COMPSCI 718 Programming for Industry
Bounce: increment III

Code Deadline: Friday June 11, 23:59
Report Deadline: Friday June 11, 23:59

January 24, 2021

Introduction

This is the final installment of the Bounce project, and perhaps the most challenging. The emphasis of Bounce III is on working with design patterns. You are required to complete and extend the application in a way that makes appropriate use of design patterns.

Bounce III is a model/view application that presents three views of a shared model. Such applications are commonplace and introduce the need for views to be mutually consistent and synchronised with a model whose state changes at run-time.

Once you have completed the tasks, you should have an application that looks similar to that shown in Figure 1.

Submission

You should submit by ensuring your GitHub repository for Increment III is up-to-date. Don't forget to submit your reflective report via Canvas before the due date.

Preparation

You will need to clone the source code from GitHub Classroom. The GitHub repository includes the source code, including test cases, a properties file and a large image file. The properties file allows you to specify application properties that are read by the application. The image file can be used to help test your application.

You are strongly advised to construct a model of the application that captures the key classes, instances and their interactions. The model needn't be "proper" UML – informal diagrams that allow you to discover, document and visualise the application's structure would be sufficient to obtain the required understanding. It is common in industry to work on existing software projects, and an ability to understand their structure is an important skill.

You need to copy all the files from the `bounce` package that you have developed for Bounce I and II into the `bounce` package of Bounce III. Please read the given code carefully before you copy all the files. You may need to make some small changes to your `bounce` code – see the following section for details.

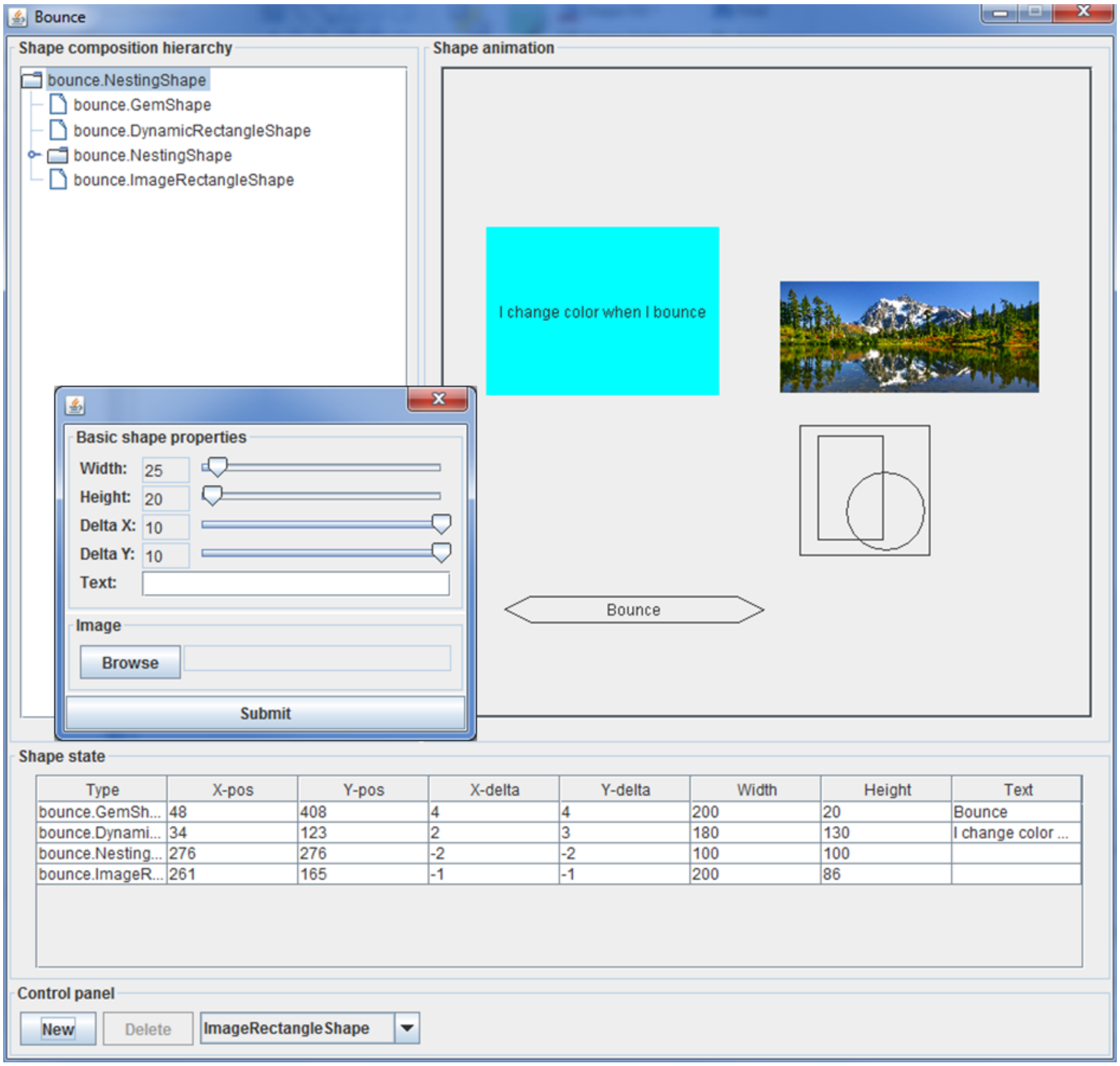

Figure 1: Bounce model/view application

Supplied code

The Bounce III project is organised around five packages:

- *bounce*. This includes the Shape hierarchy classes, Painters and the classes and interfaces concerned with the Shape *model*: ShapeModel, ShapeModelListener and ShapeModelEvent.

- *bounce.bounceApp*. This package contains the main Bounce application class and BounceConfig, which provides a convenient way of accessing application properties. This package also contains a JUnit *suite* class that collects all the test cases for the application. Running the suite runs all the unit tests.

- *bounce.forms*. Package bounce.forms contains GUI form classes that allow the user to specify Shape attribute values. These include ColourFormElement for specifying a shape's colour, ImageFormElement for specifying a Shape's image file, and ShapeFormElement for specifying general Shape attributes such as width and height. In addition, this package containes FormHandler implementations that read form data and which instantiate particular Shape subclasses.

- *bounce.forms.util*. This contains application-independent artefacts for building forms. Form, FormElement and FormHandler are basic interfaces for representing forms, form elements containing data fields, and form processing respectively. FormComponent and FormElementComponent are Swing component implementations of Form and FormElement. FormUtility is convenience class for laying out forms. Of the five Bounce packages, this is the one that you need to be least familiar with.
- *bounce.views*. This package includes the view classes for this application and their associated adapters that link them to a ShapeModel.

The supplied code makes the following assumptions about your bounce package code:

- All Shape subclasses have a 7-argument constructor that takes the following parameters in order: x, y, deltaX, deltaY, width, height and text. The first 6 parameters are of type int, while text is of type String.
- Class Shape's public interface includes methods move(int, int) and paint(Painter), according to the original Bounce specification.
- Class Shape defines a protected abstract method doPaint(Painter) that all subclasses override to perform subclass-specific painting. Either modify your Shape hierarchy classes to use doPaint() in this way, or edit the supplied bounce.ImageRectangleShape class to work with your own Shape hierarchy's painting mechanism.
- Class DynamicRectangleShape has an 8-argument constructor that takes the following parameters in order: x, y, deltaX, deltaY, width, height, text and colour. The first 6 parameters are of type int, text is a String and colour's type is java.awt.Color.
- Class NestingShape from Bounce II satisfies its specification.
- The Painter interface includes method drawImage(Image img, int x, int y, int width, int height) that allows java.awt.Image objects to be painted. A suitable implementation in GraphicsPainter simply delegates the call to its Graphics object:

  _g.drawImage(img, x, y, width, height, null);.

Once your code conforms to the above expectations, the project will still contain compilation errors because the tasks have yet to be completed. Specifically, the following units will have compilation errors:

- bounce.bounceApp.Bounce. This is the main application class. It requires bounce.views.Task2 to be implemented.
- bounce.forms.TestImageShapeFormHandler. This test case requires you to implement class bounce.forms.ImageShapeFormHandler in task 3.
- bounce.views.TestTask1 and bounce.views.TestTask2 are test-case classes. They will compile when bounce.views.Task1 and bounce.views.Task2 respectively have been implemented.
- bounce.views.TreeViewer requires that Task1 is implemented.

Task One: Implement class **bounce.views.Task1**

A key part of a ShapeModel object is its composite structure of Shapes. When completed, the Bounce application should include a hierarchical view of this composition structure. Rather than reinvent the wheel, it makes sense to reuse Swing's JTree component that is tried and tested and which is intended for visualising hierarchical structures. However, there is an interface mismatch problem in that a JTree component is not able to render ShapeModel objects directly. Rather, a JTree component can render any TreeModel object. This problem – of wanting to make incompatible objects work together – is common in object-oriented software development and can be solved using an established design pattern with which you are familiar. Apply this design pattern so that you can display a ShapeModel's shape composition using a JTree component.

Once you have completed this task, you can run the demo program bounce.views.TreeViewer. Assuming your Task1 class is correct, this program will display a ShapeModel's shape composition.

Hints:

- Examine the test cases implemented in the JUnit class, TestTask1, provided in package bounce.views. The tests will provide helpful insight in how you should implement class Task1.
- *Carefully* study the API documentation for Swing's TreeModel interface. You will spend less time on this task overall if you invest in some up-front activity to understand the TreeModel interface.
- Many of the TreeModel methods have arguments of type Object that you will need to cast to Shape and NestingShape. In implementing class Task1, be sure to use the instanceof operator so that only safe casts are performed.
- Class Task1 should define one constructor that takes as argument a reference to an object of type ShapeModel.
- TreeModel's valueForPathChanged() method can simply be implemented with an empty method body. This method is an event notification method that can safely be implemented like this in the context of Task 1.

Assessment criteria

- The design pattern intended to solve the interface compatibility problem has been applied appropriately.
- Class Task1 satisfies its specification, indicated by the supplied test cases passing.
- The TreeViewer application demonstrates correct functioning of your Task1 class.

Task Two: Implement class **bounce.views.Task2**

Your Task1 class has leveraged the Swing framework, and with very little effort you have a robust means of visualising a ShapeModel's shape composition. Awesome. However, the Task1 class suffers from a limitation that prevents the Bounce application from working correctly. Bounce is a model/view application, with the GUI showing three different views of a ShapeModel instance. The problem is that while the AnimationView and TableViewAdapter views respond to changes in the state of ShapeModel, an instance of the Task1 class does not. Consequently, when a new Shape instance is added or an existing Shape object is removed from the ShapeModel, the change is not reflected in the JTree component. Obviously, this is most undesirable as *all* views should offer consistent representations of a common model.

Introduce a new class, Task2, that extends your Task1 class such that a Task2 instance can both render a ShapeModel's shape composition *and* respond to changes that occur in the ShapeModel. Using a Task2 instance, its connected JTree component will update its display whenever the ShapeModel changes. It is helpful to think of a Task2 object as follows:

- A Task2 object plays **two** roles: the *model* of a JTree component, and the (non-visual) *view* (listener) of a ShapeModel.
- A Task2 object essentially transforms ShapeModel events that is hears about into TreeModel events that it fires to its TreeModelListeners. In the case of the Bounce application there is one such listener, the JTree component.

Implement class Task2 to do the necessary. Once completed you should have a functioning Bounce application. Once you have edited and deployed the application's properties file, discussed below, run the application and enjoy the fruits of your labour.

On startup the Bounce application attempts to read configuration information from the file named `bounce.properties`. You should locate this file within your project directory. You should ensure that the `shape_classes` property lists the fully qualified names of Shape subclasses that you want the application to use. Essentially, the combo box on the GUI is populated with the names of the classes you specify for the `shape_classes` property. Hence when using the application you can specify the kind of shape you want to add to the animation. In specifying class names for this property, each name should be separated by whitespace; note however that if names are spread across multiple lines a \ character should end each line except the last. For example, the following specifies that two classes should be loaded:

```
shape_classes = bounce.RectangleShape \
   bounce.NestingShape
```

Hints:

- As for Task 1, examine the test cases provided as they will provide useful insight in terms of how you should implement class Task2. Class bounce.views.TestTask2 implements the tests.
- *Carefully* study the API documentation for Swing's TreeModelListener interface and classes TreeModelEvent and TreePath. Similarly to before, you will spend *significantly* less time on this task if you study the API pages first rather than rushing in and trying to hack something together.
- There is a reasonable amount of complexity in the TreeModelListener and TreeModelEvent entities. Some of this complexity lies in the ability for a TreeModelEvent to describe multiple node additions/removals/changes in a TreeModel. Note that your Task2 class need only generate TreeModel events that describe a **single** addition or removal of a Shape at a time. Hence, the childIndices and children arrays that form part of the state of a TreeModelEvent should always have a length of 1.
- Each node shown in the Bounce application's JTree component represents a composite or simple shape instance within the ShapeModel object. Each node is labeled with the name of the Shape subclass that the node is an instance of. The JTree acquires the label value for each node by calling method toString on each Shape instance it discovers through making TreeModel calls on its model. Given that these label values are not subject to change in the Bounce application, your Task2 class need not make any treeNodesChanged calls on registered TreeModelListeners. In other words, your Task2 class has only to make treeNodesInserted and treeNodesRemoved calls on registered listeners, thereby communicating new and removed nodes respectively.
- Class Task2 should define one constructor that takes as argument a reference to an object of type ShapeModel.

Assessment criteria

- Class Task2 implements the required functionality in keeping with the application's model/view design.
- Class Task2 satisfies its specification, indicated by the supplied test cases passing.
- The Bounce application demonstrates correct functioning of your Task2 class.

Task Three: Add class **ImageShapeFormHandler**

The Bounce III application allows images to be loaded and added to the animation. Key classes that enable this functionality are:

- bounce.ImageRectangleShape. This is a Shape subclass that paints a given image.
- bounce.forms.ImageFormElement. An ImageFormElement allows the user to select a particular image from disk that is to be displayed by an ImageRectangleShape.
- bounce.forms.SimpleImageShapeFormHandler. An instance of SimpleImageShapeFormHandler processes data entered in form elements ShapeFormElement and ImageFormElement and uses this to instantiate an ImageRectangleShape object.

Class SimpleImageShapeFormHandler scales the image selected by a user based on the ShapeFormElement's width field value. ShapeFormElement's height is ignored as the scaling operation maintains the image's aspect ratio.

The problem with SimpleImageShapeFormHandler is that the loading and scaling operations – which are expensive for large images – are performed by the Event Dispatch thread. The image processing thus causes the application to freeze and become unresponsive when working with large images.

For this task, you should create a new class (bounce.forms.ImageShapeFormHandler) to more appropriately load and scale the image using a background thread, and make the new ImageRectangleShape instance available to the Event Dispatch thread. Like SimpleImageShapeFormHandler, ImageShapeFormHandler should have a constructor that takes ShapeModel and NestingShape parameters and should implement the FormHandler interface.

Once you have implemented ImageShapeFormHandler, you should edit the FormResolver class to change method getFormHandler() to return an ImageShapeFormHandler rather than a SimpleImageShapeFormHandler.

When testing your class – using the supplied bounce.forms.TestImageShapeFormHandler class – you should ensure that the supplied image file is in your project directory (where you stored the properties file earlier).

Hints:

The logic for acquiring form data, loading and scaling the image, and instantiating ImageRectangleShape is provided in the existing SimpleImageShapeFormHandler class. You can reuse this without modification in ImageShapeFormHandler. Don't consider this to be a case of duplicating code – in reality you would discard the naive SimpleImageShapeFormHandler implementation.

Assessment criteria

- The ImageShapeFormHandler class makes appropriate use of Swing's background processing facilities.
- The unit test bounce.forms.TestImageShapeFormHandler passes.
- The Bounce application remains responsive while simultaneously allowing the addition of ImageRectangleShape objects with large images to be added to the animation.