PGCert IT: Programming for Industry

Lab 16:More 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 15. It is important to finish Lab 15 and understand the given code before attempting the following exercises.

Exercise One: Adding Text to Shapes

In this exercise, we'll once again be extending the Bounce application. This time, we'll allow text to be displayed whenever a shape is painted. We will use the **template method** pattern to ensure that *any* subclass of `Shape`, including as-yet-unwritten ones, will be able to have text drawn.

In designing the revised class structure, you must **guarantee** that if a shape is associated with text 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.
When this exercise is complete, your program should produce output similar to Figure 1 below:

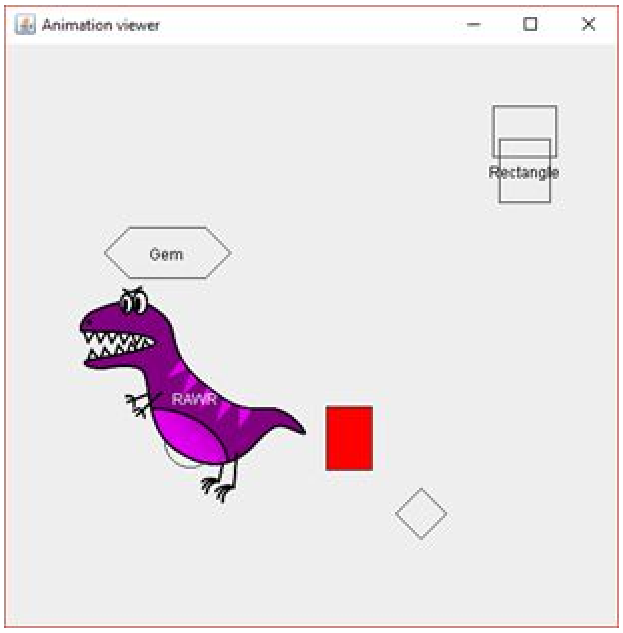


Figure 1: Screenshot showing text added to several different types of Shapes.

Constraints

1. 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. In particular, class `Shape` must continue to offer public methods move(int width, int height) and `paint(Painter painter)`.
2. The `AnimationViewer` class must not be modified in any way (unless you wish to add more shapes to it as a demonstration).

Step one: Add fields

Initially, you should add appropriate fields to the `Shape` class to store the text to be displayed, and the colour of that text. You should additionally add methods `setText(...)` and `setTextColor(...)` to allow those fields to be set.

Step two: Hook methods

Add a new abstract hook method into the `Shape` base class which will handle each individual Shapes' drawing logic. Name this method `paintShape(...)`. Modify all subclasses of `Shape` to override this method instead of the `paint(...)` method as they currently do.

Step three: Template method

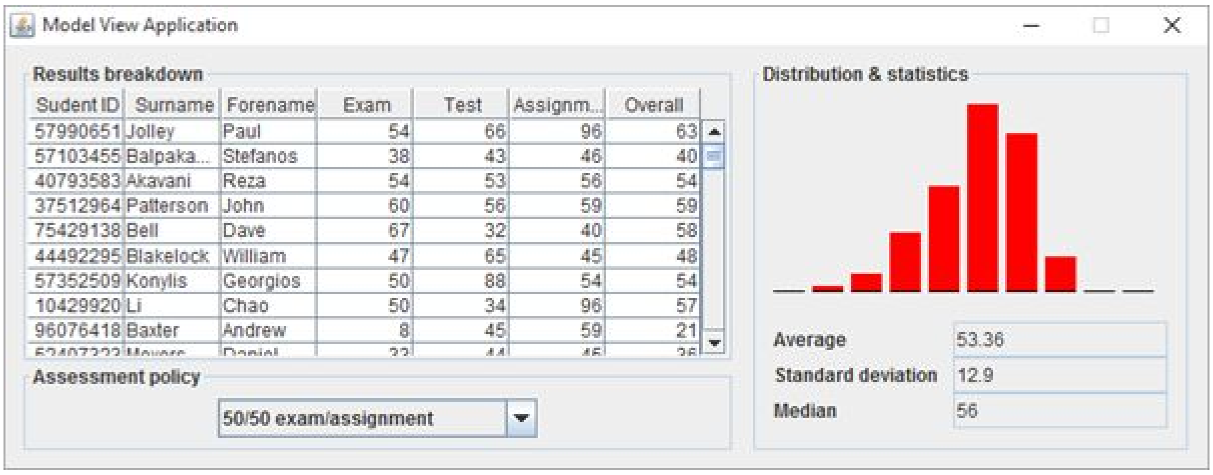
Remove the abstract `paint(...)` method declaration in the `Shape` class, and replace it with a concrete `paint(...)` implementation which does the following:

1. Draws the shape using the hook method you wrote in step two
2. Then, if the shape has text associated with it, draw the text in the appropriate colour. You may use `Painter`'s `drawCenteredText(...)` for this purpose.

Remember that this `paint(...)` method contains a method contract which we want to enforce. Make sure the method has been defined in such a way that subclasses cannot prevent its text-drawing functionality from being called.

Exercise Two: Modelling & Wiring a Model View Application

The objective of this exercise is to complete a Model View application that displays student results. We can see a screenshot of the result in Figure 2 below:



Sudent ID	Surname	Forename	Exam	Test	Assignm...	Overall
57990651	Jolley	Paul	54	66	96	63
57103455	Balpaka...	Stefanos	38	43	46	40
40793583	Akavani	Reza	54	53	56	54
37512964	Patterson	John	60	56	59	59
75429138	Bell	Dave	67	32	40	58
44492295	Blakelock	William	47	65	45	48
57352509	Konylis	Georgios	50	88	54	54
10429920	Li	Chao	50	34	96	57
96076418	Baxter	Andrew	8	45	59	21
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Assessment policy: 50/50 exam/assignment

Average	53.36
Standard deviation	12.9
Median	56

Figure 2: Screenshot showing a student results viewer app creating using MVC principles.

The GUI for the application comprises three views of student result data.

- A tabular view that identifies each student and displays their assessment breakdown in terms of exam, test, assignment and overall marks.
- A summary view in the form of a bar chart showing the distribution of marks for the course as a whole.
- An additional summary view showing key statistics: average, standard deviation and median scores for the course.

Underlying the views is a common model – represented as an instance of class `Course`. A `Course` comprises a `List` of `StudentResult` instances.

In addition to providing views, the application also provides a *control* component that allows the user to change to assessment policy used to calculate students' overall marks. The above screenshot shows that a 50/50 policy is currently in use; this policy gives equal weight to the exam and assignment components and does not take into the account a student's test score. Using the `JComboBox` control, the user can select from among several assessment policies. Importantly, whenever the policy is changed, all views must be updated to reflect the new policy. For the tabular view, the overall mark for each student should be recalculated and updated; similarly, the graph and statistics should be reproduced and displayed.

Source code layout

The source code for the application is available in package `ictgradschool.industry.lab16.ex02`, and organized in the following subpackages:

`admin`

Contains class `CourseDataManager`, which is used to generate and write `StudentResult` data to a file, and to read in the result data.

`gui`

This package contains the `Main` application program plus the view classes. `DistributionPanel` and `StatisticsPanel` represent the bar-chart and statistics views respectively. The tabular view is represented by a standard Swing `JTable` component.

`model`

The model package includes the classes that represent the application model: `Course` and `StudentResult`. A `Course` instance is essentially an aggregate of `StudentResult` objects. A `Course` additionally holds a reference to an `AssessmentPolicy` object. Each `StudentResult` object stores result data relating to an individual student. Finally, class `Course` allows `CourseListeners` to be registered with a `Course` instance. Whenever a `Course`'s state changes, the `Course` notifies any registered listeners. The `CourseListener` interface is defined in this package.

`policy`

This package contains an `AssessmentPolicy` interface and a number of implementations that differ in how they calculate a student's overall mark. In addition, this package contains class `AssessmentPolicyStore`, which essentially serves as an aggregate of `AssessmentPolicy` objects.

Completing the exercise

Completing this exercise involves writing adapter classes and wiring up the model and views via adapter instances. In particular, you should create three new adapter classes:

`DistributionPanelAdapter` in the `gui` package:

The role of this adapter is to allow the `DistributionPanel` view to be registered with the `Course` model and notified whenever the `Course` object changes its state (e.g. whenever the overall marks for its `StudentResult` objects change). The adapter should ensure that the `DistributionPanel` is repainted.

`StatisticsPanelAdapter` in the `gui` package:

This adapter has a similar role to `DistributionPanelAdapter`, but is concerned with allowing the `Course` model to notify the `StatisticsPanel` view when the `Course` changes.

`CourseAdapter` in the `model` package:

This class allows the `Course` model to be visualized using a regular Swing `JTable` component. Whenever the `Course` model changes, the adapter should ensure that the `JTable` view is updated.

In addition to implementing the above three classes, instances of the classes should be created and connected. You should change class `gui.Main` in its constructor – only where indicated in the source file. No further changes are necessary when conforming to the used patterns.

Constraints

You must **not** make any changes to the supplied classes, other than the three adapter classes and the `Main` class (where indicated in Main.java) mentioned above.

Hints

For this exercise, it is **essential** to develop a good conceptual model of the application and not get bogged down in the source code. Sketch a model (it need not be formal UML) to help you understand the key application classes in terms of models, adapters and views. Sketch how instances of the classes are to interact.

Make appropriate use of the supplied entities in keeping with the Model View and Adapter patterns that have been used in designing the application. The "thing of interest" in this application is the `Course` model. Note the existence of the `CourseListener` interface – any (adapter) that is interested in the changing state of a `Course` should implement this interface.

The implementation details of many of the classes are irrelevant. Such details can generally be ignored; as the first hint notes it is important to understand and work with the application's architecture, which, as noted above relies heavily on the Model View (Observer) and Adapter patterns.

Reflection

1. Why have the Model View and Adapter patterns been used in developing this application? What are the benefits of applying them in this context?
2. What is involved in adding an extra view, e.g. a view that generates an HTML representation of the data, or a view that writes mark updates to a file (note that "views" need not be visual)?
3. How much of the application has to be changed when adding new assessment policies?