Computer Science 230
Assignment 2: Extending Bounce

Deadline: 5th May

April 13, 2006

Acknowledgment: assignment adapted from work originally prepared by Rick Mugridge.

Introduction

This assignment aims to give you some experience with object-oriented programming, in particular class hierarchies, abstract classes, and refactoring. The assignment involves further developing the Bounce application introduced in lectures. Essentially, Bounce involves an animation comprising an extensible set of shape types. Shapes have in common knowledge of their position and velocity while each kind of special shape has a specific way of rendering itself.
You will need to investigate five *java.awt* classes: *Image*, *Graphics*, *FontMetrics*, *Color* and *Toolkit* to complete this assignment. Browsing the online JDK API should be sufficient for the needs of this assignment.
The resources for this assignment are available from my *Teaching* page `http://www.cs.auckland.ac.nz/~alexei/teaching/`.

Tasks

The assignment is broken into several tasks of varying complexity. Complete the work in the order of the tasks, using a Test-Driven Development approach. Estimate the time you expect to take for each task just before you start working on it, and record this. Also keep a record of how long you spend on each task (to the nearest 5 minutes). Notice whether your estimates improve due to this feedback.
Some of the tasks in this lab will require that you rethink the class hierarchy. The best approach is not always obvious – welcome to the real world of design.

Completion

To get the marks for this Lab, you need to have:

- Written good-quality unit tests for each task.
- Completed each task satisfactorily.
- Reported an estimate of time taken for each task.
- Tracked the time actually spent on each task.
- High-quality code with no smells.
- A good understanding of the assignment and your code.

- A good understanding of TDD and refactoring,
- Not changed the program beyond what was required.

You will need to hand in

- source code for you assignment, including TestCases and
- an executable jar file that runs your version of Bounce with a bunch of nice shapes
- a short report (one paragraph) outlining the estimated and actual times for each task and noting any important points about the design and development of the tasks.

Your mark will be 75% for implementation of tasks (15% each), but you will get no marks for tasks completed without adequate test cases. A further 10% of your marks will come from submitting a nice executable jar file of your completed Bounce program that entertains me. The final 15% will be for the short report.

1 Task One: Add an Image shape

Write unit tests for an *ImageShape*, a new type of *Sprite* which moves like the other shapes but displays itself as an image such as the one in Figure 1. Implement it and include such a shape in your animation.

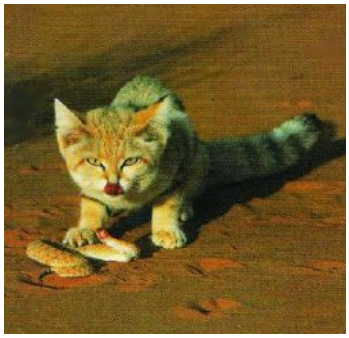


Figure 1: An image of the sand cat *Felis margarita*

Unlike the provided shapes, it is created with a specified image that will have a width and height. The arguments to the constructor are ordered as follows: *x, y, deltaX, deltaY, image*.
The *ImageShape* is to be painted to fill the size of its image, using *drawImage(Image image, int x, int y)* calls to the *Painter* (this method will need to be added to the *Painter* interface).

Hints:

- You need to reconsider the class *Shape* when you carry out this task.
- When developing the unit tests for *ImageShape*, you should include test cases to check that it bounces off the walls correctly. One way of writing these tests is to use a MockPainter and to check that its log contains the correct sequence of calls given the size of the image.
- You may need to use a *MediaTacker* or *IconImage* to force the image to be loaded in a timely fashion. Where should this code go?

2 Task Two A: Rectangle with Text

Write unit tests for a *RectangleTextShape*, a new type of *Sprite* which moves like the other shapes. It displays itself just as a *RectangleShape*, except that it displays the text, centred in the rectangle (the text may be larger than the rectangle).
It should be created with a specified width and height, In addition, it has an argument to the constructor for the text *String* to be displayed. The arguments to the constructor are ordered as follows: *x, y, deltaX, deltaY, width, height, text*.
To centre the text, you need to take account of the maximum height and width in pixels that the text will need to be displayed (using the default font, so calculate it dynamically). See the Java class *java.awt.FontMetrics* for details of how to calculate this. Rounded arithmetic is to be used to calculate the position of the text.
Implement it and include such a shape in your animation.

Hints:

- The usual way to get a *FontMetrics* is from a *Graphics*. But in this assignment, the *paint()* method of a *Shape* takes a *Painter* as argument (and it may be a *MockPainter*). So a *Graphics* is unavailable. To get around this, we'll implement the centering of the text inside the *Painter*.
- Add a method *drawCentredText(int x, int y, String text)* to the javaPainter interface. In *MockPainter*, simply implement the new method to log drawing of centered text. In *GraphicsPainter*, use the *drawString()* method of *java.awt.Graphics* to plot the text. Use the *java.awt.Graphics* object to get a *FontMetrics*. With a *FontMetrics* object, you can extract sufficient information to implement text centering.
- One way to calculate the Y coordinate to be passed to *drawString()* is as follows. Calculate the height of the text from the sum of the default font's ascent and descent. These values can be queried from the *FontMetrics* object. The Y coordinate can then be calculated as

  shape's Y coordinate - ((ascent + descent) / 2)

  However, where the ascent is greater than the descent, the Y coordinate needs to be increased by *ascent - descent* pixels. In other cases, where the descent is greater than the ascent, a similar adjustment needs to be made.
- The text is drawn after the rectangle is drawn.

3 Task Two B: Oval with Text

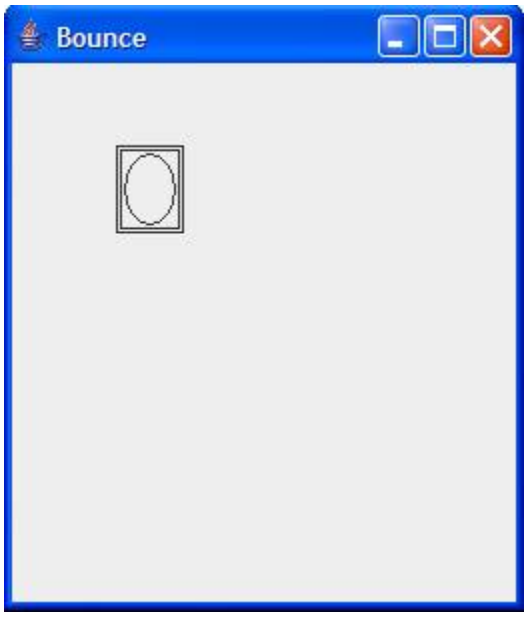
Write unit tests for a *OvalTextShape*, a new type of *Sprite* which moves like the other shapes. It displays itself just as a *OvalShape*, except that it displays the text, centred as with a *RectangleTextShape* (see section 2).
It is created with the same arguments to its constructor as *RectangleTextShape*. Implement it and include such a shape in your animation.

Hints:

This task seems simple after the previous one, but it raises a number of interesting design issues. You may want to consider refactoring the class hierarchy. You may also want to look at how *Template Methods* might feature in your revised hierarchy. As a result of refactoring, you may find that particular classes become redundant.

4 Task Three: Borders

Write unit tests for a *BorderShape*, a new type of *Shape* which moves like the other shapes. However, unlike the other *Shape*s, it contains a *Shape* and moves with that *Shape* (ie, its position is completely determined by the contained shape.
The BorderShape constructor should take a *Shape*, an integer line thickness, an integer gap between the border and the contained shape and a *Color* defining the color of the border.
It paints a rectangular border of the given thickness around its embedded *Shape*, with a user-defined gap between the internal shape and the border. The (rectangle) border is drawn after the shape is drawn.
Borders may be nested to arbitrary depth (ie, a *BorderShape* may be included inside another *BorderShape*). For example, the following screenshot shows a simple *BorderShape* which contains another *BorderShape* which itself contains a *OvalShape*.



As with any other Shape, a *BorderShape* is not permitted to move beyond the boundaries of its two dimensional world.
Implement class *BorderShape* and include instances in your animation.

Hints:

- A *BorderShape*'s properties are determined by the *Shape* it encloses. In particular, *x*, *y*, *width* and *height* are a function of the contained shape (and constructor parameters) and *deltaX* and *deltaY* are the same for the *BorderShape* and its part.
- To add support for color, introduce two new methods to the *Painter* interface: *getColor()* and *setColor()*. *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. After drawing in your preferred color, you can reset the current color to its original value.

5 Task Four: A new kind of motion

Up until now all of the shapes have shared the property that they bounce off of the walls and have a velocity that doesn't experience any friction or gravity. In this task you will write unit tests

for a new kind of motion *GravityMotion*. *GravityMotion* will implement a new interface *Motion* that will be taken by the *Shape* constructor instead of *deltaX* and *deltaY*. A class that implements *BouncingMotion* will need to be written to support the old behavior. The new *GravityMotion* class will have, in addition to *deltaX* and *deltaY*, an acceleration due to gravity, *g*. In the real world acceleration due to gravity is measured in metres/sec/sec, however in our virtual world it is easiest to measure gravity in pixels/clock/clock. This gravitational force will increase *deltaY* by *g* pixels each clock tick.

6 Task Five: BounceListener and changing shapes

Add a *BounceListener* interface that has a single method *bounceOccurred(Shape shape)*. The *Motion* interface should be updated to have *addBounceListener* and *removeBounceListener* methods. This new *BounceListener* interface can then be used to implement a shape that can change how it is rendered after it bounces off a wall. This exercise will probably involve a bit of thinking about exactly how to implement the changes in painting but the results are worthwhile. After this task you should be able to have an object that changes between a rectangle and an oval after each bounce off of a wall.