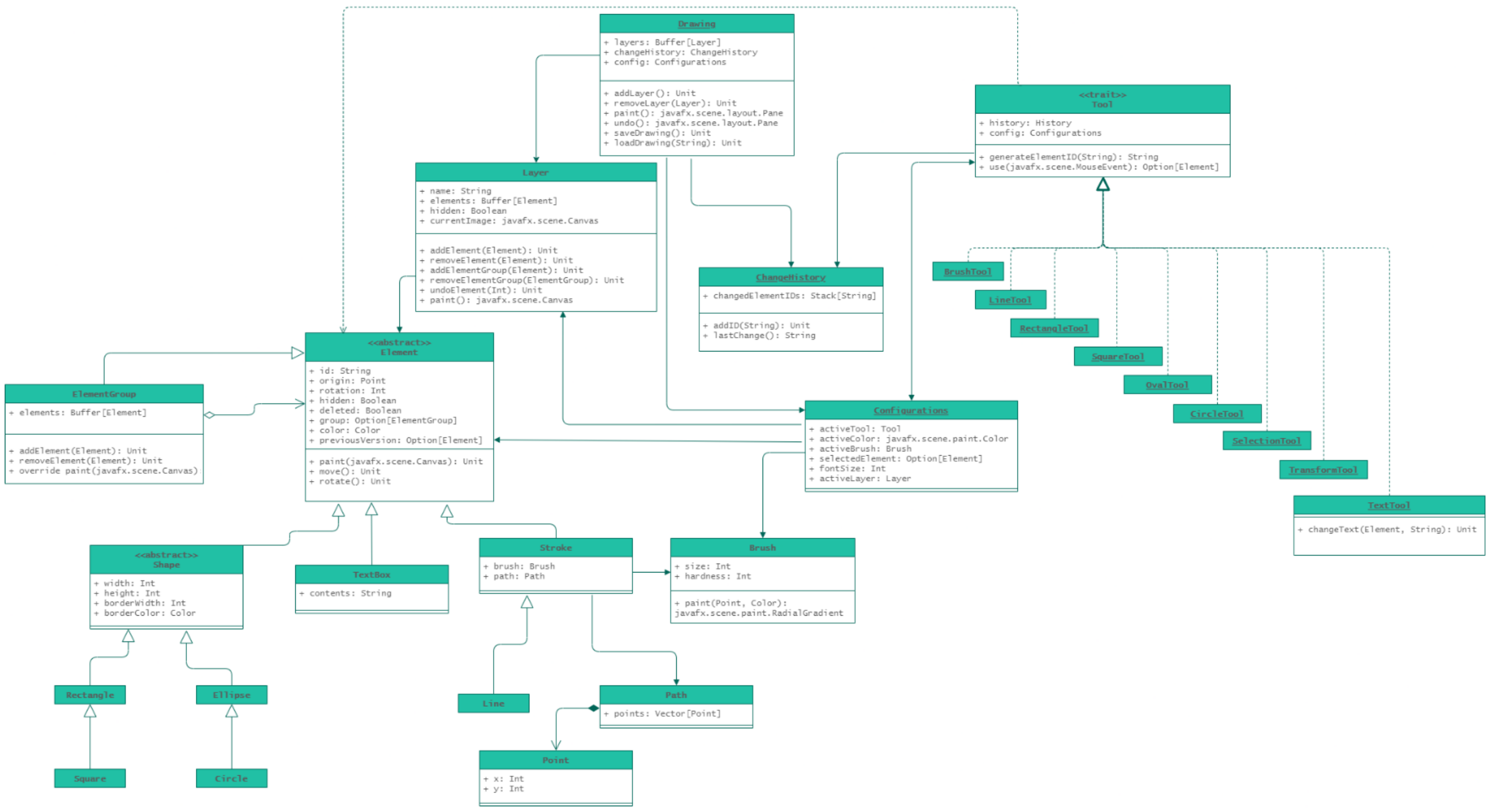
Filip Eller
1004225
18.2.2022

Sketch App – Technical Plan

Class structure

A UML diagram of the program's logic side's class structure:


Drawing
+ layers: Buffer[Layer]
+ changeHistory: ChangeHistory
+ config: Configurations
+ addLayer(): Unit
+ removeLayer(Layer): Unit
+ paint(): javafx.scene.layout.Pane
+ undo(): javafx.scene.layout.Pane
+ saveDrawing(): Unit
+ loadDrawing(String): Unit
<<trait>> Tool
+ history: History
+ config: Configurations
+ generateElementID(String): String
+ use(javafx.scene.MouseEvent): Option[Element]
Layer
+ name: String
+ elements: Buffer[Element]
+ hidden: Boolean
+ currentImage: javafx.scene.Canvas
+ addElement(Element): Unit
+ removeElement(Element): Unit
+ addElementGroup(Element): Unit
+ removeElementGroup(ElementGroup): Unit
+ undoElement(Int): Unit
+ paint(): javafx.scene.Canvas
ChangeHistory
+ changedElementIDs: Stack[String]
+ addID(String): Unit
+ lastChange(): String
BrushTool
LineTool
RectangleTool
SquareTool
OvalTool
CircleTool
SelectionTool
TransformTool
TextTool
+ changeText(Element, String): Unit
<<abstract>> Element
+ id: String
+ origin: Point
+ rotation: Int
+ hidden: Boolean
+ deleted: Boolean
+ group: Option[ElementGroup]
+ color: Color
+ previousVersion: Option[Element]
+ paint(javafx.scene.Canvas): Unit
+ move(): Unit
+ rotate(): Unit
ElementGroup
+ elements: Buffer[Element]
+ addElement(Element): Unit
+ removeElement(Element): Unit
+ override paint(javafx.scene.Canvas)
Configurations
+ activeTool: Tool
+ activeColor: javafx.scene.paint.Color
+ activeBrush: Brush
+ selectedElement: Option[Element]
+ fontSize: Int
+ activeLayer: Layer
<<abstract>> Shape
+ width: Int
+ height: Int
+ borderWidth: Int
+ borderColor: Color
TextBox
+ contents: String
Stroke
+ brush: Brush
+ path: Path
Brush
+ size: Int
+ hardness: Int
+ paint(Point, Color): javafx.scene.paint.RadialGradient
Rectangle
Ellipse
Square
Circle
Line
Path
+ points: Vector[Point]
Point
+ x: Int
+ y: Int

Drawing is the most important object. It keeps track of all the Layers that make up the user's artwork. The Drawing's paint method is what the GUI calls to get an image of the artwork. That method calls each of Drawing's layers' paint methods which in turn call each of their element's paint methods. Drawing is also in charge of undoing as well as file management.

Configurations object saves most settings the user has access to in the top and side panels of the GUI. Apart from the needed getters and setters the object doesn't have any methods. The Drawing object and many Tool objects read data from it.

Each Layer has a collection of Element which describe different kinds of graphical components, such as rectangles and lines. Elements can be added to a Layer and removed from it. Elements can also be grouped. An ElementGroup is also an Element itself, so most actions can be performed on groups just as well as individual elements. Specifically, a Layer saves its ElementGroups among all the other Elements.

Objects that inherit the Tool trait are used to make new Elements and modify existing ones. The Tools are named based on what type of an Element they create. The main functionality is accessed through the use method of which each Tool implements its own version. The Transform tool is used to move, scale and rotate the selected Element. When a Tool's use method relates to an existing Element, the method can get the selected Element as well as the current Layer from the Configurations object.

When a user has selected an Element using the Selection tool, the Element's properties can also be modified using the GUI's side panel. The relevant GUI objects have access to the selected Element's element variables through the Configurations object.

ChangeHistory keeps track of elements that have been changed/created/deleted. Calling a Tool's use method adds the relevant element's ID to the history, while calling the Drawing's undo method removes the last item in history.

Initially, I was going to use Swing as my GUI library, but I have now decided to use ScalaFX, since it's newer and presumably more advanced. The GUI is not entirely separate from the logic of the program because Element and Layer instances generate GUI components with their paint methods. I see this as justified since those classes are closely intertwined with the graphics they represent. Nevertheless, I might change this later because at the moment I don't have much experience in building GUIs, so it is hard to estimate what is or is not practical. The actual GUI outside the drawing will consist of panels with buttons, sliders and text inputs that manipulate the Configurations object's variables, the layers collection of the Drawing object and variables of an individual selected Element instance.

Use case description

Let's go through a use case of the program. The user starts the program, and a blank white drawing is displayed. They name the default layer "background". The user selects the rectangle tool and the colour blue from the palette. They drag their mouse across the whole canvas while pressing down to draw a rectangle that covers it. The rectangle is saved to the background layer's elements variable. As the mouse is pressed down a mouse event listener is fired which calls the use-method of the Drawing object's active tool, rectangle tool. Then, they create a new layer and name it "still life". The user selects the oval tool and the colour yellow. Similarly to using the rectangle tool, they drag their mouse to draw a yellow oval depicting a lemon. This time the element is saved to the "still life" layer. The user goes on and draws a green circle to depict a lime. Upon reviewing their drawing, the user decides to make some changes. Using the select tool, they select the lemon and rotate it some 30° degrees with the transform tool by dragging their mouse outside the element's bounding box. They select the lime and make it large with the transform tool by dragging the bounding box's corner. On first try the lime becomes too big, so the user undoes the scaling and tries again. When the user undoes their action, the Drawing fetches the ID of the most recently changed Element from ChangeHistory and calls the Configurations.activeLayer's undoElement method with the ID as a parameter. Still using the transform tool, the user moves the lime on top of the lemon. They switch back to the background layer, select the blue rectangle, and choose a slightly lighter tone for it from the side panel. Now the user is happy with their work, so they go ahead and save the drawing to a file.

Algorithms

The drawing updates in real time as the user draws an element or transforms an existing one. This is accomplished through re-rendering the drawing on the screen every 40 milliseconds. This way the drawing is animated with 25 frames per second while the user is making an input. This behaviour is started whenever a mouse event is registered and stops when the mouse is released.

Memory is saved by only repainting the relevant parts of the drawing. When a new element is drawn it is first placed on its own layer on top of the current layer and only that layer is animated. When the mouse is released to finalize drawing the element, the temporary layer is deleted, and the new element is appended to the active layer's elements variable.

When the user modifies an existing element, the whole active layer is animated. This is because one layer corresponds to one image component and changing one element can cause parts of other elements to be covered or become visible. The other layers are left intact.

Of course, memory won't be a problem in most use cases. However, if one was to draw an actual elaborate and detailed painting using this program, the drawing would soon consist of a few thousand brush strokes. Such a drawing is bound to have multiple layers and thus, a good bit of memory can be saved.

The program offers the user the ability to undo as many previous actions as is needed. Each element has a variable that points to the previous version of itself. The whole history of a given element is thus saved in a recursive manner. When an action is undone, the corresponding element is removed from its Layer and replaced by its previous version. That version in turn remembers its previous version so actions can be undone unlimitedly. The first version of an element does not have a previous version. At that point, undoing an action related to it causes the element to be removed for good.

Drawing circles and ellipses as well as applying a feather on a brush stroke could have needed some calculations, but thankfully the functions provided by the GUI library make things a lot easier. For example, with ScalaFX drawing an ellipse only requires the centre point and the lengths of the major and minor axes. ScalaFX then determines which pixels the perimeter goes through.

Determining the origin point to be given for a GUI function is not trivial in all cases, however. With rectangles, the origin point is the shape's upper-left corner. When a rectangle is drawn, if the user drags their mouse towards the down-right corner, the origin point is just the point they clicked first. When the mouse is dragged some other way, the origin point also depends on the position of the mouse:

Origin = (min(starting x-coordinate, current mouse x-coordinate), min(starting y-coordinate, current mouse y-coordinate))

Scaling a rectangle also need some pondering. They can be scaled from any side, so the origin point might move as a result. If the user scales the right-hand or the bottom side, the origin stays in place. When the top side is scaled the origin is (original x-coordinate, current mouse y-coordinate), and when the left-hand side is scaled (original x-coordinate, current mouse y-coordinate). To treat these cases differently, the program must recognize which side is being scaled. That depends on the position at which the mouse was clicked. For example, the top side is scaled if the click position's coordinates fulfil these conditions:

origin x <= click x <= origin x + width

origin y – 5 <= click y <= origin y + 5

Going into more detail at this point is not worthwhile.

Data structures

The Drawing keeps track of a varying number of Layer instances and each Layer in turn keeps track of multiple Elements. Both the Layers and the Elements must be kept in order so that they can be rendered correctly to the screen. I will be using Buffers for both collections as their mutability makes adding and removing items easy. The same goes for ElementGroups.

The ChangeHistory object uses a Stack[String] to save the IDs of the recently changed/created/deleted elements. This is because undoing actions always starts from the very last one and the “last in, first out” principle of stacks matches this need. When an action is performed on an Element, its ID is pushed to the stack. If the user wants to undo their last action, the ID pointing to the relevant Element is popped from the top of the stack.

Schedule

The app is heavily GUI oriented, so I will start with building the GUI’s foundations. As new features are added, the GUI is kept up to date.

Task	**Time needed (h)**	**Completed by**
Getting to know ScalaFX, base GUI	5	27.2.
Element, Rectangle, and Circle classes	3	
Base Drawing object and Layer classes	2	
Rectangle tool, square tool	2	6.3.
Configurations object and modifying it from GUI	2	
Updating canvas in real-time	3	
Oval and circle tools	1	13.3.
Layers class, working with layers	2	
Brush and line tools	3	20.3.
Selection tool	1	
Grouping elements	1	
Text tool	2	27.3.
Modifying element properties through side panel	2	
Transform tool	3	3.4.
Saving drawings to files	4	
Loading drawings	3	10.4.
Action history and undo	4	
Finishing the GUI	3	17.4.
Final testing	3	
Submitting	1	24.4.

Testing Plan

The program is very graphical by nature, so it makes sense to test it largely through the GUI. Most functions in the program have direct impact on the GUI, which means direct feedback for testing. For example, calling Drawing’s addLayer method should add a new Layer to be displayed on the GUI’s layer list, calling a Layer’s removeElement method should make it so that the Element is no longer displayed, and calling ChangeHistory’s addID method should add the ID to be displayed in the GUI’s layer list under its layer. Whenever a new tool is added, its main functionality is quick and easy to test in the GUI.

Test cases for the Rectangle tool:

1. Drawing a rectangle by clicking, dragging towards the down-right corner, and releasing.
2. Drawing a rectangle by dragging towards other corners.
3. First dragging one way, then another.
4. Just clicking once without dragging.
5. Dragging over the border of the canvas.
6. Clicking and dragging outside the canvas.
7. Drawing on top of another rectangle.
8. Drawing on a lower layer under another rectangle.

Square, Oval and Circle tools are tested similarly.

Test cases for the Brush and Line tools:

1. Drawing a stroke by clicking, dragging, and releasing.
2. Drawing with different brush sizes, specially the minimum and maximum sizes
3. Drawing with different amounts of brush hardness, specially 0 % and 100 %.
4. Clicking once.
5. Clicking, holding, then releasing.
6. Drawing over the edge of the canvas.
7. Drawing outside the canvas.
8. Drawing on top of and under other elements.

Before tools can be tested, the Element class as well as the Layer class and the Drawing object must be sufficiently tested. For Element and Layer unit testing without the GUI might be more efficient because many of their methods are called by Tools. The Drawing's addLayer and removeLayer are easy to test with the GUI:

1. Adding a Layer.
2. Adding multiple Layers.
3. Removing Layers.
4. Removing the only Layer.

Result should be seen in the GUI's Layer list.

The program uses JSON for saving drawings, so simple drawing files are human-readable. That enables examining saved files and comparing them to a human-written expected result. Loading a file has succeeded if the drawing looks the same as it was before it was saved. File management must be tested with different combinations of Elements of multiple types in varying layer and group structures. Important edge cases are mainly the same as in testing individual Elements.

I've planned the schedule so that classes that the basic structure of the program, that is the Drawing object, Layer class and Element Class, is started early on. This way other classes and objects that use or depend on them can be tested in realistic scenarios immediately.

Unit testing will also be used. Tools can be tested by giving their use method an artificial mouse event as a parameter and then examining the Element instance it returns. For example, we can give the Rectangle tool's use method a Mouse Event as input and test if the returned Element has the intended width, height, and origin. However, mouse events might be cumbersome to fake.

Testing Layers is easier since their methods use Elements which are easy to create by hand and the Layer does use an Element's internal data apart from its ID and previous version in the undoElement method. Test cases for a Layer instance include:

1. Adding an Element of each type.
2. Removing an Element that is stored in its elements collection.
3. Removing an Element that is NOT stored in its elements collection.
4. Adding an element group with a first element that is and is not in its elements collection.
5. Calling undoElement with a correct and incorrect ID.
6. Calling undoElement with an ID that points to an Element with no previous versions.

A mock version of the Element Class can be used for these tests to keep things controlled.

Testing Element's, Layer's and Drawing's paint methods can be achieved by creating Canvases independently from other parts of the program and examining what the methods do with them. The Drawing's paint method calls each of its layers' paint methods, so a mock Layer class can be used to return the intended Canvas when Drawing calls Layer.paint(). Test cases include:

1. Returning one canvas with something painted on it.
2. Returning multiple canvases (one from each layer) with something painted.
3. Returning an empty, transparent Canvas.
4. Returning Canvases of different sizes.

References and links

Programming 1 course material: https://plus.cs.aalto.fi/o1/2021/

Programming Studio 2 course material: https://plus.cs.aalto.fi/studio_2/k2022/toc/

A Swing demo from Programming Studio 1: https://version.aalto.fi/gitlab/studio-2020-demos/studio-demo-2020

Scala Standard API: https://www.scala-lang.org/api/2.13.2/index.html

ScalaFX docs: http://www.scalafx.org/docs/home/

JavaFX docs: https://docs.oracle.com/javase/8/javafx/api/toc.htm

Java AWT docs (used while researching): https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

Java Swing docs (this one too): https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

JavaFX tutorials on Tutorials Point: https://www.tutorialspoint.com/javafx/index.htm

Scala tutorials on Geeks for Geeks: https://www.geeksforgeeks.org/scala-programming-language/?ref=lbp

Youtube videos I watched while researching:

https://www.youtube.com/watch?v=Kmgo00avvEw

https://www.youtube.com/watch?v=pDafZdIIeNE

https://www.youtube.com/watch?v=5o3fMLPY7qY

https://www.youtube.com/watch?v=CmK1nObLxiw

Other

https://stackoverflow.com/questions/852631/java-swing-how-to-show-a-panel-on-top-of-another-panel/852792#852792

https://stackoverflow.com/questions/70142710/how-can-i-create-a-feathered-brush-with-javafx

https://www.tabnine.com/code/java/classes/java.awt.RadialGradientPaint