PGCert IT: Programming for Industry

Lab 15: Design Patterns

Before you start

The BitBucket repository for this lab is located at this link. Fork it to get the starting point for this lab. Take a look at the examples given in the project, as they should help you to complete the exercises. Note the use of sub-projects in this lab exercise – the starter project contains compile errors, and we didn't want this to interfere with your ability to run the example code!

In this lab, we will be working with Bounce from Lab 12. It is important to do Lab 12 or at least understand how the code works before attempting the following exercises.

Exercise One: A Nesting Shape for Bounce

A `NestingShape` instance is unique in that it contains zero or more `Shapes` that bounce around inside it. The children of a `NestingShape` instance can be either simple `Shapes`, like `RectangleShape` and `OvalShape` objects, or other `NestingShape` instances. Hence, a `NestingShape` object can have an arbitrary containment depth. A `NestingShape` object paints a rectangle at the edge of its bounding box (similar to a `RectangleShape`), and then paints its children.

The specification for `NestingShape` is presented below. Given that a `Shape` object can now be a child of a `NestingShape`, also add and implement the methods specified for class `Shape`.

Once the `NestingShape` is complete, and the changes to the `Shape` class have been made, you should be able to create shape hierarchies and display them in your `AnimationViewer` just as you could with any other shape. As an example, consider the following code snippet, which should produce output similar to the screenshot in Figure 1.

```
shapes.add(new RectangleShape(0, 0, 2, 3));
shapes.add(new RectangleShape(10, 10, 5, 7));
shapes.add(new OvalShape(15, 50, 3, 2, 40, 40));
shapes.add(new GemShape(100, 200, 3, 2, 100, 40));
shapes.add(new GemShape(200, 100, 4, 5, 40, 40));
shapes.add(new DynamicRectangleShape(300, 200, 5, 7, 30, 50, Color.red));
```

```
NestingShape topLevelNest = new NestingShape(0, 0, 2, 2, 200, 200);
NestingShape midLevelNest = new NestingShape(0, 0, 2, 2, 100, 100);
NestingShape bottomLevelNest = new NestingShape(5, 5, 2, 2, 75, 75);
ImageShape tRex = new ImageShape(20, 20, 3, 4, "TRex.png", 0.2);
RectangleShape simpleShape = new RectangleShape(1, 1, 1, 1, 5, 5);

midLevelNest.add(bottomLevelNest);
midLevelNest.add(tRex);
topLevelNest.add(midLevelNest);
bottomLevelNest.add(simpleShape);

shapes.add(topLevelNest);
```

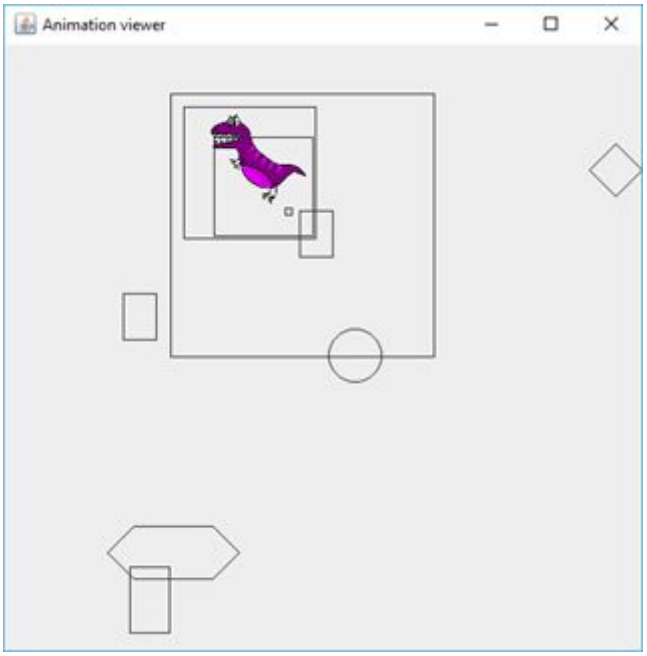

Figure 1: Screenshot showing the output when the code is used to add shapes to AnimationViewer.

Specification for class `NestingShape`

The following methods are required for the `NestingShape` class:

`public NestingShape()`
Creates a `NestingShape` object with default values for state.

`public NestingShape(int x, int y)`
Creates a `NestingShape` object with specified location values, and default values for other state items.

`public NestingShape(int x, int y, int deltaX, int deltaY)`

Creates a `NestingShape` with specified values for location, velocity and direction. Non-specified state items take on default values.

`public NestingShape(int x, int y, int deltaX, int deltaY, int width, int height)`

Creates a `NestingShape` with specified values for location, velocity, direction, width and height.

`public void move(int width, int height)`

Moves a `NestingShape` object (including its children) with the bounds specified by arguments width and height. Remember that the `NestingShape`'s children only want to move within the bounds of the `NestingShape` itself, rather than the whole screen.

`public void paint(Painter painter)`

Paints a `NestingShape` object by drawing a rectangle around the edge of its bounding box. The `NestingShape` object's children are then painted.

`public void add(Shape child) throws IllegalArgumentException`

Attempts to add a Shape to a `NestingShape` object. If successful, a **two-way link** is established between the `NestingShape` and the newly added Shape. This method throws an `IllegalArgumentException` if an attempt is made to add a Shape to a `NestingShape` instance where the Shape argument is already a child within a `NestingShape` instance. An `IllegalArgumentException` is also thrown when an attempt is made to add a Shape that will not fit within the bounds of the proposed `NestingShape` object.

`public void remove(Shape child)`

Removes a particular Shape from a `NestingShape` instance. Once removed, the two-way link between the `NestingShape` and its former child is destroyed. This method has no effect if the Shape specified to remove is not a child of the `NestingShape`.

`public Shape shapeAt(int index) throws IndexOutOfBoundsException`

Returns the Shape at a specified position within a `NestingShape`. If the position specified is less than zero or greater than the number of children stored in the `NestingShape` less one this method throws an `IndexOutOfBoundsException`.

`public int shapeCount()`

Returns the number of children contained within a `NestingShape` object. Note this method is not recursive - it simply returns the number of children at the top level within the callee `NestingShape` object.

`public int indexOf(Shape child)`

Returns the index of a specified child within a `NestingShape` object. If the `Shape` specified is not actually a child of the `NestingShape` this method returns -1; otherwise the value returned is in the range 0 .. `shapeCount()` - 1.

`public boolean contains(Shape child)`

Returns true if the Shape argument is a child of the `NestingShape` object on which this method is called, false otherwise.

Additional specification for class `Shape`

The following additional methods are required to be implemented in the `Shape` class:

`public NestingShape parent()`

Returns the `NestingShape` that contains the `Shape` that method parent is called on. If the callee object is not a child within a `NestingShape` instance this method returns `null`.

`public List<Shape> path()`

Returns an ordered list of Shape objects. The first item within the list is the root `NestingShape` of the containment hierarchy. The last item within the list is the callee object (hence this method always returns a list with at least one item). Any intermediate items are `NestingShapes` that connect the root `NestingShape` to the callee `Shape`.

For example, when given the code in Listing 1 above, a call to `tRex.path()` would yield: `[topLevelNest, midLevelNest, tRex]`.

Hints

A `NestingShape` has its own coordinate system, so that Shapes within it are within the coordinates of the `NestingShape`. So if a `Shape` with a location of (10,10) is in a `NestingShape`, it will be located 10 pixels below and 10 pixels to the right of the top-left corner of the `NestingShape`. Similarly, a `NestingShape`'s children should bounce within their parent's bounding box, rather than the width & height of the entire `AnimationViewer`.

In addition to implementing new methods in class `NestingShape`, methods handling painting and movement inherited from `Shape` will need to be overridden to process a `NestingShape` object's children.

The method that `Shape` subclasses implement to handle painting should not be modified when completing this exercise. In other words, shapes should not have to be concerned with whether or not they are children within a `NestingShape` when painting themselves.

- One way of cleanly implementing `NestingShape`'s painting behaviour is to adjust the coordinate system by specifying a new origin (the `NestingShape`'s top left corner) that corresponds to a point in the original coordinate system. This can be achieved using `Graphics`' `translate()` method. Once translated, all drawing operations are performed relative to the new origin.

- Note that any translation should be reversed after painting a `NestingShape`.
- The supplied `Painter` interface, and the corresponding `GraphicsPainter` and `MockPainter` classes, already have the `translate()` method implemented – you just need to call it appropriately.

Use the unit tests provided for class `NestingShape` (i.e. `TestNestingShape`) to help locate any defects in your implementation. We will cover these tests in more detail, including how to write your own, in the following week.

Exercise Two: A NestingShape Visualization

Develop a new `AnimationViewer` for the Bounce application that, in addition to displaying an animation of shapes, includes a GUI component which displays a visual representation of the hierarchy of shapes being animated. Within a Bounce animation, the root node can be a `NestingShape` that contain other Shapes, be they "simple" Shapes without children, or composite Shapes that themselves contain others.

Your solution should minimize the changes made to existing code. In particular, you should not need to make any changes to classes `Shape` and `NestingShape`, which are now "tried and tested". Similarly, your solution should maximize reuse – you should use Swing's `JTree` class that has been purposely designed and developed for visualizing hierarchical structures.

A slightly modified `AnimationViewer`, called `Ex02AnimationViewer`, which contains a `JTree` has been provided to you. You may use this as a starting point if you wish. If you use this as the starting point, then you should see a result similar to the screenshot below when you've completed this exercise.

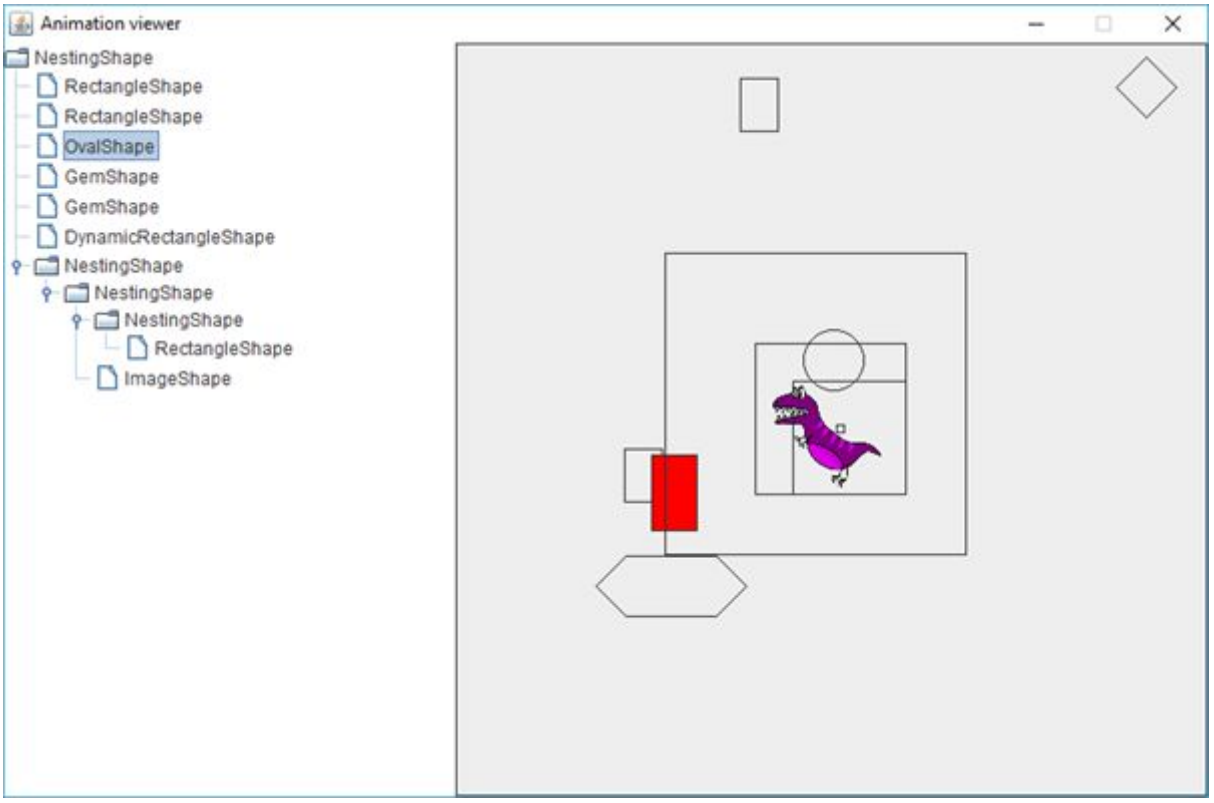

Figure 2: Screenshot showing a complete implementation of exercise two, with a shape hierarchy viewed in both an `AnimationViewer` and a `JTree`.