CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2015 / Dr. Muyan

Assignment #1: Class Associations

Due Date: Thursday, Oct 1st

Introduction

We will study object-oriented graphics programming and design by developing a simple video game we'll call *Dog Catcher.* In this game, the human player controls the size and location of a net, and accumulates points by using the net to catch as many dogs as possible. The game also includes cats that multiply quickly, and that often get into fights with the dogs.

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 by designing the program in UML, and then implementing 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, including: (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 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.

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 a *GameWorld*, calls a *GameWorld* method **`initLayout()`** to set the starting layout of the game, and then starts the game by calling a method **`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. It is important for things that we will do later that your program follows this organization:

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

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

```
class Game {
   private GameWorld gw;

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

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

Game World Objects

The "Dog Catcher" game world is a 1024x1024 area and it contains a collection which aggregates objects of abstract type **GameObject**. There are two kinds of abstract game objects: "animals" and "catchers". For this first version of the game there are two concrete kinds of animals: *dogs* and *cats*, and there is one concrete kind of catcher: *nets*. Later we may add other kinds of animals and catchers. 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 in the range 0.0 to 1024.0. The point (X,Y) is the center of the object. The origin of the "world" (location (0,0)) is the lower left hand corner. All game objects provide the ability for external code to obtain and change their location.

- All game objects have a *color*, defined by a value of Java type *java.awt.Color*. All objects of the same class have the same initial color (chosen by you), assigned when the object is created (e.g, cats could be yellow, dogs could be brown). All game objects provide the ability for external code to obtain their color. By default, game objects provide the ability to have their color changed, unless it is explicitly stated that a certain type of game object has a color which cannot be changed once it is created.

- Some game objects are *moving*, meaning that they provide an interface that allows other objects to tell them to move. Telling a moving object to *move()* causes the object to update its location; the way the location is updated depends on the type of object (that is, not all moving objects necessarily *move()* in the same way). All animals are moving.

- Some game objects are *guided*, meaning that they provide an interface that allows other objects to tell them to move left, right, up, or down. All catchers are guided.

- All game objects have an integer attribute *size.* By default, game objects provide the ability to have their size changed, unless it is explicitly stated that a certain type of game object has a size which cannot be changed once it is created.

- All animals have integer attributes *speed* and *direction,* which are used to define how they move through the world when told to do so. In addition, all animals have a *size* which cannot be changed once the object is created.

- Each dog is represented as a circle. The size attribute of the dog indicates the diameter of the circle. Each dog's size is chosen randomly when created, and constrained to a reasonable value. The location of each dog is also randomly set, but must be chosen such that the dog is completely contained in the world. Dogs also have an integer attribute *scratches*, indicating how many times it was scratched by a cat (up to 5). The color of a dog may change during the game. Several dogs are instantiated at the start of play, and no new dogs are created afterwards.

- Each cat is represented using an equilateral triangle. The size attribute of the cat indicates the length of equal sides of the triangle. Each cat's size is also chosen randomly when created, and constrained to a reasonable value. Like a dog, the location of each cat is randomly chosen when it is created, but must be chosen such that the cat is completely contained in the world. The color of a cat never changes. Several cats are instantiated at the start of the game, and additional cats may be created during the game, as described later.

- The *speed* of dogs and cats is in the range 0-5, and starts at 5. The *direction* of dogs and cats indicates heading specified by a *compass angle* in degrees: 0 means heading north (upwards on the screen), 90 means heading east (rightward on the screen), etc. *Direction* is initialized to a random value (ranging between 0 and 359) at the time of instantiation. See below for details on updating an animal's position when its *move()* method is invoked.

- Each net is represented using a square. The size attribute of the net indicates the length of equal sides of the square. This attribute is constrained to be a positive integer between 50 and 500 (inclusive), and set to 100 when the object is created. Like a dog and a cat, the location of a net is randomly chosen when it is created, but must be chosen such that the net is completely contained in the world. The color of a net never changes.

The preceding paragraphs imply several *associations* between classes: an inheritance hierarchy, interfaces such as for *guided* and *moving* objects, and aggregation associations between classes. 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. An additional important criterion is that *another programmer must not be able to misuse your classes*, e.g. if the object is specified to have a fix color, another programmer should not be able to change its color after it is instantiated.

Game Play

When the game begins, the player specifies an initial number of dogs and cats to be created, then tells the game to start. Your program then populates the 1024x1024 playing field (game world) with that number of cats and dogs, and one net, placed in random positions. Immediately the cats and dogs start running. The human can then begin operating the net.

The net can be operated in three ways: (1) guiding the net, either up, down, left, or right, (2) making the net bigger or smaller, and (3) scooping up all of the animals that happen to currently be in the net. If there are animals in the net, but the player doesn't tell the net to "scoop", they continue to run and are not caught. Catching an unscratched dog is worth 10 points, but catching a cat deducts 10 points.

The animals all run simultaneously according to their individual speed and direction. They also add (or subtract) small random values to their direction so as to not run in a straight line. If an animal hits a side of the world, it changes direction and does not run out of bounds.

If two cats run into each other, one of them has a kitten, and a new cat object is created in a nearby position. If two dogs run into each other, nothing happens.

If a cat and a dog run into each other, the cat scratches the dog. This causes the dog's scratch attribute to be incremented, and it's color to change (such as becoming more red). It also causes the dog's speed to reduce by 1. The cat's speed would not change. A dog with 5 scratches stops moving. When catching a dog, the score is reduced depending on its number of scratches. For instance, if the dog has been scratched 3 times, then catching it is worth 7 points rather than 10 points.

Note that although there is no penalty for acting slowly, over time the world gets more and more overrun with cats, and the dogs get more and more scratched up, making it harder to catch the dogs and reducing their point value. So strategically a player should act quickly.

The game keeps track of five "point" values: Dogs captured, Cats captured, Dogs remaining, Cats remaining, and Total points (which may be negative). Play stops when all of the dogs are caught. The objective is to obtain as high a score as possible. The maximum possible score is the initial number of dogs times 10, and there is no lowest possible score.

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").

Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. It is *not* a requirement in this assignment that single keystrokes invoke action -- the human hits "enter" after typing each key command (we'll see how to fix this in a later assignment). The allowable input commands and their meanings are defined below (note that commands are case sensitive):

- 'e' – **e**xpand the size of the net (the center location of the net shouldn't change).
- 'c' – **c**ontract (decrease) the size of the net (the center location shouldn't change).
- 's' – **s**coop up all the animals that are in the net. This causes all of the animal objects whose positions are within the boundaries of the net to be removed from the game world, and the score to be updated according to the rules of play described above.
- 'r' – move the net to the **r**ight.
- 'l' – move the net to the **l**eft.
- 'u' – move the net **u**p.

- 'd' – move the net **d**own.

- 'k' – **pretend** that a collision occurred between two cats. Later this semester, your program will determine this automatically [1]. But for now, if the player specifies the 'k' command, the program picks a cat and tells it to produce a **k**itten. If there are no cats, print an error message without producing a kitten.

- 'f' – **pretend** that a **f**ight occurred between a cat and a dog (a cat and a dog run into each other) [1]. The program picks a dog and scratches it once, changes its color, and reduces its speed by 1. If there are no cats, print an error message instead.

- 't' – tell the game world that the "game clock" has **t**icked. All objects are told to update their positions according to their current direction and speed.

- 'p' – print the **p**oints: (1) current score, (2) number of dogs/cats captured, and (3) number of dogs/cats left in the world. Output should be appropriately labeled in easily readable 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 confirm the user's intent to quit before actually exiting.

The code to perform each command must be encapsulated in a separate method (this is because we will be moving those methods to other locations 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).

Additional Details

- Method ***initLayout()*** is responsible for creating the initial state of the world. This should include adding to the game world at least the following: The player specified number of dogs and cats, and one net, placed in random positions. All object initial attributes, including those whose values is not otherwise explicitly specified above, should be assigned such that all aspects of the gameplay can be easily tested.

- 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,*

[1] In later assignments we will see how to actually detect on-screen collisions such as this; for now we are simply relying on the user to tell the program via a command when collisions have occurred. Inputting a collision command is deemed to be a statement that the collision occurred; it does not matter where objects involved in the collision actually happen to be for this version of the game as long as they exist in the game.

- Moving objects need to determine their new location when their *move()* method is invoked, at each time tick. As derived in the lecture notes (in "Introduction to Animation" chapter, "Computed Animation Location" slides), 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 - direction (90 minus the direction). The course notes show the 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:

```
Net: loc=130.0,565.5 color=[0,0,0] size=60
Dog: loc=690.0,830.2 color=[0,20,128] size=8 speed=4 dir=43 scratches=1
Dog: loc=190.1,60.1 color=[0,0,128] size=10 speed=5 dir=240 scratches=0
Cat: loc=1180.3,0.0 color=[0,128,15] size=7 speed=5 dir=112
Cat: loc=50.9,540.2 color=[0,128,15] size=12 speed=5 dir=313
Cat: loc=440.2,0.0 color=[0,128,15] size=6 speed=5 dir=3
```

  This example output assumes that there are currently two dogs and three cats. One of the dogs has been scratched once and is therefore moving at a slower speed. Note also 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 appendix on "Java Coding Notes").

- 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., IGuided).*

- 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 "Deliverables", below).

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

- You are not required to use any particular data structure to store the game world objects. But 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 "moving" object with code like:

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

- You can utilize `java.util.Random` class (see the appendix on "Java Coding Notes") to generate random values specified in the requirements (e.g. to generate initial random sizes and locations of animals).

- It is a requirement for *all* programs in this class that the source code contain *documentation*, in the form of comments explaining what the program is doing, including comments describing the purpose and organization of each class and comments outlining each of the main steps in the code. Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged.

Deliverables

There are *three steps* which are required for submitting your program, as follows:

1. Create a *single* file in "ZIP" format containing (1) your UML diagram in .PDF format, (2) the *Java source code (".java")* for all the classes in your program, and (3) the compiled (".class") files for your program. Be sure your ZIP file contains the proper subpackage hierarchy. Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications.

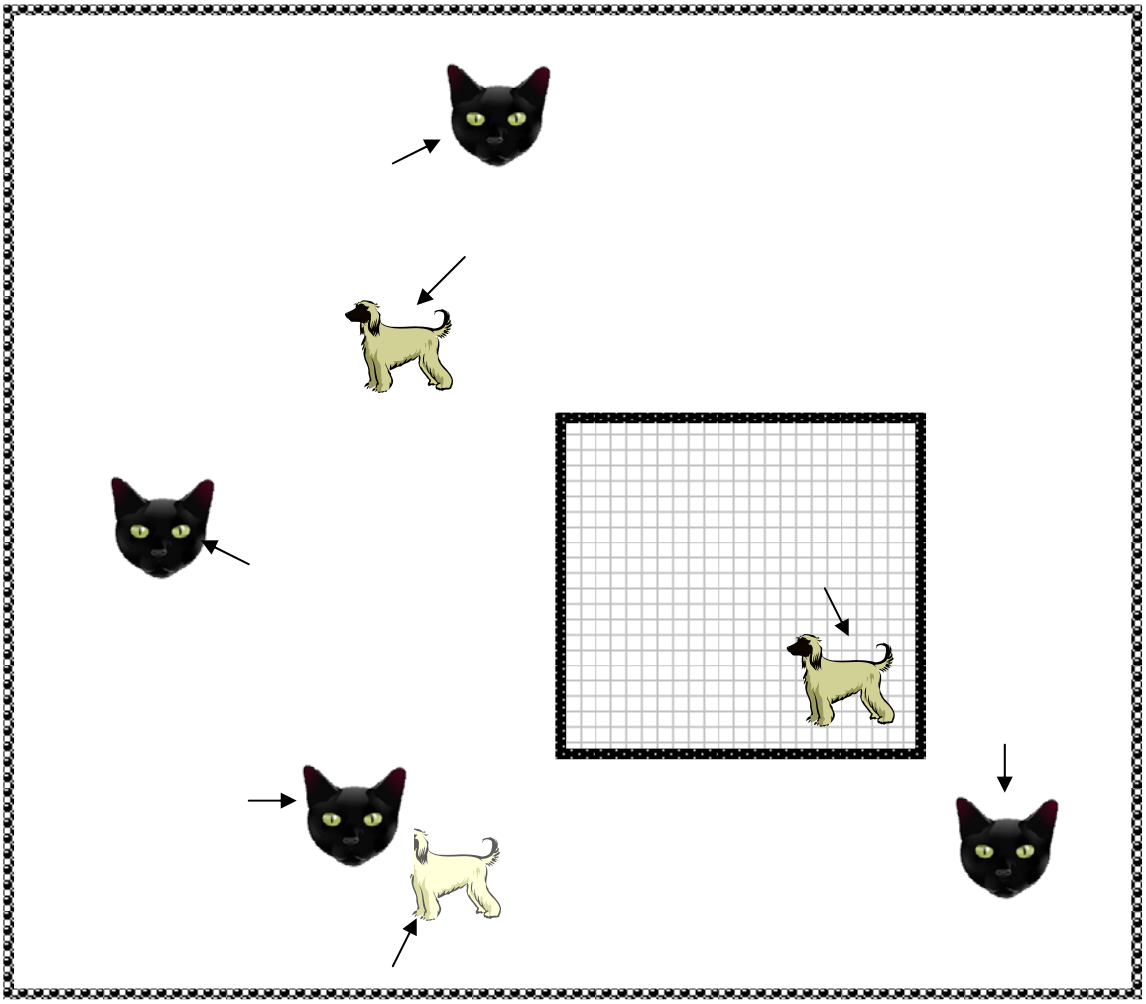
2. *Verify your submission file.* To do this, copy your ZIP file to an empty folder on your machine, unzip it, open a command prompt window, change to the folder where you unzipped the file, and type the command "**java a1.Starter"**. If this does not properly execute your program, your ZIP file is not properly structured; go back to step (1) and fix it. Substantial penalties will be applied to submissions which have not been properly verified according to the above steps.

3. Login to ***SacCT***, select "Assignment 1", and upload your *verified* 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 file).

All submitted work must be strictly your own!

Example:

The below illustration is provided to help clarify some of the elements of DogCatcher. Your program is *not* required to produce any graphical output like this (later we will learn how to draw a simple graphical depiction of the world). For this assignment, *the only required depiction of the world is the text output map as shown above*.



In this example:

- *There are three dogs and four cats,*
- *The “Net” has one dog in it,*
- *The lighter-colored dog near the bottom of the game area has been scratched by the nearby cat, so its color has changed,*
- *All of the dogs and cats are wandering around the game area,*
- *The net can be moved, enlarged, or shrunk by the player at any time during play,*
- *The human player right now could earn 10 points by “scooping” the dog in the net.*

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 several methods including **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 **nextFloat(int)**, which returns float value (between 0.0 and 1.0), and **nextBoolean()**, which returns a random boolean value (either true or false).

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 **Book**, and a subclass of **Book** named **ColoredBook** with attribute *myColor* of type java.awt.Color. An appropriate *toSring()* method in **ColoredBook** might return a description of a colored book as follows:

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

A program containing a **ColoredBook** called "**myBook**" could then display it as follows:

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

or simply:

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