Project #8 **CMSC 131**
Due: Monday 12/11 at 11:00 PM **Object-Oriented Programming I**
Type of Project: Open **Fall 2006**

Shape Decorator

(The world's slowest drawing tool)

Objective

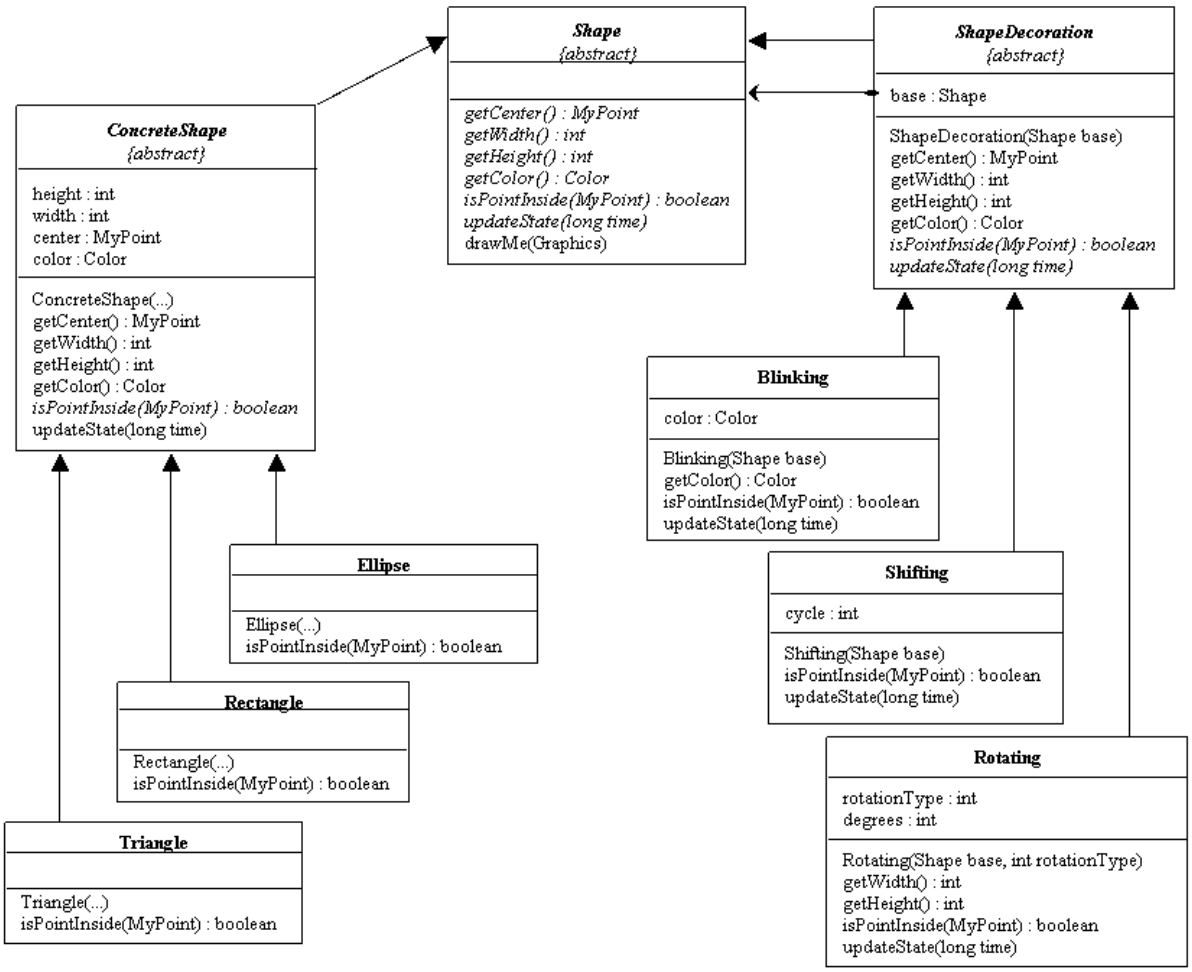
To practice inheritance, polymorphism, abstract classes and abstract methods. You'll also get to practice using formulas from your pre-calculus class as tools for drawing geometrical shapes. Finally, this project will expose you to the "Decorator" design pattern.

Click here to see a video of the project working.

Overview

Note that this project is "Open". Please remember that this policy allows more leeway in discussing the implementation of the project, but you are still required to do your own work.

For this project you will write a "shape library", which will be used by a GUI that we are providing. Below is a diagram of most of the classes you will write. This type of diagram is called a UML class diagram. (UML stands for Unified Modeling Language.) UML diagrams are used by the pros, and you'll learn to draw some of them yourself if you take CMSC 132!



How to Read the UML Class Diagram

It is common with this sort of diagram to omit details occasionally, and to show only those features that are important for the intended audience. For example, I have left out one of the classes (MyPoint) because it is not part of the inheritance hierarchy, and I also left out the parameter lists for some of the constructors, because they are not very interesting or enlightening.

Each box in the diagram represents a class, and is divided into three regions:

- The name of the class
- The "state" of the class
- The "behaviors" of the class

Note that the **types** of fields (and method return types) come *after* the name. For example, *"degrees : int"* represents an int variable called "degrees", and *"getWidth() : int"* represents a method named "getWidth" that will return an int. Strangely, the order of the type and variable name are the other way around (Java style) for parameters of methods, as in *"long time"*.

Things in *italics* are "abstract", including classes and methods.

Arrows like the one below indicate inheritance ("Is-A").

Arrows like the one below indicate composition ("Is-Made-From-A").

There are other types of arrows as well, but not in today's example.

Decorator Design Pattern

The relationship among the classes in your shape library is an illustration of a common design pattern called the "Decorator" pattern. The idea is that we have several "concrete" shapes that can be instantiated any time: Ellipse, Rectangle, and Triangle. Typical syntax might be:

```
Shape x = new Rectangle(...);   // a plain (undecorated) rectangle
```

Anytime you have a "Shape" in hand, you can wrap it inside one of the classes that are "Decorators": Blinking, Shifting, and Rotating. Each of these Decorator classes has a constructor that accepts a Shape parameter -- the Decorator "wraps" the Shape. The interesting part is that each of the Decorators *Is-A* Shape as well, so once you have wrapped a Shape inside a Decorator, you can then wrap that Decorator inside another Decorator, etc. Typical syntax might be:

```
Shape x = new Shifting(new Rotating(new Triangle(...)));    // a shifting, rotating triangle
```

Specifications

Below are detailed descriptions of the classes and methods you must write. Note that there is one class that you will write that was not included in the UML class diagram (it's the MyPoint class).

All of the classes you write must go into a package called "shapeLibrary".

1. MyPoint

This class represents an immutable "point" on the GUI. We'll use it to specify the locations of our shapes.

State:

- private int x
- private int y

Behaviors:

- public MyPoint(int x, int y) -- usual constructor
- public static MyPoint add(MyPoint p, MyPoint q) -- adds the corresponding x and y coordinates of the two parameters, and returns a point with x and y coordinates equal to the sums.
- public int getX() -- usual accessor
- public int getY() -- usual accessor

2. Shape

This class must be declared as abstract. It is the top level class in the inheritance diagram.

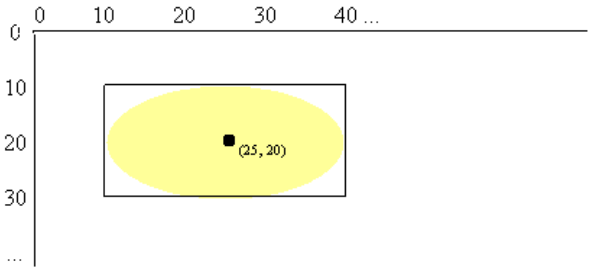
State: (none)

Abstract methods (i.e. declared, but not implemented in the Shape class):

- protected abstract MyPoint getCenter();
- protected abstract int getWidth();
- protected abstract int getHeight();
- protected abstract Color getColor(); [The class Color is defined in the package "java.awt".]
- protected abstract boolean isPointInside(MyPoint p); [Subclasses will implement this method to allow the user to check whether or not a point is actually located within the Shape. Each time a shape must be drawn, you will make many, many calls to this method -- one for each pixel in the rectangular area defined by the center, width and height of the shape. If this method returns true, you will color the pixel. This is not a very efficient way to draw a shape, but it allows for easy decorating!]
- public abstract void updateState(long time); [Subclasses will implement this method to create "animation". The way it works is this: The GUI has a Timer that pulses every 100th of a second. Each time the timer pulses, a listener is notified. The listener will then invoke the updateState method on all of the Shapes in the model, passing in a snapshot of the current time. *Side note: We don't allow the updateState method to determine the time itself, for reasons of synchronization*]

Behaviors:

- public void drawMe(Graphics g) -- The fully qualified name of the type for the parameter is java.awt.Graphics. This method will be called by the GUI to draw the current object. The parameter (g) represents the rectangular graphical area on the screen where all of our shapes will be drawn. This region consists of pixels with coordinates as shown in the diagram below. Note that the point (0, 0) is located at the upper left corner of the region, and the numbers along the vertical axis get larger as you go*down*! In the diagram below, we've drawn a Shape (an Ellipse) with a center point of (25,20), a width of 30, and a height of 20. **We will refer to the rectangle that surrounds the shape as the "Bounding Rectangle".**



Below are a couple of methods of the Graphics class that you will need to use for drawing:

1. setColor(Color c) -- specify the color you want to use while you are drawing (i.e., the color of the Shape)
2. drawLine(x1, y1, x2, y2) -- draw a line from the point (x1, y1) to the point (x2, y2)

The drawMe method that you are writing will draw the current object (the Shape) by cycling through EVERY pixel in the bounding rectangle. You can compute the dimensions and position of this rectangle based on the Shape's center location, width, and height. For each pixel in the bounding rectangle, you will check whether or not the pixel is inside the Shape by calling the shape's isPointInside() method. This method will return true if the point is inside the shape, and false otherwise. If the pixel is inside the shape, then it should be colored. If the pixel is not inside the shape, then don't color it. (This technique is very slow, but makes for a great CMSC131 project!)

Unfortunately, Fawzi couldn't figure out how to color a single pixel (if you figure it out, please post to the wiki), so the way I colored each pixel was to draw a line from it to itself. In other words, to color the pixel at point p, I used:

```
g.drawLine(p.getX(), p.getY(), p.getX(), p.getY());
```

3. ConcreteShape

This class is abstract and extends the Shape class.

State:

- private int height;
- private int width;
- private MyPoint center;
- private Color color;

Abstract method (not implemented in this class):

- protected abstract boolean isPointInside(MyPoint p);

Behaviors:

- public ConcreteShape(int height, int width, MyPoint center, Color color) -- standard constructor. In case you haven't noticed, the fully qualified name of the type of the last parameter is java.awt.Color.

- public void updateState(long time) -- this method does nothing, but implement it here as an empty method! (The method will actually *do something* when we implement it for shapes that move. ConcreteShapes don't move in this project.)
- protected MyPoint getCenter() -- accessor
- protected int getWidth() -- accessor
- protected int getHeight() -- accessor
- protected Color getColor() -- accessor

4. Ellipse

This class extends ConcreteShape.

State: (none other than what is inherited)

Behaviors:

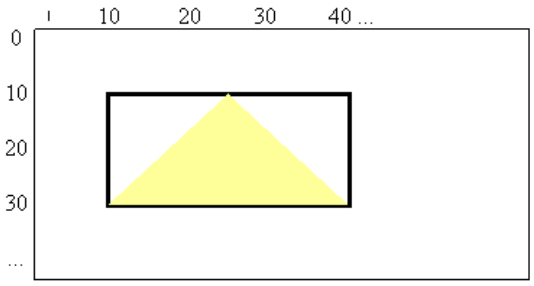
- public Ellipse(MyPoint center, int width, int height, Color color) -- standard constructor
- protected boolean isPointInside(MyPoint p) -- returns true if the point p is located inside the Ellipse, false otherwise. If you don't remember the formula for an ellipse, you're going to need to look it up! (I found it in 5 seconds using Google.)**Hint: your formula should be an inequality (not an equation) because you are interested in all of the points inside the ellipse, not just the points on the boundary.** Your formula must be based on the center, width, and height of the Ellipse.

5. Triangle

This class is similar to the one above, but for a Triangle shape. The shape of the Triangle will be as drawn below. The triangle in the diagram is centered at the point (25, 20), has a width of 30 and a height of 20. The triangle shape is an isosceles triangle, so the corner at the top is directly above the center. Note that the base of the triangle must be drawn on the bottom (don't accidentally draw the triangle upside-down!) When implementing the isPointInside method, you may want to consider the interior of the triangle as the solution to a set of three linear inequalities (one for each side of the triangle). The following formulas may be helpful:

Calculating the slope of a line passing through two given points:
$m = (y1 - y2) / (x1 - x2)$

The equation of a line with slope m, passing through the point (x1, y1). (You should turn these equations into inequalities.)
$y - y1 = m (x - x1)$



6. Rectangle

Similar to the previous two classes, but the shape is a rectangle. When implementing the isPointInside method, you have to check if the point is in the interior of the bounding rectangle. (Do not simply return "true" -- the method may be called for points that are way outside the bounding rectangle for the Shape -- this will happen in this project when the "rotation" decoration is being used.)

7. ShapeDecoration

This abstract class extends the Shape class. It is the base for all of the "Decorations". The idea is that each Decoration "wraps" a Shape, and at the same time each Decoration "Is-A" Shape. The subclasses of this class are the actual decorations. They may override some of the methods that are implemented by this class.

State:

- protected Shape base; // the Shape that is being wrapped

Abstract methods (not implemented in this class):

- protected abstract boolean isPointInside(MyPoint p);
- public abstract void updateState(long time); -- the GUI has a Timer that pulses every 100th of a second. Each time the timer pulses, a listener will invoke this method on all of the Shapes. This is how our animation is done.

Behaviors:

- public ShapeDecoration(Shape base) -- standard constructor.

The following methods must be implemented. They simply *delegate* to the underlying base Shape. (In other words, these methods simply call the corresponding methods of the base. The subclasses of this class will be overriding some of these.)

- protected MyPoint getCenter()
- protected int getWidth()
- protected int getHeight()
- protected Color getColor()

8. Blinking

This class extends the ShapeDecoration class. It wraps a Shape and makes it blink.

State (in addition to the inherited part):

- private Color color; [Flips back and forth between the shape's normal color and black (hence the blinking.)]

Behaviors:

- public Blinking(Shape base) -- standard constructor. Initialize the variable *color* to be equal to the color of the base Shape.
- protected boolean isPointInside(MyPoint p) -- delegate to base (have this method call the base's corresponding method.)
- public Color getColor() -- We are overriding this method so that the color will "blink". This method is just an accessor for the color field of the current object (not the base object). The color will be changed in the method below.
- public void updateState(long time) -- The parameter, time, will be a value measured in milliseconds that represents a snapshot of the system clock taken by the GUI when it began drawing this shape. This method will flip the color of the current object from black to the base color and back every second.

  The first thing this method must do is to call the updateState method for the base Shape. (That way, if the base Shape is itself a decoration it will also take effect.) Then compute the value of the expression: *time % 2000*

  If this value is less than 1000, set the color to Color.BLACK. Otherwise, set the color of the current object to the color of the base Shape.

9. Rotating

This class extends the ShapeDecoration class. It wraps a Shape and makes it rotate at a rate of 1 degree for every 1/100th of a second that has passed.

State (in addition to the inherited part):

- private int rotationType; [Represents the type of rotation for this shape. It will be equal to one of three values: GeometryTools.X_AXIS_ROTATION, GeometryTools.Y_AXIS_ROTATION, or GeometryTools.Z_AXIS_ROTATION. Note: The GeometryTools class is located in the package "drawingMachine", provided by us.]
- private int degrees; [Represents the current angle of rotation, in the range 0-359.]

Behaviors:

- public Rotating(Shape base, int rotationType) -- constructor that initializes the fields *base* and *rotationType* with the values of the parameters. Initialize the *degrees* field with 0.
- public void updateState(long time) -- The parameter, time, will be a value measured in milliseconds that represents a snapshot of the system clock taken by the GUI when it began to draw this shape. This method will rotate the base shape at a rate of 1 degree for every 1/100th of a second that has passed.

  The first thing this method must do is to call the updateState method for the base Shape. (That way, if the base Shape is itself a decoration it will also take effect.) Then set the field *degrees* to the value: (int)(time / 10) % 360.

- public boolean isPointInside(MyPoint p) -- This is the trickiest part of the project! The good news is that I've done the hard part for you. We need to check if a particular point of the rotated image corresponds to a point in the interior of the original base shape before it was rotated! Here is what to do:

  Pass the parameter, p, to the GeometryTools.findPreImage method, along with the other required parameters (take a look at the source-code, provided). The method will return a MyPoint object, let's call it q. What you need to do is simply check if q is inside of the base shape. (Just call the isPointInside method for the base, passing in q.) If q is in the interior of the base shape, then p must be in the interior of the rotated version, so return true. If q is not in the interior of the base shape, then p is not in the interior of the rotated version, so return false.

- public int getWidth() -- We are overriding this method because when the shape rotates, it's "Bounding Rectangle" becomes wider. A simple calculation that will always be "big enough" is to simply compute the distance from the upper left corner of the base shape's bounding rectangle to the lower right corner. (The diagonal distance). As the shape rotates, the width of it's bounding rectangle will never be bigger than this value. (Hint: Use the Pythagorean theorem!)
- public int getHeight() -- Same as above.

10. Shifting

This class extends ShapeDecoration. It will implement the isPointInside method so that numerous equally spaced diagonal lines passing through the object will not be drawn. (Kind of like thin diagonal stripes with no color.) As time progresses, these vertical lines will shift, so it looks kind of animated.

State (in addition to the inherited part):

- int cycle; This value specifies where to draw the diagonal lines. It cycles from 0 to 29, creating the animation effect.

Behaviors:

- public Shifting(Shape base) -- constructor that initializes the base with the parameter, and sets *cycle* to 0.
- protected boolean isPointInside(MyPoint p) -- we will draw all of the points in the base shape *except* for those on diagonal lines crossing through it. Check if the point p is inside the base shape. If it is, then check if the following condition is true:

  (p.getX() + p.getY()) % 29 != cycle

  If p is in the interior of the base shape, AND this condition holds, then return true; otherwise return false.

- public void updateState(long time) -- The first thing this method must do is to call the updateState method for the base. (That way, if the base shape is itself a decoration, it will also take effect.) Then set *cycle* equal to the following crazy-looking expression:

  (int)(time / 37)) % 29;

Accuracy in Calculations

Here's a big hint: You will be doing geometrical calculations based on quantities that are represented as integers -- don't forget that Java will truncate results when you do arithmetic with ints! Be sure to cast all values to type double before doing the geometrical calculations with them!

Warning: Potential for "Overflow"

Depending on how you have written your code, you may see strange things happen at rotation angles very close to 90 degrees. The reason is that the x and y coordinates of the MyPoint that you get back from the "findPreImage" method can be very large (or very large negative values). This can lead to "overflow errors" if you try to do arithmetic or other operations on them. If you're having trouble with this, I suggest hardcoding into your isPointInside methods something that checks if the x and/or y coordinates are really big (or really big negative values), and if so, just return false.

Running the GUI

To run the GUI that we have provided, run the main method in the Driver class, which is in the "drawingMachine" package.

Grading

The testing for this project will be different from previous projects. You will submit your project as usual, and the submit server will compile it. But there won't be any tests done at that time! Your project will be tested by us AFTER the due date. Don't worry: if you get the GUI that is provided to work properly with your project, then you should be fine. Don't forget that you still have to SUBMIT your project on time, even though you will not get much feedback from the Submit Server, other than being able to verify that your project was successfully received and whether or not it compiled. You will also get the usual "FindBugs" warnings if your code contains any bugs that the "FindBugs" utility is able to detect. (We've been providing this feedback all semester, in case you didn't notice, or in case your code was so good that you never saw any of the FindBugs warning messages!)
