CS245: Program #3:  
Breakout  
Due: Thurs, April 21

Breakout

You will be creating a basic version of the game of breakout. The object of the game is to bounce the balls (blue) off the moving paddle (green) controlled by mouse movement so that they bounce off the walls (white) and bricks (red) without touching a killing wall (gray). Each time a brick is hit, that brick will disappear. The game is won when all the bricks are gone.

The first thing to know is that we will be designing the breakout game to be flexible. That is, the locations and sizes of all the components will (for the most part) not be hard-coded into the application. This will allow us to design different levels easily and even store those levels in external files to be read in.
Another issue we will have to deal with is the physics of a bouncing ball. We want the ball to bounce off the walls and paddle in a realistic manner.

The following sections outline the steps you should follow to complete this project. This project is designed to be done in the order listed below. You are free to do it in any order you wish, but I think you get more intermediate checkpoints in the order listed.

**GUI**

The first step should be to build the basic GUI framework. This will include a class called `Breakout` which is in charge of setting up the JFrame and the components that go onto the frame. The main will be in this class as well. Your JFrame should have a label at the bottom to present messages to the user (it says “Breakout” in the sample above, but mine can also say “Win” or “Loss”.

The JFrame will also have a large area in the middle which does all the painting of the breakout components. This implies we will also have to have a JPanel that overrides the painting method. Call this class `BreakoutPanel`. Note that because we want the game to be a flexible as possible this breakoutpanel should simply display BreakoutComponents (next section).

You will be able to test your basic GUI at this point, but not much will show up on the screen until you complete the next section.

**BreakoutComponent**

Everything that gets drawn on the screen is a `BreakoutComponent`. You should make this an abstract class as you won’t want to create new ones, but you will need to add some shared code into the class. In particular, it should have a constructor with the following signature:

```java
public BreakoutComponent(int x, int y, int w, int h, Color c)
```

This allows the most flexibility in terms of letting sub-classes pick the upper-left (x, y) corner and size (w, h) of the rectangle along with its color. You will need to store this information in this abstract class as well. I highly suggest storing the shape information in a Rectangle class (not your own, but the built-in Java one). It is a good match to our shapes which are in fact just rectangles.

All BreakoutComponents can be drawn the exact same way as well since they are all just colored rectangles. Thus, you should provide a draw method in this class as well that takes in the Graphics context with which to do the drawing.

There may be other methods that need to go in this class as well, but we will discuss them later when needed.
Concrete BreakoutComponents

Now you should build the various concrete sub-classes of BreakoutComponent. These are: **Ball**, **Paddle**, **Brick**, **Wall**, and **KillingWall**. Each of these classes should take its location and size from an external source, but have a fixed color for the type of component it is.

Once you have these basic breakout components coded you should be able to create a few walls, bricks, paddle and ball and see them display properly on the GUI.

Ball Movement

Of course a static game isn’t that exciting. The first thing we should do is get the ball(s) moving. A couple of points before we code any movement into Ball however.

First is we will need a “game loop” that runs the game. This program isn’t using any explicit threads (there are some in the GUIs, but none we create by ourselves). Thus we will need an explicit game loop that runs until the game is over. The best place to put this game loop is in a method called run() in the Breakout class. Thus the main should simply create a new Breakout and tell it to run. To start with you can simply loop forever – you will have to identify wins and losses later on to quit the loop.

The game loop consists of the following steps: update the position of all moving objects, check for collision between all moving objects and other objects, draw all the objects on the screen, and finally wait a few milliseconds so that the loop doesn’t run too fast!

The drawing of all objects is the easiest to start with. In theory we just need to loop through all the BreakoutObjects in the system and have them draw themselves on the screen. However, we don’t have direct access to the Graphics context so we will need to ask the BreakoutPanel to do this for us.

The other two part of the game loop are operations that run on objects that move. For us, this includes Balls and Paddles. However, it could in theory include moving Walls or Bricks as well. This isn’t required for this assignment, but if someone wants to create a MovingBrick class I would be interested to see it. The point here is that not all BreakoutComponents are ones that move. Thus, we don’t really want to put methods to handle the updating of positions and collision detection into all BreakoutComponents. Thus, we will instead create a new Interface called **MovingComponent**. To start with this interface should include the following signature:

```java
public abstract void updatePosition()
```

This is the method that will be called to update the position of the moving objects each time through the game loop. There will be additional methods that belong in this interface as well.

Finally it is time to add movement to the Ball class. This simply means we need to make Ball implement MovingComponent and implement the updatePosition method. It will also be useful
to keep a List of just the MovingComponents in the GUI so your game loop can easily loop through them.

So how does a ball move? Each ball is going to need a direction it is moving. We will represent this as a unit vector in the direction of movement. A unit vector is simply a vector with a length of 1. Each time updatePosition is called the ball should change its location by some multiple of this direction. Once you have this complete you should be able to run your program and see the ball move from its starting position to off the screen.

**Paddle Movement**

Next we will want to get a moving paddle. This of course means that Paddle should also implement the MovingComponent interface. Things are a bit different than in Ball however since the movement of the Paddle is controlled by the user moving the mouse. I feel the best way to attack this issue is to make the Paddle also be a MouseMotionListener. Then we can setup the Paddle to listen for all mouse motion on the BreakoutPanel.

There is still one issue to resolve though. Whenever the user moves their mouse it will generate an event that the Paddle will know about. However, we don’t want to update our Paddle position and repaint the screen at this instance in time. We need to wait until the game loop tells us it is time to update our position. Thus, we can simply keep track of where the user wanted us to go and then during the updatePosition phase of the game loop we can use that information to slide the Paddle left or right (ignoring the y position of the mouse).

Note that you will probably want to adjust this movement so that the mouse feels like it is in the center of the Paddle rather than on the left side as this will be more natural for the user of the game.

**Collision Detection and Bouncing**

A game where the Paddle moves but never hit the ball which goes flying off the screen is about as much fun as the static version. Thus, we need to add collision detection as well as bouncing to our system. We will need to add another method to our MovingComponent interface called collisionCheck. This will have to be called after all the object has had their position’s updated and before they are drawn on the screen. Each moving component will have to check to see if it has collided with every other object in the system (moving or not). Thus, the collisionCheck method will have to be passed the list of BreakoutComponents.

The collisionCheck should do a collision detection check with each object in the list, one at a time. At times you will find that your object collides with multiple other objects (or parts of those objects). You only need to handle collision with one – simply pick one if multiple collisions happen.

The beauty of using a Java Rectangle to hold the position/shape information for each component is that Rectangle has several intersects methods already implemented in it. There is one that tests
intersection with a Rectangle and other Rectangle. While this seems to be the perfect match, it will turn out that it only really is useful for the Paddle collisions. That is because for Paddle collisions, one just needs to determine if the Paddle collided with an object and if it did it simply just needs to freeze in place. That is, the Paddle doesn’t bounce off objects it collides with like a ball would.

In order to “bounce off” the object you collide with, you need to determine which way to bounce. In particular this is the reflection direction w.r.t. the normal of the surface you hit. Since all our objects are oriented along either the horizontal or vertical axis, the reflection is either reversing the x or y coordinate. However, the important thing to note here is that in order to compute this reflection direction, one needs to know which side of the rectangle is hit. Thus, a full rectangle/rectangle collision check will determine if we intersected but it will not tell us which side we hit. In order to determine which side we hit, we will need to do a collision check between us (a rectangle) and each side of the other rectangle in question, one at a time. Thus, the other Rectangle intersects method that checks intersections with lines will be more useful for Ball collisions.

**Game Play**

Finally we need to deal with game play issues. That is, we need to make bricks disappear when hit and have the ball die when it hits a killing wall. We will also need to identify when all the bricks have been hit or when a ball is killed in the main game loop in order to quit.

The easiest approach to keeping track of when something has been hit is to add new methods to BreakoutComponent (since all components can be hit). Thus, if during the collision detection phase it is determined that a component is hit, it should tell that component it has been hit. That component should then store that information so it can later tell others if it has been hit or not. Actually, the functionality of how the hit information is saved and used depends on the specific type of BreakoutComponent. Bricks care about being hit, Walls could care less!

Additionally, certain components such a KillingWall need to actually kill the MovingObject that just collided with it. Thus, when they are informed that they are hit, they should let the object that hit them know they are killed. Note that you should **not** (in your collision detection code), use instanceof to determine if the object you collided with is a KillingWall. This is because then you have hard-coded KillingWalls as the only component that can kill a ball. If later on you wish to add another type of KillingWall (such as a MovingKillingWall) then you would need to modify this code as well. You should instead do what I suggested above. Tell the object you collided with that it has been hit and then if that is a certain type of object (such as a KillingWall) it will turn around and tell the object that collided with it (i.e. the Ball) that it has been killed. This implies that you will need other methods in MovingComponent as well to have components such as KillingWalls call back on the object that just hit it.
Level Loading

One of the features of our flexible system is that we should be able to simply design levels externally to the program itself and load them in on runtime. I have provided you with my external file that was used to create the level seen at the top of this document. You should rip out any hard-coded level information you had in your GUI and replace it by loading in a file such as this and creating your BreakoutComponents from it.

Note that although there can be any number of Walls, KillingWalls, Bricks, and Balls, there should only be 1 Paddle in the system. This is because we can only control one paddle at a time with the mouse. If we had a networked breakout game where each player could see the same screen and had their own mouse then we could have multiple Paddles.

Lastly, I want you to create your own level and include it with the submission.

Submission

Submission will be similar to the last assignment. Jar file in the W: drive, printout of your source code, cover sheet, and screen shots of your GUI.

This assignment may be done with partners.