Assignment

This assignment concerns the implementation of a system for drawing shapes. A shape is either a rectangle, or a combination of two or more shapes. Shapes can be combined in three different ways:

1. Two shapes can be **unioned** together. This means the resulting shape contains all the points from both shapes.
2. Two shapes can be **intersected**. This means the resulting shape contains all the points that were in both shapes.
3. The **difference** of two shapes can be computed. This means the resulting shape contains all the points that were in the left shape, but not in the right shape.

The shape drawing system provides two commands for drawing shapes: draw or fill. The draw command accepts a shape and a colour, and draws the outline of that shape on the canvas. Similarly. the fill command accepts a shape and a colour, and draws a filled version of the shape on the canvas; that is, every point in the shape is drawn with the given colour.

The shape drawing system accepts a sequence of one or more commands that conform to the shape drawing language. Some examples of these “shape drawing” programs, and their output, are given in Figure 1. The syntax of the shape drawing language is given as follows:

|  |  |
| --- | --- |
| **Drawing Program** | **Shape Viewer** |
| x = [50,50,100,100]  draw x #000000 | Image |
| x = [50,50,100,100]  fill x #000000 | Image |
| x = [25,25,100,100]  y = [75,75,100,100]  x = x + y  draw x #000000 | Image |
| x = [25,25,125,125]  y = [50,50,50,50]  x = x - y  draw x #aa0000 | Image |

Figure 1: Illustrating some simple shape drawing programs and their output (as seen using Canvas.show()). Observe that + and - correspond to shape union and shape diﬀerence respectively.

|  |  |  |
| --- | --- | --- |
| Expression | | Interpretation |
|  | |  |
| [x, y, width, height] | | Construct rectangle at given position with the given width and height. |
| S1 | + S2 | Construct a shape from the union of the two shapes S1 and S2 |
| S1 | & S2 | Construct a shape from the intersection of the two shapes S1 and S2 |
| S1 | − S2 | Construct a shape from the diﬀerence of the two shapes S1 and S2 |
| ( S1 ) | | A bracketed shape expression to specify evaluation order. |
| Command | | Interpretation |
|  | |  |
| draw S1 C1 | | Draw the outline of the shape given by S1 onto the canvas in colour C1. |
| fill S1 C1 | | Draw the filled shape given by S1 onto the canvas in colour C1. |
| x = S1 | | Assign the shape given by S1 to variable x. |
|  |  |  |

**Note:** Colours are specified using a 6 digit hexadecimal number of the form #rrggbb(i.e. identical tothat used for HTML). Here, rr corresponds to the red component, and takes a value between 0 . . . ff. Similarly, gg corresponds to the green component and so on.

**Part 1 — Basic Functionality**

The aim here is to implement an Interpreter class that supports the creation, assignment and drawing/filling of shapes. At this stage, you should ignore the issue of composing shapes using shape union, diﬀerence and intersection.

To get going, we suggest you follow these steps:

1. Download the file shapes.zip from the lecturer’s link.

2. Create an appropriate Eclipse project and import the files from shapes.zip

3. Implement a class called Rectangle that has an x and y coordinate, as well as width and height. This should implement the Shape interface.

4. Implement method fillShape that accepts a Shape, a Color and a Canvas. This method should first determine the bounding box of the given Shape using method boundingBox(). Then, it should iterate the coordinates within that bounding box, whilst drawing those contained in the Shape. Test this method works by writing your own main() method and using Canvas.show().

5. The Interpreter class should parse an input string and perform the required actions. To begin with, we recommend you focus on getting single line programs to work, such as these:

* + fill [10,10,50,50] #000000
  + fill ([10,10,50,50]) #00ff00

Implement methods readShape and readColor that perform the parsing of shape and colour expressions. This parses commands and expressions in a way to what is required for the shape drawing system.

1. Extend the Interpreter to support drawing the outlines of shapes. The simplest way to do this is by using a horizontal and vertical scanline algorithm. This consists of two main loops (one for horizontal lines, the other for vertical lines). The first loop goes horizontally across each line of the shape. When it moves from being outside the Shape to inside, it draws a point in the specified colour. Likewise, when moving from inside to outside, it draws a point. The second loop is similar, except that it moves in a vertical direction.
2. Extend the Interpreter to maintain a HashMap that maps variable names to their current Shape. You will need to support assignment of Shapes to variables, and the occurrence of a variable within a Shape expression.

8. Finally, ensure that the JUnit tests validFillTest and validDrawTest now pass correctly.

**HINT:** You will find it helpful to construct a simplemain()method that you can use to runshape programs and see their contents on screen.

**Part 2 — Shape Composition**

You should now extend the Interpreter class to support the +, - and & operators, which correspond to shape union, shape diﬀerence and shape intersection respectively. Having done this, all unit tests provided should now pass.

**HINT:** A sensible approach is to have one class for each of the diﬀerent shape operators, with namessuch as ShapeUnion, etc.

**HINT:** An abstract class calledShapeOperatormay be helpful in eliminating any duplicate codefound in the classes for the diﬀerent shape operators.

**Submission**

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. Note, the jar file does not need to be executable.
2. **The names of all classes, methods and packages remain unchanged**. That is, youmay add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. This is to ensure the automatic marking script can test your code.
3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, youcannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. This is to ensure the automatic marking script can test your code.
4. **You have removed any debugging code that produces output, or otherwise aﬀects the computation.** This ensures the output seen by the automatic marking script does notinclude spurious information.

**Note:** Failure to meet these requirements could result in you getting zero marks for the assignment.

**Assessment**

This assignment will be marked as a score grade (1, 2 ... 10), based primarily on the following criteria:

* **Correctness of Part 1 (30%)** — does submission adhere to specification given for Part 1.
* **Correctness of Part 2 (30%)** — does submission adhere to specification given for Part 2.
* **Style (20%)** — does the submitted code follow the style guide and have appropriate comments(inc. Javadoc)
* **UML diagrams (20%)** does the UML class diagrams describe all classes and their attributes/methods in your assignment. Note that unless you finish Part 1 and Part 2, your score will be substracted.

The qualitative marks for style are given for the following points:

* **Division of Concepts into Classes**. This refers to howcoherentyour classes are. That is,whether a given class is responsible for single specific task (coherent), or for many unrelated tasks (incoherent). In particular, big classes with lots of functionality should be avoided.
* **Division of Work into Methods**. This refers to how well a given task is split across methods.That is, whether a given task is broken down into many small methods (good) or implemented as one large method (bad). The approach of dividing a task into multiple small methods is commonly referred to as divide-and-conquer.
* **Use of Naming**. This refers to the choice of names for the classes, fields, methods and variablesin your program. Firstly, naming should be consistent and follow the recommended Java Coding Standards (see <http://gee.cs.oswego.edu/dl/html/javaCodingStd.html>). Secondly, names of items should be descriptive and reflect their purpose in the program.
* **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields andmethods. We certainly expect all public and protected items to be properly documented. For example, when documenting a method, an appropriate description should be given, as well as for its parameters and return value. Good style also dictates that private items are documented as well.
* **Other Comments**. This refers to the use of commenting within a given method. Generallyspeaking, comments should be used to explain what is happening, rather than simply repeating what is evident from the source code.
* **Consistency**. This refers to the consistent use of indentation and other conventions. Generallyspeaking, code must be properly indented and make consistent use of conventions for e.g. curly braces.