CSI-5-OOP Object Oriented Programming Coursework 2

Student Number:	4008609

Task 1 – Write an analysis of classes in the *shapes* package

The shapes package contains a vital class, **'CompoundShape'** which holds 2 instance variables: 'name' of type string, and a list of 'shapes' of type PaintableShape[]. This class is needed for shapes which are combined (compounded) for example a shape oval with shape text inside of it, upon further reflection, I personally believe this class is a vital implementation as it will save programmers time in the future when wanting to combine further shapes, meanwhile maintaining the structure of the project.

The 'CompoundShape' class also contains 9 instance methods, all of which are implemented from interface class 'PaintableShape'. Hence, the need for overriding in other classes where these methods are used.

These methods include:

- CompoundShape() – Constructor used in the compound package for classes 'BoundedOval' (combination of OpenOval and FilledOval (classes) and 'TextOval' (combination of Filled Oval, Open oval and Text Shape)

- getName() – Getter for 'showConfigurationDialog()' method in class 'ImagePane' and 'getUI()' method in class 'CompoundShape'

- setPosition(point : Point) – Takes 1 parameter (Point object) and loops through an array of PaintableShape objects named shapes. For each PaintableShape in the shapes array, the method calls setPosition() on the said object and passes the Point object as an argument.

- getPosition() - returns the position of the first object in the shapes array.

- setSize(dimension: Dimension) - takes a single parameter, a dimension object. loops over the list of shapes and calls the setSize method on each shape, it also passes the dimension parameter as an argument.

- getSize() - returns the size of the first shape in the shapes array. This is used in the future to call the dimensions of the shape since the width and height are stored as separate variables.

- getBounds() – Returns a rectangle object.

- paint(g : Graphics2D) - Loops through the list of shapes and calls the paint method of each PaintableShape object. It also passes the Graphics2D object g as an argument. The class Graphics2D is a subclass of the class Graphics, which provides basic graphics in java.

- getUI() - returns object type 'ShapeConfigurationUI' as object ui. It also loops through the list of shapes and calls getName() and getUI() on the shape.

The 'compound' package contains **classes 'TextOval' and 'BoundedOval'** which are subclasses of the class 'CompoundShape'. They both extend/inherit class 'CompoundShape'.

The **TextOval class** contains a constructor (we know this because the name of the method is the name of the class, also, the method does not have the keyword void and still does not return anything), which can create 3 other objects (FilledOval, OpenOval and TextShape). 'Imagepane' is initialised as type JComponent. These 3 objects are being passed the 'imagepane' object. This method also sets the transparency (property) of the 'FilledOval' to 0.5 by default. NOTE: The keyword super is used since this invokes a superclass constructor which is needed since this is a subclass.

The **'BoundedOval' class** contains a constructor which allows the initialisation of the bounded oval shape, along with a filled and open oval.

The 'CompoundShape' class implements and aggregates the interface **class 'PaintableShape'** (AbstractPaintableShape class also implements the paintable shape class but does not aggregate). An interface class is a type of abstract class meaning it cannot create objects, hence not needing constructors. As seen in the PaintableShape class, an interface class contains methods, however, said methods do not contain a body, the class implementing must provide the body of the method, for example, class CompoundShape (see above). As a result, the implementation class must override the method, as seen in the 'CompoundShape', 'ImagePane', 'FilledOval', etc. classes. The 'PaintableShapes' interface class is needed as it provides a high level of abstraction for the slightly complex program. Upon reflection, this class is a great implementation idea, as it will allow programmers in the future (even those not fully familiar with the structure) to use these bodyless methods to implement additional methods for other shapes or functionality.

The 'PaintableShape' class includes the 9 instance methods stated on pages 1-2 (for class CompoundShape) (excluding the CompoundShape constructor), but without a body (since it's an interface class).

The **'AbstractPaintableShapes' class** implements the methods from class 'PaintableShape'. This class initialises instance variables including imagePane of type JComponent, a name of type string, position of type Point (this value is updated when the shape is moved to a different position on the gui) and size of type Dimension.

This class also initialises a tree map object with the field "properties", this maps PropertyKey objects. This also stores key-value pairs in a tree structure in ascending order. The "final" keyword shows that the properties field cannot be reassigned in the future to reference a different object.

It also has methods for interacting with shapes such as:

- getName() – Returns value of field ‘name’
- getPosition () – Returns the value of the position field with type Point
- setPosition() – Can be used to set the position of the shape with type Point
- getSize() – Returns the size with type Dimension
- setSize() – Can be used in the future to set the size of the shape with type Dimension
- getBounds() – Returns a rectangle object with X and Y coordinates for the position, along with the width, height (size)
- getPropertyKeys() - returns a Set of PropertyKey objects
- getPropertyValue() - returns the value for a given key in the "properties" map. It also takes a single argument, a "final" PropertyKey object named "key" and returns an object.
- setPropertyValue() – Takes property key object and, key and a value object. It sets the value of the key to the properties map to a certain value

The **AbstractColouredPaintableShape class** is a subclass of and extends class ‘AbstractPaintableShape’. This class is needed for shapes that have a colour and a transparency property. This class could also be subclassed, as is in this program (discussed below) to create other coloured shapes such as squares. 2 fields are defined, the colour key and the transparency key, this creates an object of type propertyKey. It is worth noting that since these are static fields, they can be used using the class name rather than having to create an object in the future.

This class also contains a constructor for parameters name and imagepane. The constructor also calls a setPropertyValue() method which sets the value of the transparency (set to 0 by default) and colour properties (set to white by default). The keyword super is used since this constructor is calling the constructor from class AbstractPaintableShapes. This is how the name and imagePane parameters are passed to the superclass constructor.

The AbstractColouredPaintableShape class also contains a method called getColor() which returns the colour object. The colorKey field is also passed as a parameter to the getPropertyValue method.

The **‘AbstractLineBasedPaintableShape’** class is an abstract class that extends/inherits and is a subclass of class AbstractColoredPaintableShape. This class is needed to allow the user to set the line thickness property of the open and bounded oval shapes, as well as maybe the rectangle in the future. This class provides a way to encapsulate the functionality and properties common to these shapes as well as allowing subclasses to define their own specific appearance/behaviour. Since this is an abstract class, it is used as a blueprint for other classes to extend it in the future. This method defines a static lineThicknessKey of type string (PropertyKey) and contains a constructor which first calls the constructor from class AbstractColouredPaintableShape and then sets the value of the thickness to 10 by default.

The ‘simple’ package contains 4 classes, OpenOval, TextShape, FilledOval and ImageShape.

The **OpenOval class** inherits, extends and is a subclass of the class AbstractLineBasedPaintableShape. The OpenOval Class first invokes a constructor inherited from class AbstractLineBasedPaintableShape (hence the use of ‘super) which allows the OpenOval object to be created. It also holds a paint() method which is inherited and overwritten from the interface class ‘PaintableShape’ which retrieves the size, thickness and transparency variables, it is worth noting the thickness is of type float, giving more precision to the user. The method also creates a ‘BasicStroke’ object using the thickness variable.

The **TextShape class** is inherited from and is a subclass of class AbstractColoredPaintableShape. This class is needed because it allows the program to draw text on an image pane with a specified font, colour, text content and transparency. The class first creates a static field of 2 property keys (text and font) which can be used to map keys to properties. The class then defines a constructor which takes parameter JComponent (from swing library) and names it imagepane. It then initialises a TextShape object by using a constructor from the parent class (AbstractColoredPaintableShape). The class then sets the textKey property to “hello world” and fontKey to SANS-SERIF-PLAIN-32 by default. It is worth noting that this class does not extend the AbstractLineBasedPaintableShape class as the program does not support the ability for the user to set the thickness of the font. This could however be changed for future use.

The **FilledOval class**, also extends class AbstractColoredPaintableShape (not LineBased since there is no thickness for a filled oval). This class is needed to allow the user to draw a filled oval on the GUI as it provides properties such as colour, size and transparency. The class also has a constructor for the filled oval. It also contains the method paint() which overrides the method in class PaintableShape, this method is used to draw the oval, as well as setting the colour and getting the size from the user’s input.

The **ImageShape class** is a subclass of and inherited from class AbstractPaintableShape, this is not the same as the above-stated (extension of AbstractColoredPaintableShape) since an image does not have a settable colour, but has colours of its own which cannot be changed. The class first initialises a static field

(imagekey) which is used to store the image file, the class also initialises an imageFilePath of type File, and an image of type BufferedImage. The constructor provided in this class calls the setPropertyValue() method on the image key and image file path, this method is also called again on the transparency key. The class also provides a method, paint() which passes the Graphics2D g parameter, the method is used to draw the image on the GUI. The paint() method first checks if the image is “null” or whether the file path is different to that of the image key. If successful, the image is stored in the image field and the file path field is updated.

Task 2 – Describe the execution sequence of *addShape* method

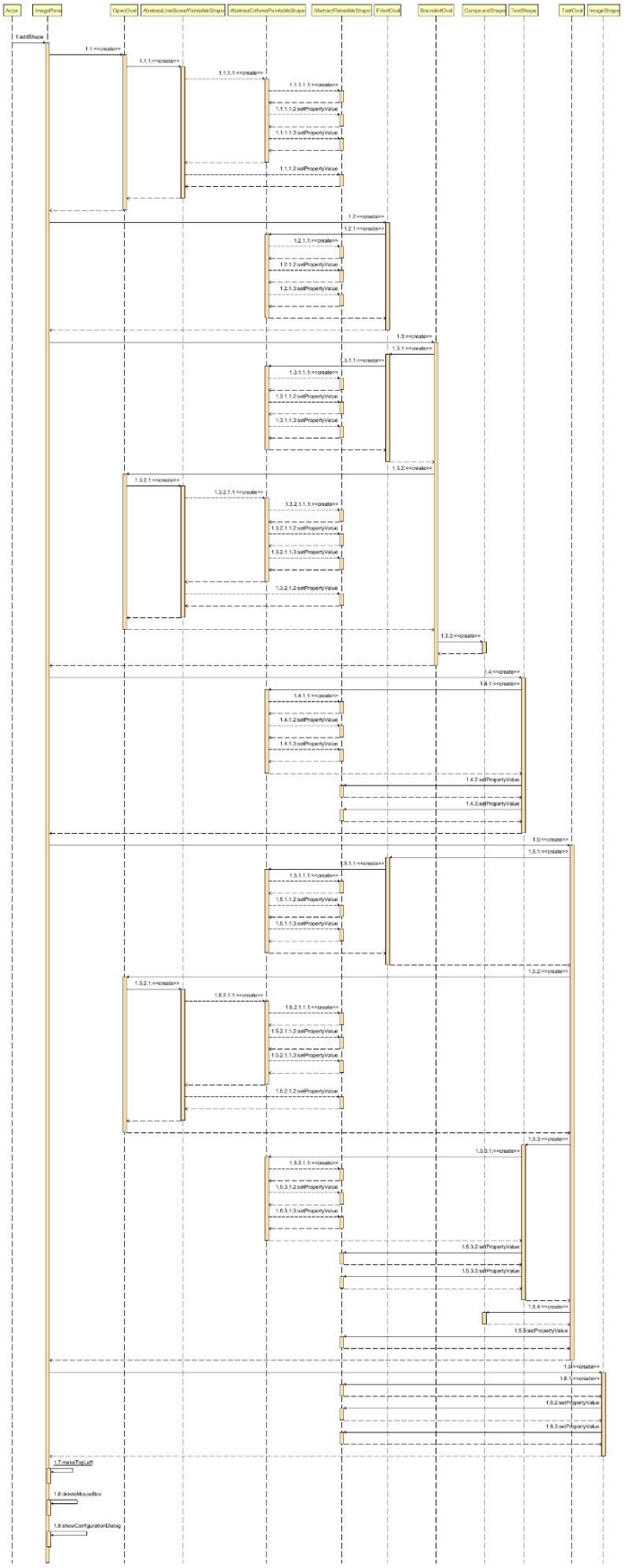


Figure 1: Sequence diagram for addshape method

The sequence diagram for the **addshape()** method (see figure 1) has been generated using the SequenceDiagram plugin in IntelliJ.

The addShape method in ImagePane is invoked when the mouse button is released AFTER dragging it out (Child.M 2022).

After the rectangle is drawn and mouse button released, the method first sets the shape variable (type PaintableShape) to null. This is then followed by conditionals which check what the current shape is that the user has selected, this is done by comparing the strings, the method then creates the respective object. The objects involved in this conditional sequence are 'OpenOval, 'FilledOval', 'BoundedOval', 'Text', 'TextOval', and 'image'.

Once the **OpenOval** object is created it is proceeded to the AbstractLineBasedPaintableShape class which is responsible for the line thickness, then the AbstractColouredPaintableShape class, and finally the AbstractPaintableShape class where the colour and thickness properties are set.

If the shape **filled oval** is selected and drawn is the method directly invokes the filled oval class to create the object which is, then followed by the AbstractColouredPaintable shape class which processed (for a short period (3 times)) to the AbstractPaintableShape class to create and set the property values, this is shown using the alt boxes that indicate conditions alternatives using the horizontal dashed lines, along with their stated conditions. Note that for the filled oval, the AbstractLineBasedPaintableShape class is not used since the filled oval does not have a thickness.

When the **Bounded Oval** is selected and drawn, it uses the 'Filled oval' class to create a filled oval object, and THEN it proceeds to the Open Oval class (follows the same sequence states above for open oval) (to create an open oval object), and finally the CompoundShape class is used to combine the two shapes.

The **TextShape** shape utilises classes AbstractColouredPaintableShape, which sets the colour properties of the text, along with the AbstractPaintableShape class which implements methods from class PaintableShape. The AbstractPaintableShape class is used again (later on) to set the property values of the font and text contents (i.e., hello world).

If the user selects and draws the Text Oval shape, the ImagePane first proceeds to the TextOval class to create the text oval object, it then creates a filled oval object using the constructor, and classes AbstractColoredPaintableShape and AbstractPaintableShape to set the transparency and colour. The sequence then shortly returns to TextShape before creating an open oval object, first using the open oval class then, AbstractLineBasedPaintableShape then the AbstractColoredPaintable shape which is inherited from AbstractPaintableShape to set the colour and transparency. The sequence then shortly returns to class AbstractLikeBasedPaintable shape to set the line thickness with help from the AbstractPaintableShape class. This is then followed by the creation of the text object using the TextShape class, this is created with the help of classes AbstractColoredPaintableShape and the AbstractPaintableShape, which sets the property values of the text transparency and colour. The sequence is then again (shortly) moved back to the TextShape class before proceeding to the AbstractPaintableShape class to set the font and text content (i.e., hello world). The property values are set using the constructor in class Text Oval. The 3 shapes are then compounded (combined) using the CompoundShape class, and finally, the property values are “put” into the map using the AbstractPaintableShape class.

If the user selects the **Image** option, the method will use the superclass constructor in class **ImageShape** will be used to create the object, then using the AbstractPaintableShape class, the properties for the image path and transparency (0.0 by default) will be set by passing the respective keys.

The addshape() method, after attempting to create a shape object, will check whether the check the user has selected a shape (and it is not null) and will provide the user with a draggable box (starting from the top left). This is done using the **makeTopLeft()** method which is initialized and defined in the same (image pane) class. This MakeTopLeft() method is responsible for recording (in relative variables) and returning the dimension of the rectangle (template for the shape i.e., circle) as well as its coordinates. This method passes the mousebox object first initialised in the class. The method also does not contain any sub-methods

The addshape() method will then set the position and size of the shape using the setPosition and setSize methods initalised in the PaintableShape class.

The shape will then be added to the painteddshape array list using the **add(shape)** method. The add() method is defined in the List.java class provided by the util.ArrayList library.

The mousebox is then deleted using the **deleteMouseBox()** method (again defined in the same class) by setting all variables (dimensions and coordinates) to 0. This does not contain any sub-methods.

The addshape() method then invokes the **repaint()** method, which is defined in the AbstractPaintableShape class, however, is derived from the Apt package in the Component.Java class provided in java. See figure 2 below)

```
public void repaint(long tm, int x, int y, int width, int height) {
    if (this.peer instanceof LightweightPeer) {
        // Needs to be translated to parent coordinates since
        // a parent native container provides the actual repaint
        // services.  Additionally, the request is restricted to
        // the bounds of the component.
        if (parent != null) {
            if (x < 0) {
                width += x;
                x = 0;
            }
            if (y < 0) {
                height += y;
                y = 0;
            }

            int pwidth = (width > this.width) ? this.width : width;
            int pheight = (height > this.height) ? this.height : height;

            if (pwidth <= 0 || pheight <= 0) {
                return;
            }

            int px = this.x + x;
            int py = this.y + y;
            parent.repaint(tm, px, py, pwidth, pheight);
        }
    } else {
        if (isVisible() && (this.peer != null) &&
            (width > 0) && (height > 0)) {
            PaintEvent e = new PaintEvent( source: this, PaintEvent.UPDATE,
                                           new Rectangle(x, y, width, height));
            SunToolkit.postEvent(SunToolkit.targetToAppContext( o: this), e);
        }
    }
}
```

Figure 2: repaint() method derivation

Since the repaint() method (see figure 2) is in the Component.java class, upon research (Oracle 2022) states that a component is an object providing a graphical representation which could be displayed on a screen and is interactable by the user. These can involve scroll bars, buttons etc. The repaint() method is responsible for drawing and outputting the respective shape onto the screen. As expected, this is an asynchronous method of applet as verified by (ebucba 2022).

Finally, the **showConfigurationDialog()** method is called, this method is responsible for displaying the window which allows the user to set the various properties for the shape (see figure 3 below). This was tested, after evaluating the method, by deleting the method initialisation (in class imagepane) and commenting out the code lines where this method was called (in the ImagePane and PaintOn class, see figure 4 below), as expected the shape was drawable however the property keys could not be set as the program did not display the dialogue window. Due to limitations of personal knowledge I was unable to determine the difference between the dialogue window and the configuration UI.

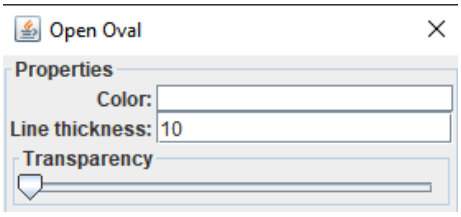


Figure 3: Configuration Dialog window

```
//showConfigurationDialog();
```

Figure 4: Test to ensure dialog window use

```
2 usages
public void showConfigurationDialog() {
  if (this.selectedShape != null) {
    if (this.configDialog == null) {
      this.configDialog = new JDialog((Window)getTopLevelAncestor());
    }
    this.configDialog.setTitle(this.selectedShape.getName());
    this.configDialog.getContentPane().removeAll();
    this.configDialog.getContentPane().add(this.selectedShape.getUI());
    this.configDialog.pack();
    this.configDialog.setVisible(true);
    repaint();
  }
}
```

Figure 5: showConfigurationDialog() method code (reference for the below)

The above method (See figure 5) will first initalise a JDialog object passing the getTopLevelAncestor() method.

```
public Container getTopLevelAncestor() {
    for(Container p = this; p != null; p = p.getParent()) {
        if(p instanceof Window || p instanceof Applet) {
            return p;
        }
    }
    return null;
}
```

Figure 6: getTopLevelAncestor() method derivation

The getTopLevelAncestor() method (see figure 6) is a member of the JContainer class, which is a subclass of Component in the Java Swing library. The Component class is the base class for all Swing components.

```
public void setTitle(String title) {
    String oldTitle = this.title;

    synchronized(this) {
        this.title = title;
        DialogPeer peer = (DialogPeer)this.peer;
        if (peer != null) {
            peer.setTitle(title);
        }
    }
    firePropertyChange( propertyName: "title", oldTitle, title);
}
```

Figure 7: SetTitle() method definition in Dialog.Java

The method showConfigurationDialog() then invokes the setTitle method (see figure 7) which is defined in the Dialog.java class. This method then invokes a firePropertyChange() which is located in the Component.java class, and does not contain any more sub-methods (see figure 8 below)

```
protected void firePropertyChange( @NonNls String propertyName,
                                   Object oldValue, Object newValue) {
    PropertyChangeSupport changeSupport;
    synchronized (getObjectLock()) {
        changeSupport = this.changeSupport;
    }
    if (changeSupport == null ||
        (oldValue != null && newValue != null && oldValue.equals(newValue)))
        return;
    }
    changeSupport.firePropertyChange(propertyName, oldValue, newValue);
}
```

Figure 8: firePropertyChange() method derivation

The showConfigurationDialog() method then invokes a pack() method (see figure 9 below) which is located in the Window.java class which extends container.

```
public void pack() {
    Container parent = this.parent;
    if (parent != null && parent.getPeer() == null) {
        parent.addNotify();
    }
    if (peer == null) {
        addNotify();
    }
    Dimension newSize = getPreferredSize();
    if (peer != null) {
        setClientSize(newSize.width, newSize.height);
    }

    if(beforeFirstShow) {
        isPacked = true;
    }

    validateUnconditionally();
}
```

Figure 9: pack() definition

The above method pack() also invokes the addNotify() method (see figure 10 below)

```
public void addNotify() {
    synchronized (getTreeLock()) {
        Container parent = this.parent;
        if (parent != null && parent.getPeer() == null) {
            parent.addNotify();
        }
        if (peer == nu
            peer = get
        }
        synchronized (
            allWindows
        }
        super.addNotif
    }
}
```

java.awt.Container

public void addNotify()

Makes this Container displayable by connecting it to a native screen resource. Making a container displayable will cause all of its children to be made displayable. This method is called internally by the toolkit and should not be called directly by programs.

Overrides: addNotify in class Component

See Also: Component.isDisplayable, removeNotify

< 1.8 > (rt.jar)

Figure 10: overridden example of addnotify() method

It is worth noting that this method is originally seen in the Component.Java class and contains many more sub-methods, explaining them all would be unnecessary and confusing, however, some of the code for the addnotify() method in the component class is shown below (see figure 11)

Figure 11: addnotify() method derivation from component class

Refocusing on the showConfigurationDialog() method (sub-method of the addshape() method). The showConfigurationDialog() method (see figure 5) then invokes a setVisable() method and passed Boolean true. The derivation of the setVisable() is shown in the snippets below (see figures 12-17) with relative captions.

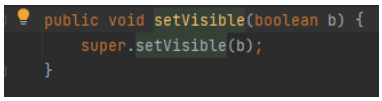


Figure 12: setVisable() method in Dialog .java

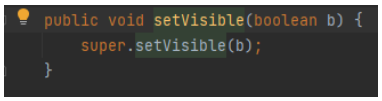


Figure 13: setVisable() method in window.java class

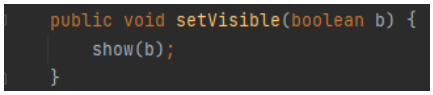


Figure 14: setVisable() in Component.Java

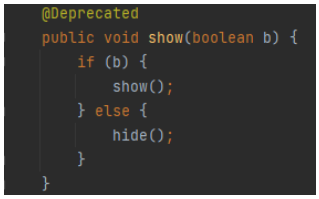


Figure 15: show() in Component.java (cropped)

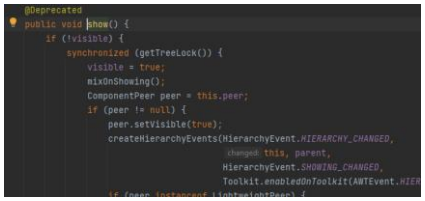
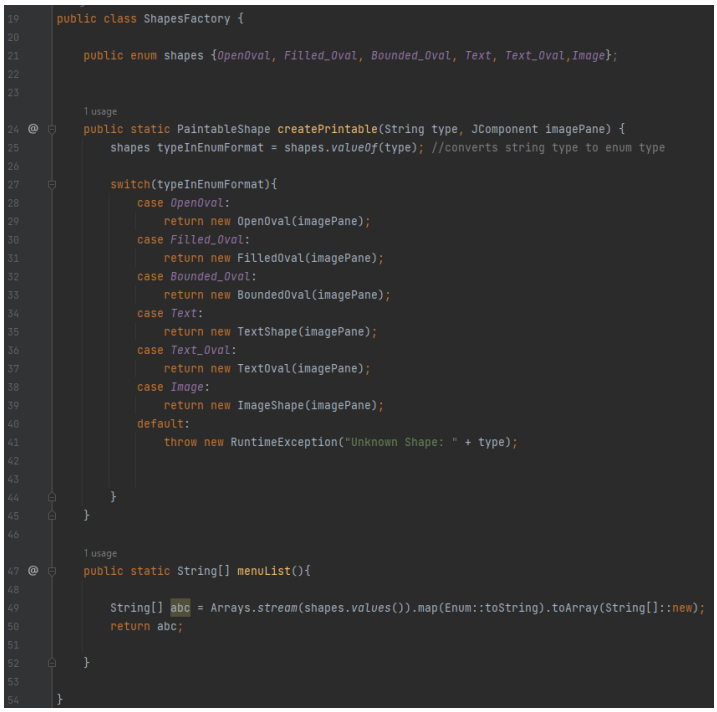


Figure 16: show() in Component.java (cropped)

Figure 17: hide() in Component.java

The showConfigurationDialog() method then invokes the repaint() method, for which the original derived code is provided above (see figure 2).

Task 3 – Create a factory class and refactor the application

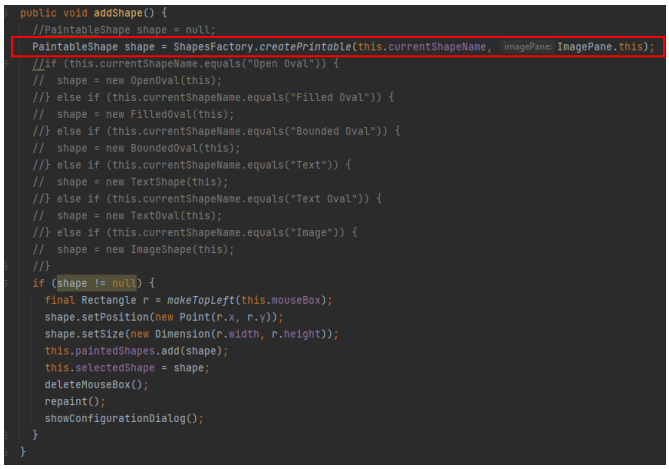


```
public class ShapesFactory {

    public enum shapes {OpenOval, Filled_Oval, Bounded_Oval, Text, Text_Oval,Image};


    1 usage
    public static PaintableShape createPrintable(String type, JComponent imagePane) {
        shapes typeInEnumFormat = shapes.valueOf(type); //converts string type to enum type

        switch(typeInEnumFormat){
            case OpenOval:
                return new OpenOval(imagePane);
            case Filled_Oval:
                return new FilledOval(imagePane);
            case Bounded_Oval:
                return new BoundedOval(imagePane);
            case Text:
                return new TextShape(imagePane);
            case Text_Oval:
                return new TextOval(imagePane);
            case Image:
                return new ImageShape(imagePane);
            default:
                throw new RuntimeException("Unknown Shape: " + type);


        }
    }

    1 usage
    public static String[] menuList(){

        String[] abc = Arrays.stream(shapes.values()).map(Enum::toString).toArray(String[]::new);
        return abc;

    }

}
```

Figure 18: Factory class

To allow additional shapes to be added in the future without significant modification of current code, a factory class name ShapesFactory has been created (see figure 18). This class is capable of instantiating and providing PaintableShape objects on demand (Child.M 2022), as well as having the relevant shape names added to the shapes menu. The factory class contains a method ‘createPrintable’ which uses enums within a sequence of switch statements to determine which shape object to instantiate once the drawable rectangle is unclicked . Because the CurrentShapeName variable is defined as type String, the createPrintable() method also converts the String type passed in the method to an enum type.

The factory class also contains the method ‘menuList()’ (see figure 18 above) which is responsible for returning the list of shape names which will be used to add the available shape objects to the menu. This eliminates the need for the addMenuOperation() method to be called multiple times for every shape, as well as eliminating the need for a programmer to have to modify the PaintOn class when adding or removing shapes. This has been achieved by converting the list of enums to a string array which will then be passed to the addMenuOperation() method once within the PaintOn class, this string array is returned as abc.



```
public void addShape() {
  //PaintableShape shape = null;
  PaintableShape shape = ShapesFactory.createPrintable(this.currentShapeName,  imagePane: ImagePane.this);
  //if (this.currentShapeName.equals("Open Oval")) {
  //  shape = new OpenOval(this);
  //} else if (this.currentShapeName.equals("Filled Oval")) {
  //  shape = new FilledOval(this);
  //} else if (this.currentShapeName.equals("Bounded Oval")) {
  //  shape = new BoundedOval(this);
  //} else if (this.currentShapeName.equals("Text")) {
  //  shape = new TextShape(this);
  //} else if (this.currentShapeName.equals("Text Oval")) {
  //  shape = new TextOval(this);
  //} else if (this.currentShapeName.equals("Image")) {
  //  shape = new ImageShape(this);
  //}
  if (shape != null) {
    final Rectangle r = makeTopLeft(this.mouseBox);
    shape.setPosition(new Point(r.x, r.y));
    shape.setSize(new Dimension(r.width, r.height));
    this.paintedShapes.add(shape);
    this.selectedShape = shape;
    deleteMouseBox();
    repaint();
    showConfigurationDialog();
  }
}
```

Figure 19: Change to addshape() method for simplification of object instantiation

The previous sequence of if-else statements in the addshape() method has been replaced with one line of code (see figure 19 above) which uses the createPrintable() method in the factory class (see figure 18 above) and passes the currentShapeName as a parameter. ‘CreatePrintable was typed by mistake, a more appropriate name for createPrintable would be createPaintable. This design

decision ensures that a programmer in the future will not be required to edit this section of code (when adding or removing shapes) but instead can simply modify the ShapesFactory class. Note that the currentShapeName variable is capable of being passed because the createPrintable() method converts the String type to an enum type.

```
60      //addMenuOperation("Open Oval");
61      //addMenuOperation("Filled Oval");
62      //addMenuOperation("Bounded Oval");
63      //addMenuOperation("Text");
64      //addMenuOperation("Text Oval");
65      //addMenuOperation("Image");
66
67      for (String s : ShapesFactory.menuList()){
68        addMenuOperation(s);
69      }
70
```

Figure 20: Change to sequence of addMenuOperation() in PaintOn class

To add additional shapes to the menu, the addMenuOperation() method previously called multiple times for every shape, has been replaced with a for loop iterating through the list of shapes defined in the factory class which are returned in the listMenu() method and adding each shape name to the menu using the addMenuOperation() method (essentially one line of code) (see figure 20 above). This works using enums without the modification of other parts of the code as we had previously converted the enums to a string array in the menuList() method initialisation in the factory class (see figure 18 above).

Task 4 - Implement additional classes for the factory to create

To show the benefits of a factory class, the code below outlines the implementation of additional shapes including FilledRectangle, OpenRectangle, BoundedRectangle and TextRectangle. This has been done by creating additional relevant classes, however, it does not require the modification of any code that is not within the Factory Class.

```
10  public class OpenRectangle extends AbstractLineBasedPaintableShape {
11
        2 usages
12      public OpenRectangle(final JComponent imagePane) {
13          super( name: "Open Rectangle", imagePane);
14      }
15
        2 usages
16      @Override
17  @   public void paint(final Graphics2D g) {
18          g.setColor(getColorWithTransparency());
19          final Dimension size = getSize();
20          float thickness = 1;
21          final String tv = (String) getPropertyValue(lineThicknessKey);
22          if (tv != null) {
23              try {
24                  thickness = Float.parseFloat(tv);
25              } catch(final NumberFormatException x) {
26              }
27          }
28          final BasicStroke stroke = new BasicStroke(thickness);
29          g.setStroke(stroke);
30          g.draw(new Rectangle2D.Double( x: 0,  y: 0, size.width, size.height) {
31          });
32      }
33  }
```

Figure 21: OpenRectangle class

Figure 21 (above) shows the implementation of the OpenRectangle class which contains a superclass constructor (inherited from AbstractLineBasedPaintableShape) which allows the instantiation of the Open Rectangle object. Note that this class extends AbstractLineBasedPaintableShape which extends AbstractColouredPaintableShape, as for this shape we would like to allow the user to set the line thickness of the shape using the arbitrary Stroke object.

The paint() method overriding the interface class paintableshape is responsible for initialising the size (type Dimension) using the getSize() method which contains x and y lengths. The method automatically sets the thickness to 1 by default and maps it to the LineThicknessKey. This method also allows the transparency of the open rectangle to be set by the user. The colour of the shape can also be set due to the use of the getColorWithTransparency() method. The addpaint() method takes a single parameter, a Graphics2D object, which is used to draw the rectangle with the specified size onto the Graphics2D object. The Graphics2D class is a subclass of the ‘Graphics’. This class has been added to the ‘simple’ package as it is a singular shape (not compound)

Figure 22: FilledRectangle class

Figure 22 (above) shows the initialisation of the filled rectangle, note that this class does not extend AbstractLineBasedPaintableShape but extends AbstractColouredPaintable shape, this is because a filled shape cannot have a line thickness. This class has a constructor for the FilledRectangle allowing the instantiation of the FilledRectangle Object inherited from AbstractColouredPaintableShape which sets the transparency key to 0 and the colour to white by default. It also overrides the paint() method in the interface class which takes the same parameters as the OpenOval. The difference being that it does not require the Stroke object for the line thickness as stated above. The method then draws the object using the fillRect() method defined in the Graphics class. This class is also within the simple package.

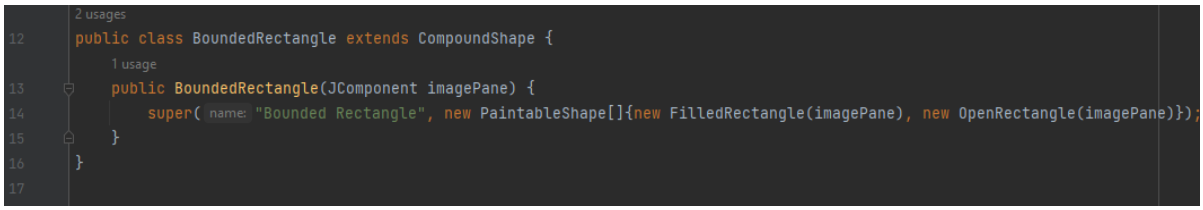


Figure 23: BoundedRectangle class

The BoundedRectangle class (see figure 23, above) extends compound shape as we would like to combine two shapes (open rectangle and filled rectangle). The constructor calls a superclass constructor which passes the array of PaintableShapes objects, it also consists of the FilledRectangle and OpenRectangle objects using the imagePane argument as their parameter. The super constructor is responsible for combining the two shapes, this is inherited from the CompoundShape class. It is worth noting that this class has been placed in the compound package

Figure 24: TextRectangle class

The TextRectangle class (figure 24 above) is responsible for allowing the instantiation of the TextRectangle which is a combination of the TextShape, FilledRectangle and OpenRectangle. Hence, the class must extend the CompoundShape class which contains a constructor allowing for this combination. The body of the constructor in the TextRectangle class first calls the superclass's constructor, passing it an array of three PaintableShape objects (FilledRectangle, OpenRectangle

and TextShape) as arguments. After, the constructor sets the transparency property of the FilledRectangle transparency key to 0.5

```
public class ShapesFactory {

    public enum shapes {OpenOval, FilledOval, BoundedOval, Text, TextOval,Image, OpenRectangle, FilledRectangle, BoundedRectangle, TextRectangle};

    1 usage
    public static PaintableShape createPrintable(String type, JComponent imagePane) {
        shapes typeInEnumFormat = shapes.valueOf(type); //converts string type to enum type

        switch(typeInEnumFormat){
            case OpenOval:
                return new OpenOval(imagePane);
            case FilledOval:
                return new FilledOval(imagePane);
            case BoundedOval:
                return new BoundedOval(imagePane);
            case Text:
                return new TextShape(imagePane);
            case TextOval:
                return new TextOval(imagePane);
            case Image:
                return new ImageShape(imagePane);
            case OpenRectangle:
                return new OpenRectangle(imagePane);
            case FilledRectangle:
                return new FilledRectangle(imagePane);
            case BoundedRectangle:
                return new BoundedRectangle(imagePane);
            case TextRectangle:
                return new TextRectangle(imagePane);
            default:
                throw new RuntimeException("Unknown Shape: " + type);


        }
    }
```

Figure 25: Updated Factory Class

The implementation of these shapes requires the modification of some code within the factory class. These changes are minimal due to the design and including, adding the additional shapes to the list of enums, and adding additional case statements for each shape added which return the respective shape. The menuList method (see figure 18) does not require any modification as it uses the list of enums which are converted and put into a string array allowing them to be passed to the addMenuOperation() method. This design decision reduces the workload on future programmers when changing or implementing additional shapes.