COMPSCI 230 (S1 2020) Assignment

Bounce: part II/III

Introduction

This is the second of three parts comprising the Bounce project. Bounce II involves applying design knowledge to further develop the Bounce application. 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 `Shape` class hierarchy.

Assessment criteria

Each task is associated with assessment criteria. To check that you have met the criteria, each task is associated with CodeRunner 3 (CR3) tests. The tests include a mix of:

- tests that run your compiled code to check that it is functionally correct and,
- other tests that examine your source to ensure that it conforms to object-oriented principles.

The CR3 submission carries **80%** of the marks, and the ADB submission carries the remaining **20%**.

The marking scheme for CR3 questions is provided for each task. *ADB submissions will be marked based on whether your submitted code compiles and runs successfully to show the expected GUI as per specifications given in this assignment brief.* As such, if your code passes all four CR3 tests, and your code presents the expected GUI in *your* IDE, submitting the full code should work fine at the marker's end too. You can ensure this by carefully packaging, zipping and submitting the verified code to ADB as per the submission instructions given below.

Submission

For Bounce II, you must:

- (8 marks) pass the *CodeRunner* tests by Week 10 (Saturday 23:59, 17 October 2020)
    - Visit **coderunner3.auckland.ac.nz.**
    - Under 'Assignments', you should access 'A3 – Bounce II' Quiz.
    - The quiz has a total of four questions, each of which requires you to paste the source code of one of your classes that you wrote in your IDE to complete each task.
    - Each task has two CR questions (one question for passing tests and another for static analysis/specification checking of source code)
    - You may click on 'Check' for each question multiple times without any penalty.
    - You may however make only one overall CR submission for the whole quiz.
- (2 marks) submit your **source code (including the original Canvas code)** to the Assignment Drop Box (ADB – ***adb.auckland.ac.nz***) by Week 10 (Saturday 23:59, 17 October 2020).
    - The code submission **must** take the form of **one zip file** that *maintains the Bounce package structure*.
    - You **must** name your **zip** file as "**YourUPI_230a3_2020.zip**".
    - You **may** make more than one ADB submission, but note that every submission that you make replaces your previous submission.
    - Submit **ALL** your source files in every submission.
    - Only your most recent submission will be marked.

- Please double check that you have included all the files required to run your program in the zip file before you submit it. **Your program must compile and run (through Java 1.8).**
- Include the code for **creating and showing all required shapes** in `AnimationViewer` class.

NOTE: It would be ideal to first complete all tasks in an IDE like Eclipse, and then only proceed to answer CR3 questions once you have finished writing working code for this iteration in your IDE.

Constraints

For all three parts of the of the assignment, any changes you make to the `Shape` class hierarchy must not break existing code (e.g. the `AnimationViewer` class). In particular, **class `Shape` must provide the following public interface** *(note this is not the Java "interface"; here the term "interface" refers to the set of public methods in a class that may act as entry points for the objects of other classes)* **at all times**:

- `void paint(Painter painter)`
- `void move(int width, int height)`

Task 1: add 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.

The Javadoc description for `NestingShape` is given in Appendix 1. Appendix 2 describes additional functionality required of class `Shape` for this task.

The resulting `Shape` and `NestingShape` structure is an application of the **Composite design pattern**.

Specific requirements

Specific requirements are listed in the Appendices 1 and 2.

Once created, edit the `AnimationViewer` class to add in instance of `NestingShape`, with a depth of at least three containment levels, to your animation.

Hints

- A `NestingShape` has its own coordinate system, so that `Shape` instances 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`'s subclasses implement to handle painting should not be modified when completing this task. In other words, `Shape` objects 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 use **graphics translation** – adjusting 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 `Painter`'s `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`.

E.g. the following code sets the new origin (0, 0) to (60, 40). After this, a `Painter` call like `painter.drawRect(10, 10, 40, 20)` would cause the rectangle to be painted at absolute position 70, 50. The second `translate()` call restores the origin to (0, 0).

```
painter.translate(60, 40);
// Code to paint children …
painter.translate(-60, -40);
```

Testing

From the CodeRunner quiz named *Bounce II*, complete the first two questions:

- (3 marks) *Bounce II NestingShape*, which runs a series of tests on your `NestingShape` class.
- (1 mark) *Bounce II NestingShape (static analysis),* which examines the source code for your `NestingShape` class.

Your code will need to pass all tests.

Task 2: add provision for any Shape to display text

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

In designing the revised class structure, you must guarantee that if a shape is associated with text that it will *always* be painted. This responsibility should not be left to developers of `Shape` subclasses. To elaborate, consider the intention to make your `Shape` class hierarchy available – only the bytecode and not the source files – to other developers. The above guarantee should hold for any Shape subclasses added by other developers in the future. Note that since such developers will not have the source code for your class hierarchy they cannot edit your classes; all they can do is extend the classes that you provide.

Specific requirements

- You must solve this problem by applying the **Template Method design pattern**. Recall the constraint that it must be possible to call `paint(Painter)` and `move(int, int)` on any object that is an instance of class `Shape` or its subclasses.
- The **mandatory hook method** required for the Template Method solution must be named `doPaint()` and take a `Painter` parameter.
- Class `Shape` and all of its subclasses should provide a 7-argument constructor where the arguments are *x, y, deltaX, deltaY, width, height* and *text*, where *text* is a `String`. This adds the capability for text to be supplied when constructing a shape. If other constructors are used, the resulting shape is assumed not to have text.
- During painting of a shape, any text should be painted last – so that it is superimposed over any solid figure or image that has been painted for the shape.

Once you have implemented the text painting facility, add some shapes with text to your animation.

Testing

From the CodeRunner quiz named *Bounce II*, complete the remaining two questions:

- (3 marks) *Bounce II Shapes with text*, which runs a series of tests on your classes.
- (1 mark) *Bounce II Shapes with text (static analysis),* which examines the source code for your classes.

Your code will need to pass all tests.

Debugging

I strongly recommend that you use Eclipse for all assignments, and use Eclipse debugging feature for your assistance.

ACADEMIC INTEGRITY

The purpose of this assignment is to help you develop a working understanding of some of the concepts you are taught in the lectures.

We expect that you will want to use this opportunity to be able to answer the corresponding questions in the exam.

We expect that the work done on this assignment will be your own work.

We expect that you will think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help.

This really shouldn't be necessary, but **note the comments below about the academic integrity related to this assignment**:

The following sources of help are acceptable:

- Lecture notes, tutorial notes, skeleton code, and advice given by us in person or online, with the exception of sample solutions from past semesters.
- The textbook.
- The official Java documentation and other online sources, as long as these describe the general use of the methods and techniques required, and do not describe solutions specifically for this assignment.
- Piazza posts by (or endorsed by) an instructor.
- Fellow students pointing out the cause of errors in your code, *without providing an explicit solution*.

The following sources of help are **NOT** acceptable:

- Getting another student, friend, or other third party to instruct you on how to design classes or have them write code for you.
- Taking or obtaining an electronic copy of someone else's work, or part thereof.
- Give a copy of your work, or part thereof, to someone else.
- Using code from past sample solutions or from online sources dedicated to this assignment.

The Computer Science department uses copy detection tools on all submissions. Submissions found to share code with those of other people will be detected and disciplinary action will be taken. To ensure that you are not unfairly accused of cheating:

- Always do individual assignments by yourself.
- Never give any other person your code or sample solutions in your possession.
- Never put your code in a public place (e.g., Piazza, forum, your web site).
- Never leave your computer unattended. You are responsible for the security of your account.
- Ensure you always remove your USB flash drive from the computer before you log off, and keep it safe.

Concluding Notes

Not everything that you will need to complete this assignment has or will be taught in lectures of the course. You are expected to use the Oracle Java API documentation and tutorials as part of this assignment.

Post any questions to Piazza or ask via Zoom (avoid emails) – this way, the largest number of people get the benefit of insight!

We may comment along the lines on Piazza to deal with any widespread issues that may crop up.

Appendix 1 Javadoc for class NestingShape

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

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

/**
 * Creates a NestingShape with specified values for location, velocity and direction. Non−specified
 * attributes 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. A NestingShape first moves itself, and then moves its children.
 */
public void move (int width, int height);

/**
 * Paints a NestingShape object by drawing a rectangle around the edge of its bounding box. Once
 * the NestingShape's border has been painted, a NestingShape paints its children.
 */
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.
 */
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.
 ∗/
```

```
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 Javadoc for class Shape

```
/**
 * 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();

/**
 * Sets the parent NestingShape of the shape object that this method is called on.
 */
protected void setParent(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 java.util.List <Shape> path();
```