source code for all the classes in your program, and (3) the compiled (".class") files for your program. 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.

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
2/19/14