Exercise 9: Pizza Delivery and Building HTML Documents

TECHNISCHE UNIVERSITÄT DARMSTADT

Due Thursday, July 16, 23:59

Task 1 First Part: Order Management System

The word has spread about your successful implementation of the order management system (OMS) for Starbuzz Inc. You received an email from the manager of TastyPizza Inc. who heard about your success. He wants you to implement a new OMS for his pizza delivery shop. He has defined several requirements that need to be fulfilled by your OMS:

- The shop offers pizza, several beverages (lemonade, water and wine) as well as some franchise items (shirts, mugs). The system needs to be able to represent these products.
- All products have a price and pizza and beverages have specific nutrition facts.
- An order should maintain a list of ordered products with all their associated information.
- Implement a bill printer that uses the interface introduced in the last subtask to print a bill in a simple but readable format to a string. If the order contains alcohol, a notice for the delivery boy should be printed to verify the customer's age.
- Every pizza contains at least cheese and tomato sauce (Pizza Margherita). It can be combined with additional toppings (ham, onions, etc.), which can be ordered multiple times.
- Pizzas with certain topping combinations are named, e.g., Hawaiian Pizza for the base pizza with ham and pineapple. Those names should be contained in the bill. You do not need to auto-detect names for such combinations if they are not explicitly requested in the order.
- In addition to the normal size, pizza can be ordered in family size.

The manager sent you the menu card of TastyPizza for a better understanding of their available items:

Pizzas	**Calories**	**Price**
Pizza Margherita (tomato, cheese)	1104	4.99
Hawaiian Pizza (tomato, cheese, ham, pineapple)	1024	6.49
Salami Pizza (tomato, cheese, salami)	1160	5.99
Family Size for Pizza	x 1.95	+ 4.15

Toppings	**Calories**	**Price**
Cheese	92	0.69
Ham	35	0.99
Onions	22	0.69
Pineapple	24	0.79
Salami	86	0.99

Drinks	**Calories**	**Price**
Lemonade (0.33l)	128	1.29
Water (0.5l)	0	1.29
Wine (0.75l, 13%)	607	7.49

Franchise	**Price**
TastyPizza Shirt	21.99
TastyPizza Mug	4.99

Your tasks for this exercise are as follows:

- Implement an OMS library in Scala that can be used to manage orders for TastyPizza, Inc. You do not need to implement a frontend, since your library should be integrated in an existing cashing software. Provide tests to document the usage of your library. Provide at least 3 test cases that reflect the workflow of an order and print a bill to the console.
- Your library design should use immutable objects, i.e., orders and pizzas are never changed in-place. For example, adding a topping to a pizza creates a new pizza object.
- Use Scala objects instead of classes where possible.
- Use the Decorator Pattern to implement the different variations of pizza (additional toppings, family sized).
- Use the Visitor Pattern to implement the bill printer. Use a flexible variant of the visitor with two methods per product: `def startVisit(...)` and `def endVisit(...)`. It would visit some objects twice, for example, a pizza once via `startVisit()`, then all its toppings, then the same pizza again via `endVisit()`. This will make certain printing tasks easier.
- Create a UML class diagram that illustrates your design. If you are not familiar with a UML design tool, you can scan a (clean!) hand drawn diagram.

Task 2 Extensions?

Is your retail system open for extensions? In particular, is it possible for clients to add items to the menu or extend the system with additional operations? Does the Visitor Pattern variant discussed on Slides 6.5 help with the extensibility issues you discovered and if so how? Elaborate in less than 150 words.
Submit your UML diagram and discussion as a PDF.

Second Part: Building HTML Documents

In the second exercise, you will develop a small library to build simple HTML documents. You will use different design patterns and build documents that render images either as embedded PNGs, as linked PNGs, or in HTML 5 canvas elements. In the project template, you will find partial implementations (they will not compile) and a test application that creates a few test pages. Your implementation must adhere to the interface used in the given `HTMLTests` object, i.e., once you have implemented your library, `HTMLTests` will compile and open a few generated pages.

Task 3 Source Code Builder

Eventually we want to emit correctly indented HTML and Javascript code. Implement a builder `SourceCodeBuilder` that helps you do that. You will find a template in the project that you simply need to complete.
The builder maintains a current level of indentation, which can be changed with methods `indent()` and `unindent()`. Note that the builder needs to recognise when a new line is started in order to correctly indent the code. A new line is started either by a call to `newline()` or by appending a string ending in `'\n'`.
Most methods in `SourceCodeBuilder` have return type **`this.type`**, which simply means you have to return **`this`** at the end of each method. This will allow you to do method call chaining such as in:

```
val age = 99
val b = new SourceCodeBuilder("␣␣")
b += "Hi,␣I␣am␣" += age += "years␣old."
```

Use a `StringBuilder` to implement `SourceCodeBuilder`, which is more efficient than composing strings.

Task 4 Document Builder

Implement a builder that lets clients gradually build HTML documents. Your implementation needs to be able to emit HTML 5 as well as HTML 4 documents. Moreover, you should be able to generate single HTML documents and documents that are split across multiple files.
Use the bridge pattern to implement these two axes of variability. When using the bridge pattern, one has two class hierarchies, the abstraction and the implementation hierarchy. In this case, the implementation side should care about emitting code for different HTML versions. The abstraction side cares about how to logically arrange the HTML document. You will find the base traits of both sides in `DocumentsBuilders` and `HTMLBuilders`.
For this task, you need to do two things:

- Create two subclasses for `HTMLBuilder` as indicated in the project template. The only difference between them for now is that they emit different DOCTYPE declarations (DTD). For HTML 5 it should simply be `<!DOCTYPE html>`. See `http://www.w3schools.com/tags/tag_doctype.asp` for the HTML 4.01 Strict DTD. Use your `SourceCodeBuilder` for the implementation side.

- Create two subclasses for `DocumentBuilder` as indicated in the project template. The `SimpleDocumentBuilder` will emit HTML is a straightforward way, eventually creating a simple string containing a valid HTML document including a DTD, head and body elements. The `PaginatedDocumentBuilder` will create a new HTML document each time a client adds a `<h1>` headline. The new document will again be a valid HTML document including a DTD, head and body elements, where the body will contain the headline and everything that follows until the next `<h1>` headline. The result of `PaginatedDocumentBuilder` should be a `Seq[String]` and it should use a (fresh) `SimpleDocumentBuilder` to build each page.

Task 5 Image Hierarchy

We now want to be able to add images to HTML documents. In order to do so, we first need a representation of them. Since images can be either embedded into HTML (in Data URIs using Base64 encoding[1]), or linked via a URL, we want to represent these two alternatives. It is important to notice that in the latter case, we don't need to load the image data into memory.
Therefore, develop a minimal image hierarchy to represent pixel images. In `ex09.part2.Images`, you will find the base trait of the hierarchy, which is as follows (excluding comments):

```
trait Image {
  def width: Int
  def height: Int
  def toPNG: Array[Byte]
}
```

The only way to obtain image data is via method `toPNG`, which returns the image in the PNG format. Create the following two subclasses:

- Use the adapter pattern to adapt a `java.awt.image.BufferedImage` to your `Image` trait. You can obtain a PNG byte array from a `BufferedImage image` as follows:

  ```
  val out = new ByteArrayOutputStream
  ImageIO.write(image, "png", out)
  out.toByteArray()
  ```

- Use the proxy pattern to represent an image at a given URL. The proxy gets the width and height as constructor arguments but the image data is loaded from the URL (using `ImageIO.read()`) only when needed. You will find utility method `Image.resizeImage()` to scale a given image.

Create factory methods in the companion object of the `Image` trait to create images from a `java.io.URL` (using your proxy) and a `java.io.File` (using your adapter and `ImageIO.read()`).

Task 6 HTML Images

Add a method `image(ing: Image)` to `DocumentsBuilder` and its implementations which adds an image element to the document under construction. It should add a linked image as in `<img src="http://upload.wikimedia.org/wikipedia/en/7/78/Design_Patterns_cover.jpg">` if the given image is an instance of your image proxy. Otherwise, it should use a Data URI with base64 encoding. You will find a utility method in the project template to convert PNG byte arrays to base64 strings.

Task 7 Graphics

We now want to be able to add procedurally rendered images into an HTML document, i.e., draw lines and rectangles into an HTML 5 canvas element or fall back to pre rendered PNG images for HTML 4. For this purpose, we need a canvas to draw on. This canvas is in fact again an instance of the builder pattern.
Your tasks are:

[1] See the HTML example on `http://en.wikipedia.org/wiki/Data_URI_scheme#Examples`

- You will find the base class `Canvas` of the canvas builder in the project template. Implement two subclasses, `HTML5Canvas` and `PNGCanvas`. Class `HTML5Canvas` should eventually append HTML as follows:

```
<canvas id="canvas1" width="300" height="200"></canvas>
<script type="text/javascript">
  function draw_canvas1() {
    var canvas = document.getElementById('canvas1');
    if (canvas.getContext) {
      var c = canvas.getContext('2d');
      c.fillStyle = 'rgba(255,0,0,0.5019607843137255)';
      c.fillRect(10,10,50,50);
      c.fillStyle = 'rgba(0,0,255,0.5019607843137255)';
      c.fillRect(30,30,50,50);
    };
  };
  draw_canvas1();
</script>
```

  Class `PNGCanvas` should render the image into an in-memory `BufferedImage` and add a Data URI with base64 encoding.

- Add a method `canvas(width: Int, height: Int)(c: Canvas => Unit)` to your document builders. It accepts a width and height of the canvas and a function that takes a handle to the canvas to construct. This function can then draw shapes to the canvas, e.g.,:

```
builder.canvas(400, 300) { c =>
  c.setColor(Color.Blue)
  c.fillRect(50, 50, 100, 100)
  c.setColor(Color.White)
  c.drawLine(10, 10, 140, 140)
}
```

- For the previous point, you also need to modify your HTML builders to choose the right canvas instance.

The code template already contains a few utility methods and partial implementations for this task.

Task 8 Adapter Pattern?

Do traits help implementing the adapter design pattern? Please leave a comment in the HTML Builder Document implementation to answer this question!

Solve it on your own!

Although this exercise is not graded, it is highly recommended to do it by yourself. Just looking at a solution is much easier in comparison to actually coming up with it.

Requirements if you want to submit your solution

You can, once in the semester, submit your solution to get it corrected. Send your solution to `weiel@st.informatik.tu-darmstadt.de`. Make sure you zip your complete sbt project and make sure that is out of the box working by running `sbt run` and `sbt test`. If the project doesn't compile properly you will not receive any feedback!