# Step-by-step application development description.

## Requirements List

1. Application GUI should have a look according to fig 1.

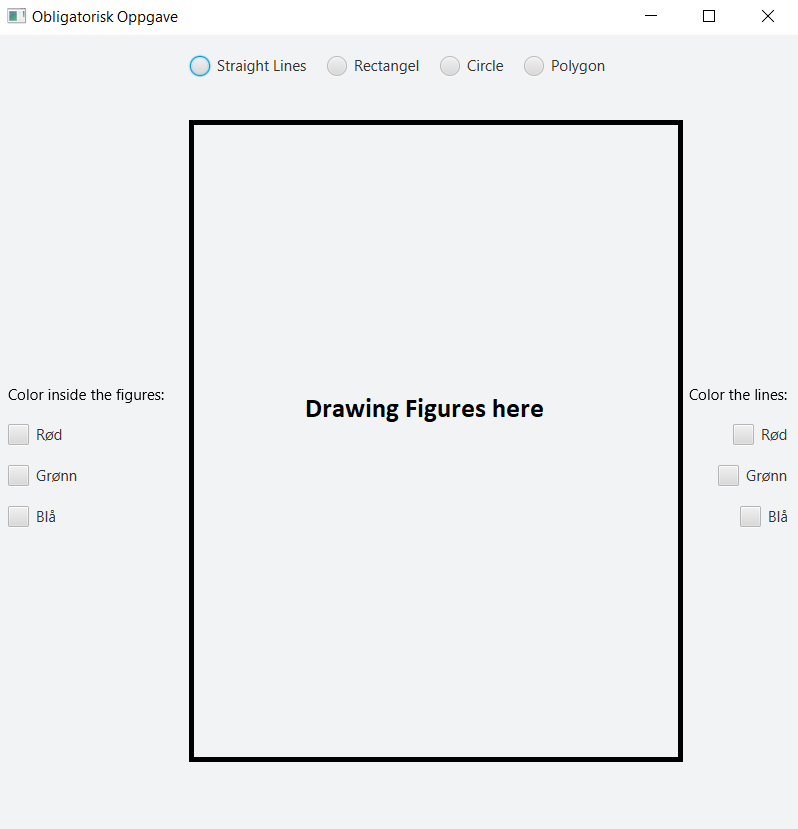


Fig 1. GUI approximate view

1. Additional to fig 1, GUI should include a “clear figures” button at the bottom of it.
2. GUI default size should be 800\*800 pixels
3. RadioButtons should allow only one of them to be selected in the same time. One should be selected by default. Radiobuttons switch a type of figure being drawn. Switching to a different figure ends current drawing process
4. Checkbox should allow multiselection. They determine stroke and fiil of currently drawn figure. Switching their state while figure is being drawn should update its stroke and fill
5. Button should clear all figures from screen/
6. Pane in the center and only it should allow user to draw currently selected figure type using mouse interface similar to drawing with mouse in MS Paint.
7. Drawings should be represented on screen by Shape objects
8. Existing drawings should be kept in ArrayList and shouldn’t be removed from screen when a new drawing is created
9. Only left mouse buttons interactions are to be processed
10. Shapes strokes should have width 4 and have a round line cap

## 1, GUI development

### Application development

GUI is defined by requirements 1 and 2. But before the requirements could be met, we first need to display a JavaFX window on screen. So we start from creating a class derived from JavaFX Application class, that’s going serve as an entry point for an application. Implementation of such class required two methods: “main”, which is default application entry point and which only function is to launch our JavaFX application object, and “start” where we performs operations as soon as our JavaFX Application is launched: here we create a window, by packing top level frame into the scene and scene into the stage. And here we comes for necessity of top level frame class

### Top level frame (pane)

This pane would be the one that is displayed for the end user on screen and the one, that should implement the requirements 1 and 2 Since it is a top level pane, it is a good idea to inherit this class somewhere from a Pane class hierarchy. According to requirement 1, the best candidate is a BorderPane, which layout perfectly matches requirement 1 and necessity to place components into 5 distinct parts of the pane: top, bottom, left, right and center. After class is created, to met requirements 1 and 2 we should go into its constructor and add and setup JavaFX components until resulting pane is perfectly matched with the required one. To do so we should:

* Add radiobuttons to the top section of the pane. To group them and organize them along x-axis, we place a HBox container on the top of the pane and put them into it.
* Add button to the bottom section. Since button is singular any container would suffice but for the sake of symmetry we put the HBox at the bottom as we did at the top and store our button in it.
* Add label with header text and three checkboxes both on left and on right sections of the pane. Since they are organized vertically, we use VBoxes as containers for this two sections.
* Finally, we add a simple plain Pane to center section of the top-level Pane to serve as parent for the to be drawed shapes.
* We setup text for all labeled elements, their alignment in their parent containers (Center for top and bottom, Left-Center for left and right), margins between them and containers borders and shrink/grow limits in such way, that all shrinking/growing is done by the central section, while other four keep their size constant. We also setup border of the central pane, using CSS style.
* We set the pane preferred size to meet requirement 3.

### Controls behavior

To implement requirements 4-7, we must finish setting up controls created previously. For requirement 4, we must add a ToggleGroup, which allows radioButtons to work as group and guarantees that only one of them would be selected. We also need to set up their default selection state, choosing one (Line one) as a selected by default. To implement last part of requirement 4 we must make our radioButtons inform the class about selection changing. This is done via Listener (Observer pattern) by adding Listener to the “Selected” property of the ToggleGroup, which would notify class as soon as selected radio button changes.

For requirement 5 we set the default state of all checkboxes and then add a handler for their action event, which represent switching checkbox state and so changing the color. Handler is common for all six checkboxes and so both Fill and Stroke would be updated, when one of them changes

For requirement 6, we just need to do the action handler for the button, that would call our class method when the button is pressed

Finally for requirement 7, we need to add a plethora of MouseEvent handlers to the central Pane, to keep track of Mouse Presses, Releases and Drags. We also should add a clip rectangle to the pane to prevent the drawings to spilling outside of it and bind this rectangle size to the pane size. (In real life, clipping is added after first drawing test with a thought: “Useless pane, can’t even do clipping by default!”)

Here the GUI is complete and we can move onto the inner logic of the app.

## 2. Logic development

### Handling GUI events

Since requirement 9 states, that we should use an ArrayList to store the drawings, it means that they must ***share a common superclass of Interface***. Java way is to use an Interface in moments of doubt, so we add an empty interface Figure, create an ArrayList “figures” of it in the pane class and should make sure to populate it according to the needs discovered during handlers implementation.

First thing first: requirement 8 states that drawing should be represented by Shape objects on screen and so our Figure must return a corresponding Shape. We add this to the interface contract. Then given Shape we need to customize it: add to it Fill and Stroke colors and set it accordingly to requirement 11. To do this we add an additional method to the pane class, which would perform this customization on the shape - decorateShape method . To use Fill and Stroke colors in it, we would need a way to transform checkbox states into the corresponding colors. We add two methods for that as well: getFillColor and getLineColor, fulfilling more of requirement 5.

One of the shapes is different from the others: all shapes of already drawn figures are static, but the shape of currently drawn figure can change. We keep it separate in the class field ***tempShp*** ***(temporary shape),*** to be able to easily add and remove it from the children list of our central Pane, which is working as canvas. We add a method update Temporary Shape to solve a task of receiving it from current figure, decorating it and replacing the old shape with updated one in a Pane children list. And this required us to have an additional link to a current figure, which we would keep as another class field: tempFig.

After this we can finish requirements 5 – by setting up handler for the checkbox state changes to redecorate the tempShp (which would change its appearance on screen immediately), and 6 – by setting up handler for the button press to clear ArrayList of figure, removing current figure and removing all shapes from the central pane children list.

Now to the MouseEvents. To fulfill requirement 9, we add checks at the beginning of each handler to only pass on events caused by left mouse button. Then following the requirement 7 we would have two modes in accordance to MS Paint drawing rules – when nothing is drawn and when something is drawn or NonDrawMode and DrawMode. To distinguish them we add a boolean flag to the class. In NonDrawMode, mouse press is a signal to start drawing, First we must create an appropriate figure: we define a method for that: getSelectedFigure, but for now leave it be. We would fill after implementing our interface and its implementations. After getting figure, we get it shape and add it to the children list of our “canvas” pane, decorating it first. Then we officially switch to a DrawMode. In DrawMode mouse press adds another point to a figure, that is being drawn. We would add corresponding method to an interface. And after calling it, we should update the figure shape on screen.

Next event is mouse drag. It only has effect in DrawMode and result in moving the last added point of a figure. We add method to an interface for this as well. After point was moved, we again need to update the figure shape, so we call our update method.

Finally, there is mouse release. It also only has effect in DrawMode. Here we first want our figure to be notified that user have chosen the final position of its new point and no more movements would follow. So we add a required method for this to an interface. After that, we assume that shape in reaction could have changed, so we update it. Then we need to check if the drawing will keep going or is the figure done. So we add a method to an interface, which returns Boolean answer for this question. If figure is done, we remove DrawMode flag and move it into the permanent figures ArrayList.

At last, there is only one requirement left: last part of requirement 4 – about ending drawing process. To achieve this we again need to expand the interface by adding a method that forces figure completion. We would call it in the corresponding handler, which handles the changes of selected radio button.

And that’s it. All requirements is fulfilled… well except the new bunch that we defined via filling up interface. Let’s look at it.

### Interface Figure

This interface now has 6 methods:

* Shape extraction getShape
* Adding additional point addNextPoint
* Moving last added point moveLastPoint
* Being notified that it last point would no longer be moved: completePoint
* Checking that is has been complete: isFigureComplete
* And forcing it to become complete: completeFigure

The contract is as follow: we create an object of an interface and may immediately call moveLastPoint on it any number of time, until we send completePoint call to it signaling that we are done. After that we can call addNextPoint on it and then going back to moving that point, until figure answers true for isFigureComplete request. At that point we wouldn’t be able to add points to it. We also can make it complete by calling completeFigure, which would make following calls of isFigureComplete return true. At any moment we can request and should receive a corresponding Shape object from the figure.

### Implementing line, circle and rectangle

Those three figure has a lot of common, more exactly they are fully defined by exactly two points: two ends for a line a and two opposite corners of a bounding box for rectangle and circle(actually ellipse). One of this two points would be fixed and be the origin point. Second would be movable by user. Those figures are complete at creation moment and no new point addition is possible for them. So to not implement all this common traits multiple times, we create a helper class TwoPointsFigure which has all this traits, implements Figure interface and serves as a super class for those three.

Each individual of them only differs in a way how it implements getShape method, converting two store 2D points into either Line, Rectangle or Ellipse object. Take note, that returned Shape isn’t connected back to the Figure and changes to Shape doesn’t affect the figure.

### Implementing polygon

The polygon may be defined by any number of points. This class stores this points in flattened (1-D , where elements 0,2,4 are X coordinates and 1,3,5 are Y coordinates) ArrayList to be compatible with its paired shape object – polyline. Polyline Shape is chosen over polygon, since polyline can display open not fully drawn polygon with one of its edges missing. By using such ArrayList we achieve that getShape is trivial method, easily converting or List into the Polyline object. Moving last point is implemented as removing and then replacing last two coordinates in the list with a new values. Adding next point is trivially adds its coordinates into list. The only other non-trivial feature of this class is snapping. Every time new point is placed (not added, but finally placed with guarantee of not being moved after via completePoint call) it is checked for proximity to polygon origin point. If it is in square 11\*11 around it, then it would be moved (removed and replaced)to origin point coordinates and polygon would become complete. On forced completion polygon just adds point at the end of a list at location of origin point, which closes its last open edge.

### Final touch

And now we can go back to the MainFrameclass and finally implement the getSelectedFigure method: it would go through all the radio buttons and for one selected it would create and return corresponding figure-implementing objects. (pattern Factory). In impossible case of no radio buttons selected it would return null. As a last touch we add a check for this null into the MousePress handler ,where we call getSelectedFigure, so that in case of null, we just didn’t go into the DrawMode.