CAB 302: Software Development Project

Group 200
Julie Haga n10320008
Ingvild Giset n10402624

May 2019

1 Functionality

Our team managed to create a fully functional Vector Design Tool. The Java software allows loading, preview and editing of already existing VEC files, as well as to design/create new VEC files. We have managed to implement basic functionality such as a toolbar where you can select different drawing shapes, a color palette used to set colors to the drawing tools and options to save or load images. Our system is user friendly and easy to use, with a simple and tidy layout.

1.1 Tools

Our system has a `Load` button, which allows the user to navigate through their file system to find already existing VEC files. Only .VEC files are displayed by default. Our system will open and display the loaded image correctly.

The `Save As` button opens a file save dialog, where the user can select the path to where the VEC file should be saved. The image will automatically be saved as .VEC if the user saved the file without an extension.

Our system allows the user to remove the latest graphical operation of the image by clicking at the `Undo` button, or by pressing CTRL + Z. The operation will also be deleted from the VEC file. This feature works both on new images, and on freshly loaded VEC files. The function can be used repeatedly, always deleting the latest operation until the image is completely blank.

A quick way to delete all the graphical operations of the image and VEC file is by using the `Clear All` button. This can also be done on both new images and loaded files.

If the user tries to close the system window after starting drawing, they will receive a question asking for a exit confirmation. This dialog will not appear for a blank canvas.

1.2 Shapes

The submitted program has a toolbar with different drawing shapes as buttons. All the different drawing shapes works as they should, with no issues.

Figure 1: Different drawing shapes

The line tool is set by default, and allows lines to be drawn by using the mouse. The first click with the mouse implies the start of the line, the user then drags and drops the mouse to create the line. If the user press the rectangle tool, rectangles will be drawn onto the image by using the mouse. The same drag-and-drop functionality is used for the rectangle and ellipse.

The plot tool allows the user to draw dots by clicking the mouse. The dot will be drawn once the user releases the mouse.

The system also has a button for a polygon tool. This tool allows the user to draw self-made polygons. With clicks of the mouse, the user can place vertices of the polygon. To close off the polygon and to end the drawing, the user has to click back on the first starting point of the polygon.

1.3 Colors

Figure 2: Colorbar

The color palette is used to set colors to the different drawing tools, where you can choose different colors for pen and fill. The color palette has a set of quick selection color buttons, as well as a button for a color chooser. By clicking the color chooser button, the user can select any possible color.

The user manually selects colors to pen and fill separately, by switching between the pen and fill radio boxes. That is, if the fill box is checked, the user will select which fill color to use. If the pen box is checked, the user are selecting the outline/pen color. By default, the pen is black with no fill color. The user can at any time click on the no-color button when selecting fill-color, which will result in no fill color. This is not possible when the pen box is checked, because the user always has to have an outline color.

1.4 Additional functionality

For our additional functionality we chose to implement a history function. In our toolbar we added a button called `View History`, where a pop-up window occurs if the user clicks it. The new window contains a list of all actions done by the user in a chronological order. The actions are shown by the name of the shape and the color, for example "Rectangle #000000". The user can click on any command for a preview of how the image was at the time. If the user selects an earlier version of the image, and then clicks select, the program will render that version of the image and add an "Edit from history" event to the history log.

2 Contribution

The team consists of two people, Julie Haga and Ingvild Giset. Both members have equally been working on both software and the final report. We met in person every time we worked on the project, so all approaches and solutions have been discussed face to face and solved together. This made our teamwork unarguably good.

Both members have frequently added code to GitHub and updated the report throughout the project. As visible on Github, we worked on different branches with different tasks to avoid mess and conflicts in the code. Ingvild was responsible for the front end and has worked mainly in the GUI branch, whilst Julie has worked with the back end implementation and unit testing in the drawManager and fileManaging branch. We have splitted the documentation of the code between us, and both have frequently done testing sessions of the program.

Ingvild has been the one responsible for reviewing the report and the final delivery of the assignment.

3 Agile software development

To create a software application from scratch, it is important to think things through before getting started. Planning and creating a good modular architecture makes it easier to find concrete tasks and separate them between the group members, as well as to avoid spaghetti code.

Our first step was to sit down together and draw out a class architecture with pen and paper. We then took use of GitHub for all the implementation for our project. As our team worked on different branches and parts of the system, a certain structure and a good working plan for organizing everything was important for us. To achieve this, we made use of an agile development process called the Kanban method. The Kanban method basically helps the user to visualize, manage and arrange tasks, as well as to maximize the work efficiency by using a board. Our Kanban board was created on the website *Trello*.

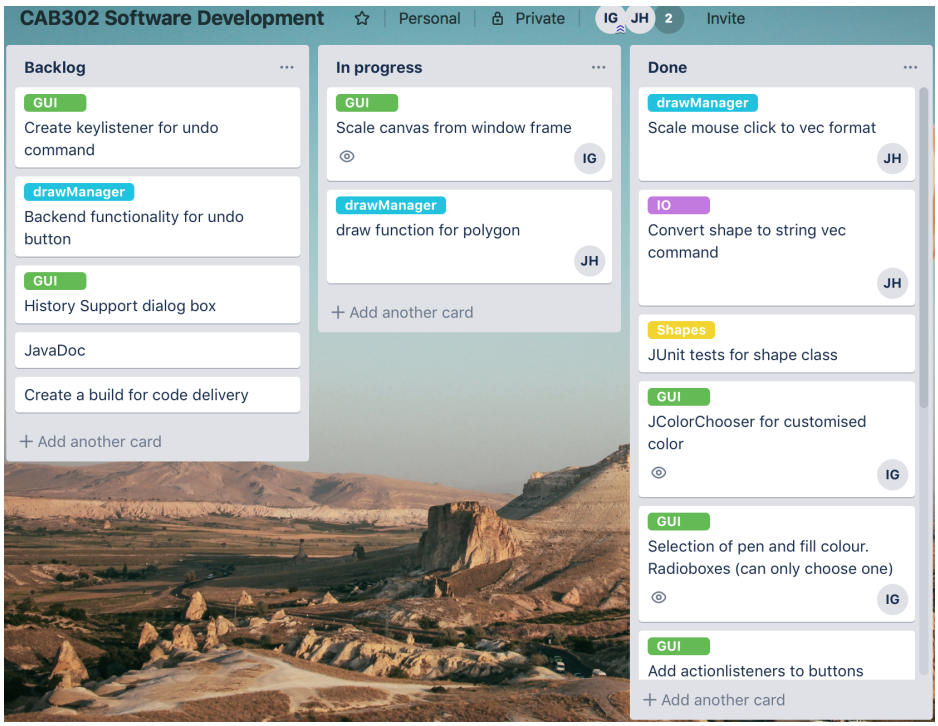

Figure 3: Kanban board

Together we started by creating a Kanban board with three lists: *Backlog*, *In progress* and *Done*. All our thoughts and ideas, project criteria and must-haves were added to the backlog list as cards. We created different labels with different colors (*drawManager*, *fileManaging*, *undoHistory*, *GUI* and *Shapes*) corresponding to the different packages in the project. We also organized our branches in GitHub the same way. By placing all the cards/tasks within one of those categories we had a good overview of what needed to be done in the branches at all times.

With the board as a project base, we systematically worked our way through the listed cards. Focusing at only one thing at a time increased the efficiency as we slowly but steady minimized the backlog.

We made use of a *feature driven development*. All the cards was different features from the project specification and we kept working on the same card until we were satisfied. This was done by frequently running and testing the functionality, then make refinements and changes based on the results. When correctly implemented, the card was moved to the *Done* list.

Another Agile method we frequently used was *pair programming*. Even though we mostly worked on separate problems, we also programmed a lot together. If one of us ran into some implementation issues, we sat down in front of one computer to solve the problem. With us having different strate-

gies and suggestions, we could discuss and agree on the best approach for the implementation. This made it easier to address and solve the issue.

When a bigger task or part of the program was completed, we wanted to merge this into each others branches - as well as the master branch. This to make sure that we both had the most recent version running. If a merge conflict emerged, we both looked over the conflict making sure that the correct version of the file was commited.

Overall, the use of a Kanban board improved the development process of our project, as well as the design, functionality and quality of our system.

4 Software Architecture

Our project consist of five packages. `DrawManager`, `IO`, `Menus`, `Shapes` and `SquareImage`. In this section we are to describe each class and how they interact with each other. In the class diagram in figure 4 an overview is presented. The blue outline models a package and the blue arrows models a dependency. The direction of the arrows shows the direction of the dependency.

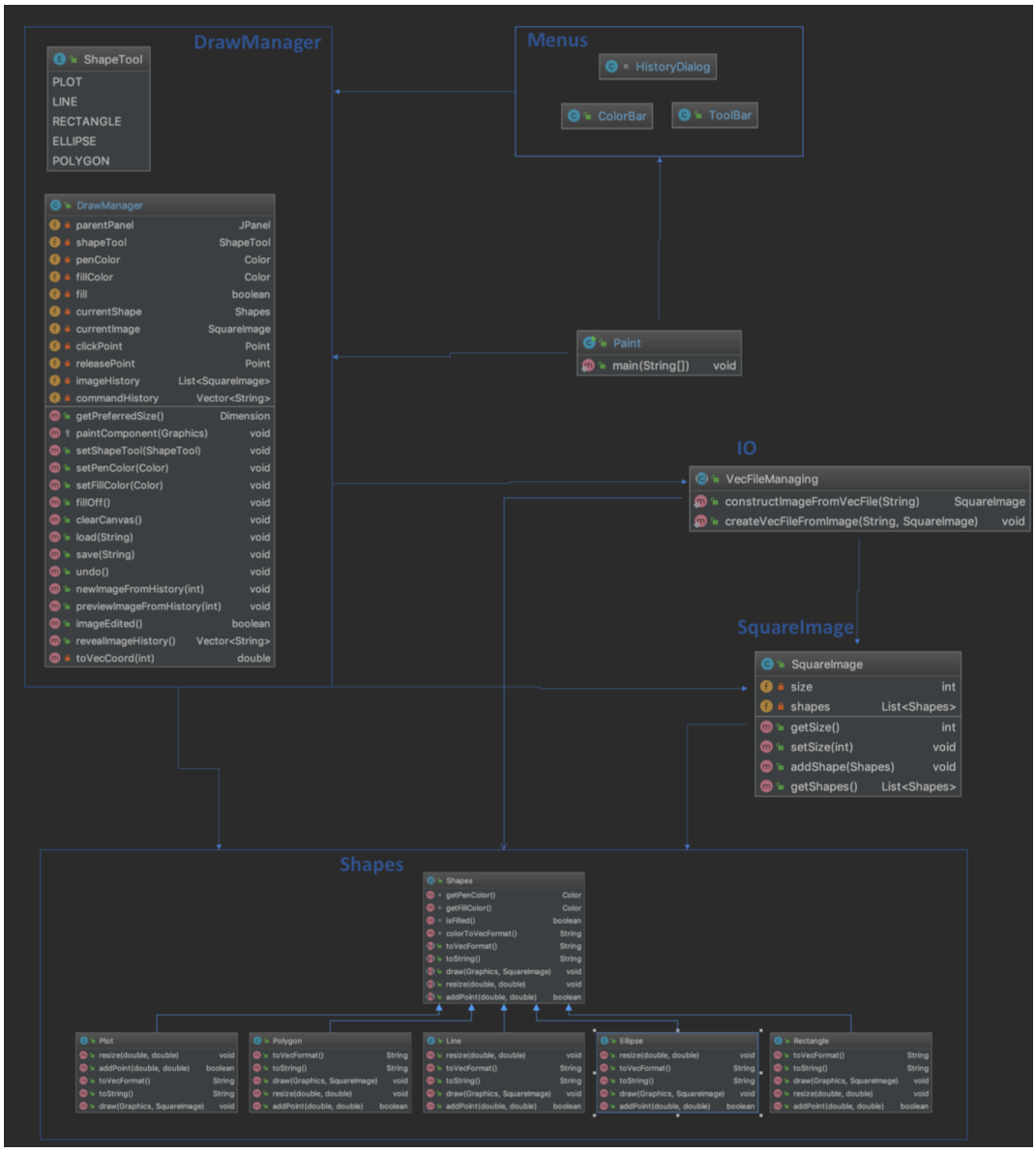

Figure 4: Class architecture

4.1 Paint

The `Paint` class is a simple class that runs our main. It creates the GUI and is a connection point between the other packages. `Paint` extends the java class `Frame`. In the frame we create instances of the different panels we have implemented. As represented in the diagram in figure 4, `Paint` is depended of the

Menus and the DrawManager package.

4.2 DrawManager

DrawManager is our most complex class and most of the functionality of the program is implemented here. In the package we have created an Enum for the different drawing commands to improve readability. The class has three dependencies; it includes the package Shapes, IO (VecFileManaging) and SquareImage.

DrawManager handles the drawing. The class extends the java class JPanel and allows us overwrite paintComponent to draw figures. We have also overritten the getPreferredSize() to make sure that the panel always is square. A mouselistener is implemented to track the click point, release point and dragging of the mouse as the user draws.

Because it handles the drawing, the class needs variables for the tool (color, fill and drawing command). The tool can be changed using different set functions. setPenColor(), setFillColor(), setShapeTool(), fillOf().

As the Shapes objects are created, we want to add them to a SquareImage object. This image can be saved to a .VEC file. We can also load an existing image, which is done by using functionality in the IO package.

The DrawManager keeps track of the history of the image by storing every version of the image along with the commands that are used. This lets us create history support for the system.

4.3 Menus

The Menus package contains the different menus that are displayed in the GUI. The classes contains only front end implementation, so they don't have any methods to be used by other modules of the program. The menus are all dependent of the DrawManager package.

The ColorBar and ToolBar class extends a JPanel. We have implemented action listeners for the different buttons. These calls different set functions from DrawManager when they are triggered.

The HistoryDialog class extends JDialog. It is a simple dialog box that displays a JList containing the command history. The command history is obtained from the DrawManager object.

4.4 IO

IO package contains one class, VecFileManaging. This class contains only static methods, so no objects are to be instantiated of this class. As figure 4 shows, it has no public constructor.

The VecFileManaging class is dependent of the SquareImage class as it can create a SquareImage object from file. It is also dependent of the Shapes class. When creating a SquareImage the class reads vec commands, creates Shapes objects and add them to the image. To write vec commands to file, it iterates

over the `Shapes` of the image and uses functionality from the `Shapes` class to obtain strings.

4.5 Shapes

This package contains one abstract class `Shapes` and five subclasses that inherits from that superclass. One subclass for each drawing command; plot, line, ellipse, rectangle and polygon. This package is not dependent of any of the other classes in the project.

The class implements functionality for different shapes to be constructed, resized and drawn. It also has methods for converting shape objects into strings and .VEC format.

4.6 SquareImage

The `SquareImage` class is the image itself. All images are created square as stated in the project specification. This class is very simple containing a getter and a setter for the image size. It has one normal constructor in addition to a copy constructor to make a deep copy. Each image contains a list of all the `Shape` objects that are drawn at it. It is therefore dependent of the `Shape` class.

5 Object Oriented programming principles

5.1 Abstraction

In our program we have made use of abstraction through an abstract class called `Shapes`. This class contains everything that is common for the different drawing commands, that are the sub classes.

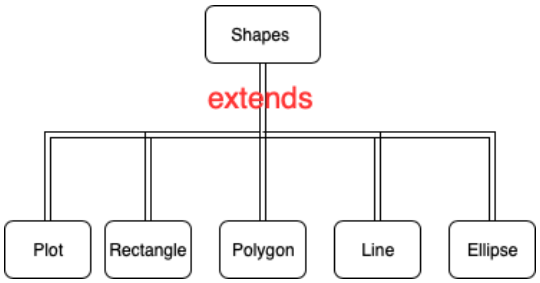

Figure 5: Abstract class

Since the `Shapes` class itself is abstract we can't create an instant of it. That's because it is not a complete implementation, it is only a base for the sub classes. That means that if we want to create a Shape object we must decide whether it should be a rectangle, ellipse or a line and so on. Making use of an abstract class limits the duplicate code between the different shape types because they have a lot in common. All the shapes has a pen color and a fill

color (or no fill at all). Another thing they have in common is that they need to be compatible with a vec format, described as a string, resized and drawn. However, the implementation of this functionality would be different for every drawing command. Therefore we make the functions abstract, meaning that each subclass needs to make a custom implementation of it.

5.2 Encapsulation

Using encapsulation, the variables of a class will be hidden from other classes. This means that they can only be accessed through methods of their own class. Also known as *data hiding*. (*Java Encapsulation*, n.d.) We made use of encapsulation in a lot of our classes. By declaring some of the variables `private` they are hidden inside that class. To change or read that variable we have used `get` and `set` functions. One example from our code is the sub class Line (or any other sub class) of `Shapes`. We have encapsulated the coordinates the line is drawn between. These are set in the constructor and can be changed through `resize(double x2, double y2)`. When the line is drawn, it reads its own variables and draws the line on the image. Since no other part of the program actually needs this data, we have no getter for it. Another example is the private variable `size` in the class `SquareImage`. The size is set in the constructor when an instance is created. As the window frame changes we change the size of the image, we therefore need a `setSize(int size)`. To decide the vector coordinates of a mouse click on the canvas we need the image size. This is returned by the `getSize` function in `SquareImage`. Not only variables can be encapsulated. In `DrawManager` we have a function `toVecCoord` that coverts a canvas position to a vector coordinate (number between 0 and 1). This is a function to help with calculations, and is not needed outside the class. We have therefore hid it to the outside by making it private.

5.3 Inheritance

Inheritance means that one class inherits attributes and methods from another class. The class being inherited from is called the superclass and the class that inherits is the subclass. (*Java Inheritance*, n.d.)

As described earlier we have used inheritance in `Shapes`. We created a superclass and different subclasses to inherit from that superclass. We have also made use of inheritance from embedded Java classes, making a customized implementation to fit our use. A lot of examples can be found in our implementation of the GUI. By letting `drawManager` inherit from the java class `JPanel` we get a lot of functionality we can customize. `JPanel` inherits from `JComponent` which has a method `paintComponent`. By overriding this we can decide how we want our drawing to function. Another example is the `getPreferredSize`. Since the canvas always is supposed to be square, we overrode this function making sure the size always is square when rezising the window. Inheriting from `JPanel` in `drawManager`, `toolBar` and `colorBar` let us write our own customized mouse and key listeners, giving us the behaviour we want.

We also made use of inheritance in the History Support. The History support is presented in a dialog box. We have made our own class `HistoryDialog` that inherits from `JDialog`. We then get to design it as we want with content and layout. In addition, we can decide the behaviour in the program when a selection is made (preview) or the box is closed (update with a new image from history).

5.4 Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. (*Java Polymorphism*, n.d.) We have made great use of polymorhism in our code. All the different shapes are a subclass of `Shapes` and we have overridden methods to suit each subclass. One example from our code is when we are drawing the image on the canvas. The squareImage object contains a list of all the shapes drawn on it. The list is of type `List<Shapes>`. Iterating through this list calling the function `draw` for each object will call the different implementations of draw depending on what subclass the object belongs to. When we are saving an image as a VEC file, we again iterate through this list and call the method `toVecFormat`. The text string will then be constructed based on the subclass.

We use the same concept when the figure is drawn by dragging the mouse. The `drawManager` has a `Shapes` variable called `currentShape`. `currentShape` can be any of the subclasses, depending on the tool chosen by the user. When the method `resize` is called, the behaviour depends on the subclass.

By using polymorphism we don't need to handle each shape as a different case. This saves us for a lot of redundant code, which again increases the readability.

6 How to use our software

Our software is easy to use for normal people as well as for experienced programmers. Once started the program, the user can choose which shape to play with by clicking the corresponding button at the top of the program.

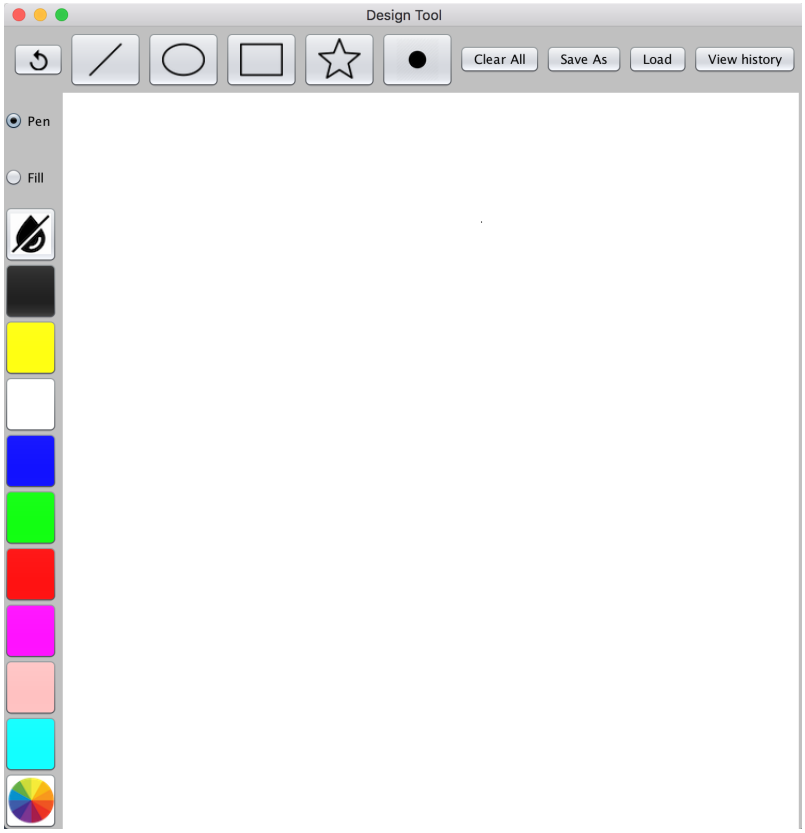

Figure 6: Program

A quick selection of colors for the user to use is found on the left side, but any other color is possible by clicking the color chooser at the bottom of the quick selection palette.

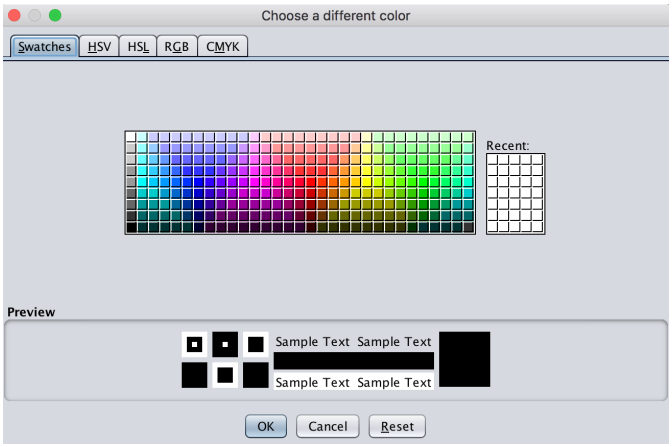

Figure 7: Color chooser palette

After selecting the desired shape and color, the user can start drawing on the white canvas. The user can at any time change both color and shape.

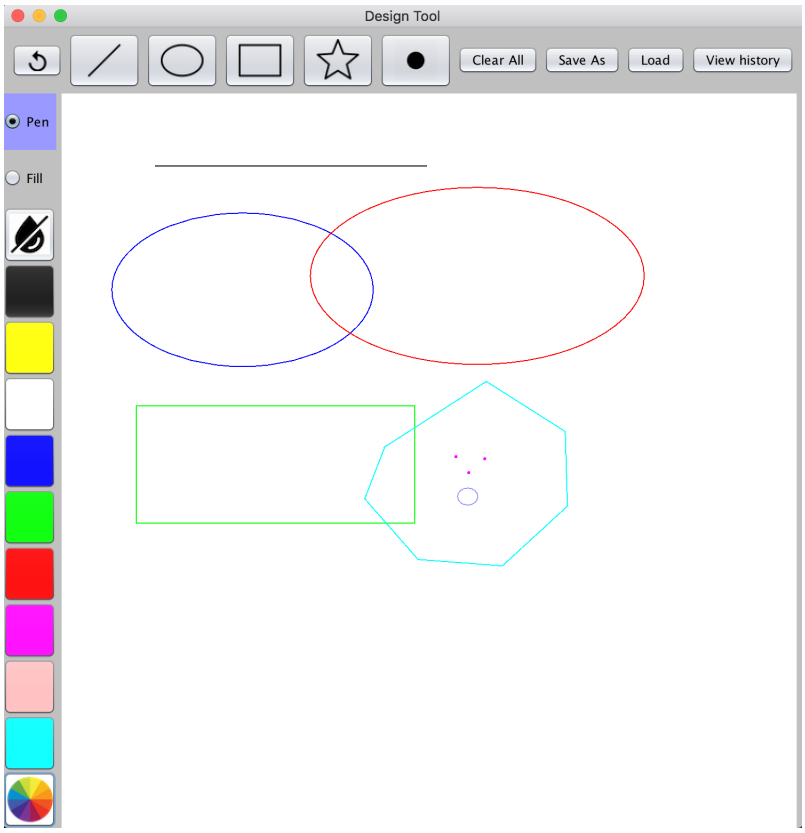

Figure 8: Shapes

If the user want the future drawings to be filled with color, this is done by first selecting the `fill` box to the left followed by selecting the wanted fill color. The color chooser can also be used to obtain the desired fill color. The user can change fill color for all the shapes, except from plot - which are only affected by the pen color selection.

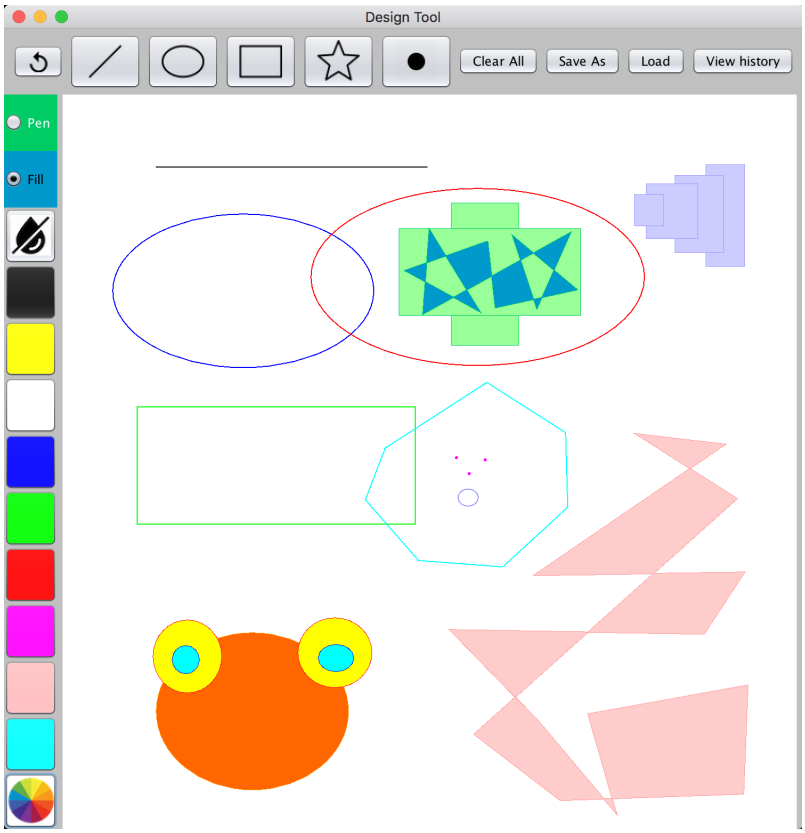

Figure 9: Filling

Drawings can easily be undone by clinking the `undo` button in the top left corner, or by using the CTRL+Z command. The last drawn shape will disappear from both the canvas and the VEC file. The user can keep undoing drawings until the canvas is blank. If for some reason the user want to wipe out all drawings, the `Clear All` button can be used. This button resets the image, with a blank VEC file and canvas as a result.

The user can preview or go back to earlier versions of the drawing by clicking the `View history` button. All drawing shapes, together with pen and fill color, and different commands made by the user will be listed in a new window as shown in figure 10.

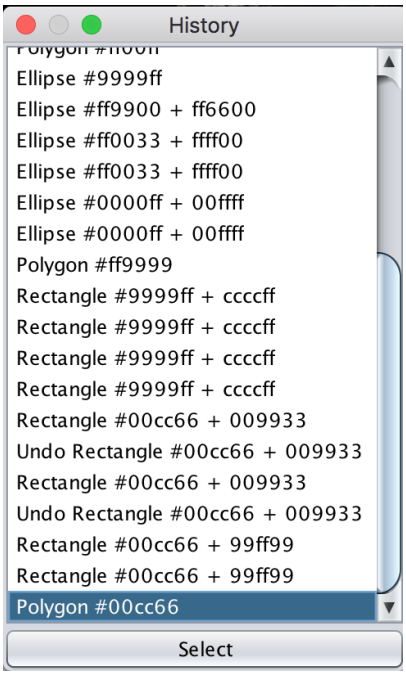

Figure 10: Drawing history

By clicking on the listings, the user gets a preview of how the image looked at the time. The listings are shown chronological, so the first listing will only show the first shape that was drawn. This way the user can go back and forth between versions and changes. Each event in the history is a new action the user has made. An example is shown in figure 11, where the user has selected an earlier version of the image. By clicking Select, the selected version will appear on the canvas and an "Edit from history" will appear in the history log as shown in the corner.

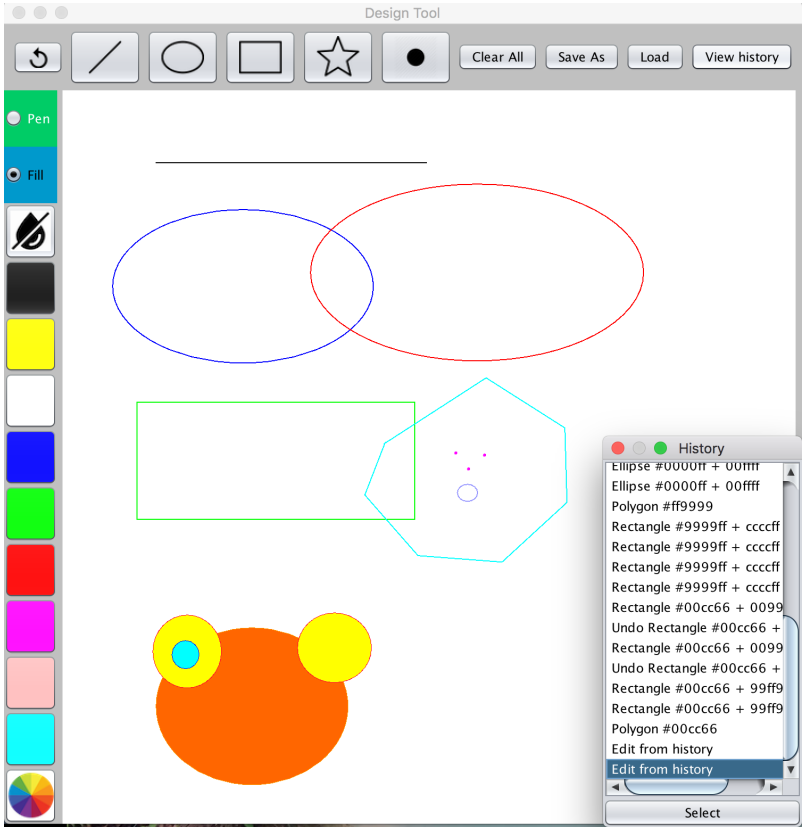

Figure 11: Earlier version of drawing

To save an image just press the `Save As` button, and a filesave dialog will appear. Choose a path and a name for your file and press save. The image will automatically be saved as a VEC file.

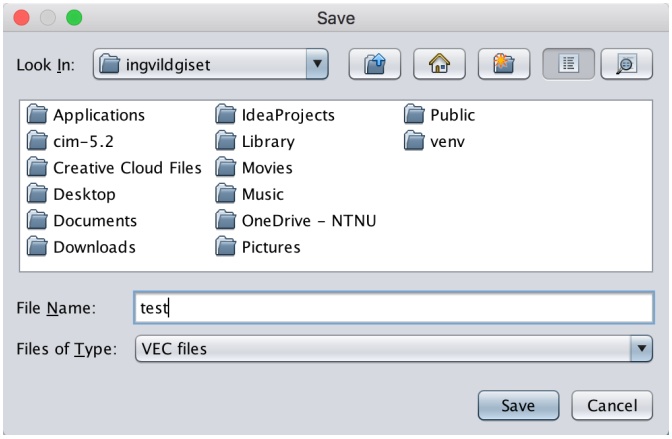

Figure 12: Saving image as VEC file

Loading a VEC file is done by clicking `Load`, where a new window pops up. The user can freely search between paths to find the wanted file, and click open to import it. It is only possible to select and load VEC files.

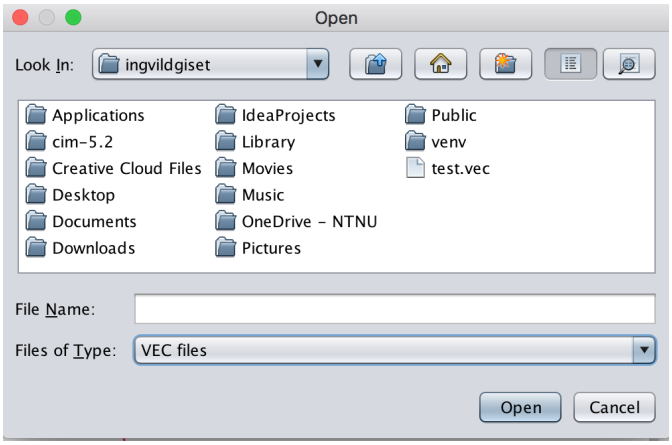

Figure 13: Loading an existing VEC file

If the canvas contains drawings and the user tries to close the program, a confirmation window appears asking if the user really wants to exit. This way the user can return to the system and for example save the file before quitting. The window will not appear it the canvas is blank when exiting.

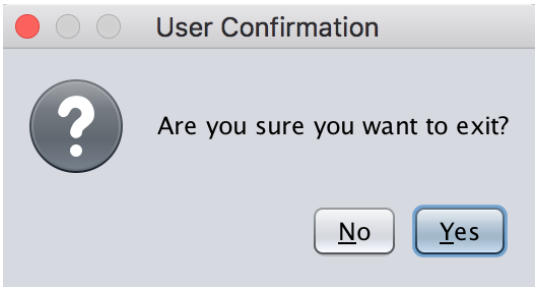

Figure 14: Exiting the program

References

Java encapsulation. (n.d.). Retrieved 2019-05-27, from https://www.tutorialspoint.com/java/java_encapsulation.html

Java inheritance. (n.d.). Retrieved 2019-05-27, from https://www.w3schools.com/java/java_inheritance.asp

Java polymorphism. (n.d.). Retrieved 2019-05-27, from https://www.tutorialspoint.com/java/java_polymorphism.htm