Aix*Marseille université
Initiative d'excellence
Année 2018-19

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : Corrigé du TD 7
Code UE : SIN3U02

Images

1 Exercices

1. L'interface `Image` pourrait contenir les méthodes suivantes :

```
import javafx.scene.paint.Color;

public interface Image {
    Color getPixelColor(int x, int y);
    int getWidth();
    int getHeight();
}
```

   On pourrait aussi rajouter un nom à l'image avec un getter (`String getName()`) ou encore une méthode pour avoir le rapport de forme (`double getAspectRatio()`) renvoyant le rapport entre la largeur et la hauteur de l'image.

2.
```
public class BruteRasterImage implements Image {
    private final int width;
    private final int height;

    private final Color[][] pixels;

    public BruteRasterImage(Color[][] pixels) {
        width = pixels.length;
        height = pixels.length[0];
        this.pixels = pixels.clone();
    }

    @Override
    public Color getPixelColor(int x, int y) {
        return pixels[x][y];
    }

    @Override
    public int getWidth() {
        return width;
    }

    @Override
    public int getHeight() {
        return height;
    }
}
```

3. Le plus propre pour tester cela est de créer une méthode dédiée pour tester qu'une matrice n'a pas des dimensions nulles.

```java
/**
 * Ensures that the given matrix (assumed to be rectangular) does not have zero rows or zero
 *
 * @throws IllegalArgumentException if the matrix have zero rows or zero columns.
 * @param matrix the matrix to be tested.
 */
public static void requiresNonZeroDimensions(Object[][] matrix) {
    if (getRowCount(matrix) == 0) {
        throw new IllegalArgumentException("The matrix must not have zero rows.");
    }
    if (getColumnCount(matrix) == 0) {
        throw new IllegalArgumentException("The matrix must not have zero columns.");
    }
}

/**
 * Give the number of rows of a matrix.
 *
 * @param matrix the matrix.
 * @return the number of rows of the matrix.
 */
public static int getRowCount(Object[][] matrix){
    return matrix.length;
}

/**
 * Give the number of columns of a matrix (assumed to be rectangular).
 *
 * @param matrix the matrix.
 * @return the number of rows of the matrix.
 */
public static int getColumnCount(Object[][] matrix){
    return matrix[0].length;
}
```

4. Là aussi, le plus simple est de créer un méthode dédiée :

```java
    /**
 * Ensures that the given matrix is rectangular, i.e., all rows have the same size.
 *
 * @throws IllegalArgumentException if the matrix have rows with different sizes.
 * @param matrix the matrix to be tested.
 */
public  static void requiresRectangularMatrix(Object[][] matrix) {
    for (int x = 1; x < getRowCount(matrix); x++) {
        if (matrix[x].length != matrix[0].length)
            throw new IllegalArgumentException("The matrix must be rectangular.");
    }
}
```

On peut donc changer le constructeur en :

```
public BruteRasterImage(Color[][] pixels) {
        requiresNonZeroDimensions(pixels);
        requiresRectangularMatrix(pixels);
        width = pixels.length;
        height = pixels.length[0];
        this.pixels = pixels.clone();
}
```

5. Pour que l'utilisateur puisse différencier les erreurs, il suffit de mettre des messages différents expliquant l'erreur comme cela a été fait dans le code ci-dessus.

6. ```
   package image;
   import javafx.scene.paint.Color;
   import java.util.List;

   public class PaletteRasterImage implements Image{
       List<Color> palette;
       int[][] indexesOfColors;
       private final int width;
       private final int height;

       @Override
       public Color getPixelColor(int x, int y) {
           return palette.get(matrix[x][y]);
       }

       public PaletteRasterImage(List<Color> palette,
                                 int[][] indexesOfColors) {
           this.indexesOfColors = indexesOfColors;
           this.palette = palette;
       }

       @Override
       public int getWidth() {
           return width;
       }

       @Override
       public int getHeight() {
           return height;
       }
   }
   ```

7. Il y a du code dupliqué au niveau de la hauteur et la largeur mais aussi au niveau de la matrice : les deux classes contiennent une matrice.

8. Cela donne les classes suivantes :

   ```
   package image;

   public abstract class AbstractImage<E> extends AbstractImage
           implements Image {
   ```
   ```

```
    E[][] matrix;

    public AbstractRasterImage(E[][] matrix) {
        super(matrix.length, matrix[0].length);
        this.matrix = matrix;
    }
}
```

```
package image;
import javafx.scene.paint.Color;

public class BruteRasterImage extends AbstractImage<Color>
        implements Image {

    public BruteRasterImage(Color[][] matrix) {
        super(matrix);
    }

    public Color getPixelColor(int x, int y) {
        return this.matrix[x][y];
    }
}
```

```
package image;
import javafx.scene.paint.Color;
import java.util.List;

public class PaletteRasterImage
        extends AbstractImage<Integer> {
    List<Color> palette;

    @Override
    public Color getPixelColor(int x, int y) {
        return palette.get(matrix[x][y]);
    }

    public PaletteRasterImage(List<Color> palette,
                              Integer[][] indexesOfColors) {
        super(indexesOfColors);
        this.palette = palette;
    }
}
```

9. Afin de factoriser le code, on crée une nouvelle classe `AbstractImage` qui regroupe les parties communes entre les images : essentiellement le fait d'avoir une hauteur et une largeur.

```
package image;

public abstract class AbstractImage implements Image{
    int width;
    int height;
```

```
    public AbstractImage(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public int getWidth() {
        return width;
    }

    @Override
    public int getHeight() {
        return height;
}
```

On remplace l'ancienne classe `AbstractImage` par la classe suivante :

```
package image;
public abstract class AbstractRasterImage<E> extends AbstractImage
        implements Image {

    E[][] matrix;

    public AbstractRasterImage(E[][] matrix) {
        super(matrix.length, matrix[0].length);
        this.matrix = matrix;
    }

}
```

Pour la classe `SparseImage`, cela nous donne la classe suivante :

```
package image;
import javafx.scene.paint.Color;
import java.util.List;

public class SparseImage extends AbstractImage {
    List<Pixel> nonWhitePixels;

    @Override
    public Color getPixelColor(int x, int y) {
        Point point = new Point(x,y);
        int indexOfPixel = nonWhitePixels.indexOf(point);
        if(indexOfPixel<0)
            return Color.WHITE;
        return nonWhitePixels.get(indexOfPixel).getColor();
    }

    public SparseImageTemp(int width, int height,
                           List<Pixel> nonWhitePixels) {
        super(width, height);
        this.nonWhitePixels = nonWhitePixels.clone();
```

```
    }
}
```

10.
```
package image;
import javafx.scene.paint.Color;
import java.util.List;

public class VectorImage extends AbstractImage {
    List<Shape> shapes;

    public VectorImage(List<Shape> shapes, int width, int height) {
        super(width, height);
        this.shapes = shapes.clone();
    }

    @Override
    public Color getPixelColor(int x, int y) {
        for(Shape shape : shapes){
            if(shape.contains(new Point(x,y)))
                return shape.getFillColor();
        }
        return Color.WHITE;
    }
}
```

11.
```
package image;
import javafx.scene.paint.Color;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SparseImage extends AbstractImage{
    Map<Point, Color> pointColorMap;

    public SparseImage(List<Pixel> nonWhitePixels,
                       int width, int height) {
        super(width, height);
        pointColorMap = new HashMap<>();
        for(Pixel nonWhitePixel : nonWhitePixels){
            pointColorMap.put(nonWhitePixel,
                    nonWhitePixel.getColor());
        }
    }

    @Override
    public Color getPixelColor(int x, int y) {
        return pointColorMap.getOrDefault(new Point(x,y),
                Color.WHITE);
    }
}
```