

COMPSCI 718 Programming for Industry

Bounce: increment I and II

Code Deadline: Friday June 10, 23:59
Interview: Tuesday June 07 - Thursday June 09
Report Deadline: Monday June 20, 23:59

May 24, 2022

Introduction

This project serves to reinforce skills in object-oriented programming. You will work with JDK libraries and the JUnit testing tool.

Bounce involves an animation comprising an extensible set of shape types. Shapes have in common knowledge of their position, velocity, direction and bounding box, while each kind of special shape has a specific way of rendering itself.

You will need to investigate a couple of `java.awt` classes: `Graphics` and `Color` to complete this part of the project. Browsing the online JDK API should be sufficient for the needs of this part.

Assessment criteria

Each task is associated with assessment criteria. In addition to the task-specific criteria you will need to demonstrate that you have understood the work in your reflective report.

Interview

You will have an interview to demonstrate your understanding of the work you have done in Increment I and II. We will ask you questions in your scheduled interview slot. You must attend the interview session to have your work assessed. Failure to demonstrate work in the interview will result in a mark of zero for Increment I and II. Prior to your interview session, you should have everything ready – you will have no more than 5 minutes to convince us that you have met the stated assessment criteria.

Submission

You submit by ensuring your GitHub repository for Task One to Task Six is up-to-date. We will download your repository for marking on the due date.

The reflective report is due by 23:59 on Monday 20th June. The reflective report should include the class diagram, sequence diagram, answers to all questions, and reflections on your learning experience in this course. You should submit your report as a PDF to Canvas.

Constraints

The changes you make to the `Shape` class hierarchy should not break any existing code (e.g. the `AnimationViewer` class) that has been written to use the hierarchy.

Reflective Report

As part of the final project, you need to submit an individual reflective report that details your learning experience. In your report, you also need to model the **bounce** package and answer all questions relating to the project. The reflection part of your report should reflect your overall experience of this course. Here are some questions that might help you to reflect:

- What modules did you learned the most from?
- What was your favourite aspect of this course? Why?
- What concepts did you find straightforward / difficult?
- What did you enjoy the most / least in this course? Why?
- What would you like to learn further or do more in this course?

Please use the IEEE conference template for the final report, using A4 page format, 10pt font for the body, and conference style. The reflective part of the report should be not more than 2 pages using IEEE format.

Model the **bounce** package

Modelling is generally carried out when starting a new project to understand the problem to be solved and to assist with developing a software solution. However, modelling is also useful to help you understand how an existing piece of software works. This task involves *reverse engineering* a model from source code to help you understand the structure and behaviour of the Bounce application prior to developing it further in subsequent tasks.

You should construct a class diagram to model the static structure of the Bounce application, plus a sequence diagram to reveal how instances of the classes work together. Your class diagram should capture the key classes and interfaces and the relationships between them.

The sequence diagram should capture important interactions between key instances to effect the following scenario. The `AnimationViewer`'s object knows of exactly two `RectangleShape` instances. The scenario starts with the `Timer` generating an `Action` event. The sequence diagram should expose object interaction in response to the `ActionEvent` that results in the `RectangleShape` objects being painted and moved.

Note that you only need to model the initial bounce package, i.e. only the given classes in Bounce I. The UML diagrams need not be created using a tool. Hand-drawn diagrams should be clear and legible.

Hints:

- Remember that models are by definition an abstraction of what is being modelled. Therefore a model should expose what is important and suppress irrelevant detail.
- Your class diagram need not include the classes associated with testing.

Questions on Increment I and II

1. Drawing on your work in Increment I and II, given an example of polymorphism.
2. What is the advantage of having the `Painter` interface?
3. Explain the changes you have made in Task Three and why the changes are in accordance with good object-oriented programming practice.
4. Drawing on your work in Increment II, why is it useful for `Shapes` to have parent references?
5. Justify a design decision that you have made in Task Six and why the decision follows good object-oriented programming practice.

Questions on Increment III

1. What is the name of the design pattern you have applied in Task One? What is a key benefit of applying this design pattern in Task One? Explain the benefit in terms of the Bounce application.
2. Explain the role that a Task 2 object plays in the application.
3. Why is multi-threading generally necessary in a GUI application?
4. Explain how you have used `SwingWorker` in this project.
5. Explain the usefulness of unit testing in software development as well as in this application.

Increment I

For this increment, you will need to clone the source code from GitHub Classroom.

Task One: Add an Oval shape

Write unit tests for `OvalShape`, a new type of `Shape` that moves like the other shapes but displays itself as an oval shape that fits into its bounding box. It must be possible to create an `OvalShape` using the same set of creation options as offered by class `Shape`. Implement `OvalShape` and include such a shape in your animation.

Assessment criteria

- The class hierarchy should be developed sensibly and in accordance with good object-oriented programming practice.
- The unit test(s) should demonstrate that an `OvalShape` instance paints itself correctly.
- The Bounce application should demonstrate correct functioning of an `OvalShape` instance.

Task Two: Add a Gem shape

Write unit tests for a `GemShape`, a new type of `Shape` which moves like the other shapes but displays itself as a hexagonal shape that fits into its bounding box. Again, it must be possible to give values for instance variables at construction time or rely on default values. Implement `GemShape` and include such a shape in your animation.

The `GemShape` is to be painted to fit within the given width and height, using `drawLine()` calls to the `Painter`. The lines of the gem are to be painted starting with the left-most vertex and proceeding in a clock-wise direction.

The definition of the coordinates of the points is phrased here in terms of (x,y,width,height):

The top-left and bottom-left vertices of a `GemShape` are normally 20 pixels to the right of the left hand side of the shape. Similarly, the top-right and bottom-right vertices of a `GemShape` are normally 20 pixels to the left of the right hand side of the shape. However, if the width of a `GemShape` is less than 40 pixels, the top-left and top-right vertices are both positioned at point $(x + \text{width}/2, y)$. Similarly, the bottom-left and bottom-right vertices are both positioned at point $(x + \text{width}/2, y + \text{height})$. In other words, “small” `GemShapes` are four-sided figures.

Hints:

- When developing the unit tests for `GemShape`, you should include test cases to demonstrate that the “small” and “regular” `GemShape` instances are constructed correctly. One way of writing these tests is to use a `MockPainter` and to check that its log contains the correct line sequences for `GemShape` objects.

Assessment criteria

- The class hierarchy should be developed sensibly and in accordance with good object-oriented programming practice.
- The test cases should show that a `GemShape` paints itself correctly based on its width and height.
- The Bounce application should demonstrate correct functioning of a `GemShape` instance.

Task Three: Add a Dynamic Rectangle shape

Write unit tests for a `DynamicRectangleShape`, a new type of `Shape` that paints itself similarly to a `RectangleShape`. At construction time, an additional `java.awt.Color` argument can be supplied. If not given, the new `RectangleShape` object's color should default to black.

A `DynamicRectangleShape` moves like the other shapes, but it sometimes changes its appearance when it bounces. After it bounces off the left or right wall it paints itself as a solid figure, in the color specified at construction time. After it bounces off the top or bottom wall it switches its appearance to that of a `RectangleShape`, i.e. rendering itself with an outline. If it bounces off both walls, the vertical (left or right) wall determines its appearance.

Implement it and include such a shape in your animation.

Hints:

- To thoroughly test `DynamicRectangleShape`, you should prepare 8 test cases that verify that a `DynamicRectangleShape` instance renders itself correctly when bouncing off *one* wall: top, bottom, left and right. In addition you should develop a further four test cases to check that following a bounce of *two* walls (top-right, top-left, bottom-right, bottom-left) a `DynamicRectangleShape` object's appearance is consistent with its specification.
- To add support for color, introduce three new methods to the `Painter` interface: `fillRect()`, `getColor()` and `setColor()`. `fillRect()` should take the same arguments as `awt.Graphics.fillRect()`. `getColor()` should return a `java.awt.Color` value and `setColor()` should take a `java.awt.Color` argument. Using these methods, you can query the current color before setting a new color and drawing a filled rectangle. After drawing the filled rectangle, you can reset the current color to its original value.
- The `MockPainter` implementation of `Painter` should be extended to log color changes and drawing of filled rectangles. Class `GraphicsPainter` can implement the new methods to use its `java.awt.Graphics` object.

Assessment criteria

- The test cases should be sufficiently thorough to demonstrate that a `DynamicRectangleShape` meets the above specification.
- The implementation of `DynamicRectangleShape` and any changes you might make to `Shape` should be in accordance with good object-oriented programming practice. Note that this criterion will not be satisfied if you duplicate code in `Shape` and `DynamicRectangleShape`.

- The Bounce application should demonstrate correct functioning of a `DynamicRectangleShape` instance.

Task Four: Add a Border shape

Write unit tests for a `BorderShape`, a new type of `Shape` that moves like the other shapes. However, unlike the other `Shapes`, it contains a `Shape` and moves with that `Shape`. That is, its position is completely determined by the contained shape. It paints a rectangular border around its embedded `Shape`, with a gap of 2 pixels between the internal shape and the border. The `Shape` can be bordered an arbitrary number of times (borders around borders). On bouncing, none of the borders should pass beyond the world's boundaries. Implement `BorderShape` and include such a shape in your animation.

Assessment criteria

- The test cases should be sufficiently thorough to demonstrate that a `BorderShape` meets the above specification.
- The implementation of `BorderShape` and any changes you might make to `Shape` should be in accordance with good object-oriented programming practice. Note that this criterion will not be satisfied if you duplicate code in `Shape` and `BorderShape`.
- The Bounce application should demonstrate correct functioning of a `BorderShape` instance.

Optional Task: Be imaginative!

Add a new and interesting kind of shape. Possible suggestions include (but are not limited to):

- An *aggregate shape* that contains two member `Shapes`. On bouncing, the aggregate shape toggles between its members and paints itself accordingly.
- A shape that displays an image. For this, you could investigate class `java.awt.Image`. There are several options for loading `Image` instances, e.g. `javax.imageio.ImageIO`.

Increment II

This increment involves applying design knowledge to evolve the Bounce application as follows. First, the `Shape` class hierarchy is to be extended to support the concept of a nesting shape – a shape that can contain other shapes, be they simple or nesting shapes themselves. Second, a text painting facility is to be integrated into the hierarchy.

For this increment, you will need to extend the code you have developed from Increment I. You may need to make some changes to your code.

The source code for `TestNestingShape` is available on Canvas. You can use this class to help identify any defects in your solution to Task Five.

Task Five: Introduce class `NestingShape`

Define a new subclass of `Shape`, `NestingShape`. 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 and then paints its children.

The specification for `NestingShape` is given in Appendix 1. Given that a `Shape` object can now be a child of a `NestingShape`, also add and implement the methods specified in Appendix 2 for class `Shape`.

Once implemented, add a `NestingShape` instance, with a containment depth of at least three, to your animation.

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`.
- 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 task. 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`.
- In implementing painting of a `NestingShape` as suggested above, the `Painter` interface will need to include a `translate(int x, int y)` method. In the `GraphicsPainter` implementation, this method should be implemented to forward the request to a `GraphicsPainter`'s `Graphics` instance. For the `MockPainter` implementation, take a look at the provided test.
- Use the unit tests provided for class `NestingShape` to help locate any defects in your implementation.

Assessment criteria

- The class hierarchy should be developed sensibly and in accordance with good object-oriented programming practice.
- The unit test(s) should demonstrate that a `NestingShape` instance behaves correctly.
- The Bounce application should demonstrate correct functioning of a `NestingShape` instance, with a containment structure of three levels.

Task Six: Display text for any `Shape`

Develop the `Shape` class hierarchy to allow text to be displayed when a shape is painted. Text should be centred, horizontally and vertically, in a `Shape`'s bounding box, but may extend beyond the left and right sides.

In this task, you need to guarantee that if a shape is associated with text it will always be painted. That is, this responsibility should not be left to the subclasses of `Shape`. Any `Shape` subclasses should be able to display associated text.

Hints:

- See the Java class `java.awt.FontMetrics` for details of how to centre text.
- Add a method `drawCentredText()` to the `Painter` interface. In `GraphicsPainter`, implement this method to calculate the position at which to display the text and paint the text using the `drawString()` method of `java.awt.Graphics`. In `MockPainter`, either an empty `drawCenteredText()` method or one that simply logs that a request has been made to draw centered text is fine. Note that it is not possible for `MockPainter` to log the coordinates that the text has been drawn at because this requires a `FontMetrics` object to extract data from to calculate the coordinates. A `FontMetrics` object can be obtained from a `Graphics` object – which is not available to a `MockPainter`.

Assessment criteria

- The class structure should be developed sensibly and in accordance with good object-oriented programming practice.
- The solution should guarantee painting of text for an instance of any `Shape` subclass that is associated with text.
- The Bounce application should demonstrate correct functioning of any `Shapes` with text.

Appendix 1

Listing 1: Specification for class NestingShape

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

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

/**
 * 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);

/**
 * Creates a NestingShape with specified values for location, velocity,
 * direction, width and height.
 */
public NestingShape(int x, int y, int deltaX, int deltaY,
    int width, int height);

/**
 * Moves a NestingShape object (including its children) within the bounds
 * specified by arguments width and height.
 */
public void move(int width, int height);

/**
 * 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 paint(Painter painter);

/**
 * 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. Note that this method has package visibility – for reasons
 * that will become apparent in Bounce III.
 * @param shape the shape to be added.
 * @throws 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 add(Shape shape) throws IllegalArgumentException;

    /**
     * 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. Note that this method has package
     * visibility – for reasons that will become apparent in Bounce III.
     * @param shape the shape to be removed.
     */
    public void remove(Shape shape);

    /**
     * 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.
     * @param index the specified index position.
     */
    public Shape shapeAt(int index) throws IndexOutOfBoundsException;

    /**
     * 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 shapeCount();

    /**
     * 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.
     * @param the shape whose index position within the NestingShape is
     * requested.
     */
    public int indexOf(Shape shape);

    /**
     * Returns true if the Shape argument is a child of the NestingShape
     * object on which this method is called, false otherwise.
     */
    public boolean contains(Shape shape);

```

Appendix 2

Listing 2: Specification for new Shape methods

```
/**
 * 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 NestingShape parent();

/**
 * 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. E.g., given:
 *
 * NestingShape root = new NestingShape();
 * NestingShape intermediate = new NestingShape();
 * Shape oval = new OvalShape();
 * root.add(intermediate);
 * intermediate.add(oval);
 *
 * a call to oval.path() yields: [root, intermediate, oval]
 */
public List<Shape> path();
```
