Report for Paint++

Henrik T, Anthony T, Hampus E, August S

28/10-2018
3.0

1 Introduction

The project aims to implement a raster graphics editor program which allows users to create and edit images. In today's society there are high expectations on both the quality and quantity of graphics. To produce illustrations at a high pace, and at the same time retain a satisfying quality level, would be a difficult task through physical means. For that reason, our program will contain digital tools and features which allow the process of photo editing and creating art in general to be considerably more efficient. Transformations such as scaling, translating or rotating an image or selection can be performed with less required time and effort, whilst producing a precise result. The purposes of a graphics editor range from e.g. photo editors enhancing the quality of a picture through modification of attributes, to artists creating by using digital brushes in combination with graphics tablets.

General characteristics:

- The application will initialize a project with a blank canvas on startup.
- The user will be able to locate and open an existing project at any time.
- The application will use a single consistent GUI similar to Photoshop/Paint.
- The application will be used offline.
- The application will let the user choose and use a single tool at a time.

1.1 Definitions, acronyms, and abbreviations

GUI - Graphical user interface

MVC - Model, view and controller

FXML - a xml-based language for building a user interface using JavaFX components

WriteableImage - a JavaFX class containing an image which a developer can edit. Put in an JavaFX ImageView class to display the image

2 Requirements

2.1 User Stories

Story Identifier: Epic 1
Story Name:
Description: (EPIC) As an artist, I want to draw and paint pictures through digital means because it is more efficient in terms of availability of tools and photo editing features.
Confirmation:
Functional requirements:

- User can use the program to draw art with different colors and use some features to edit photos.

Non-functional requirements:

- Performance-wise the application should work as a established raster picture editor such as Photoshop and Paint.

Story Identifier: Story 1
Story Name: Draw on canvas
Description: As a user, I want to draw on a canvas to create art.
Confirmation:
Functional requirements:

- Can I select a pencil to draw on a blank canvas?

- Can I press down the left mouse button onto the canvas to draw?

Non-functional requirements:

- Performance-wise the pencil should be able to draw fine strokes with low latency.

Story Identifier: Story 2
Story Name: Toolbar
Description: As a user, I want to have a toolbar for easy access to my tools to work efficiently.
Confirmation:

Functional requirements:

- Can I see the toolbar?

- Can I select tools and use them?

Non-functional requirements:

- Extensibility-wise it should be relatively easy to add more tools to the toolbar.

Story Identifier: Story 3
Story Name: Draw in different colors
Description: As a user, I want to draw in different colors to enhance my images.
Confirmation:
Functional requirements:

- Can I choose between different colors when drawing?

- Can I enter colors with hex-code?

Non-functional requirements:

- Choosing between colors should be effortless and easy.

Story Identifier: Story 4
Story Name: Opacity
Description: As an artist, I want to draw with adjustable opacity to reflect the real life drawing process.
Confirmation:
Functional requirements:

- Can I adjust opacity when drawing?

- Can I layer paint strokes with lower opacity to create a paint stroke with higher opacity?

Story Identifier: Story 5
Story Name: Save project
Description: As a user, I want to save projects to avoid losing progress.
Confirmation:
Functional requirements:

- Can I save projects with the ”Save” feature in File menu?

- Can I save the project as an image with the ”Save” feature in File menu?

- Can I specify the location where to save the new created project the first time saving? Else the user has to use ”Save as” function to choose location.

- If I have not saved the project before, the ”Save” feature will act as a ”Save as” feature.

Non-functional:

- Reliability-wise the save feature have to meet the users/clients expectations. It has to be reliable in which the picture always can be saved, and the saved picture stores the correct data.

Story Identifier: Story 6
Story Name: Open pictures from desired location
Description: As a user, I want to open existing pictures from different locations/directories to continue working.
Confirmation:
Functional requirements:

- Can I open a picture by clicking on a button and locating the picture in the disk?

- Can I open existing images with the ”Open” feature in File menu??

Story Identifier: Story 7
Story Name: Line tool
Description: As an artist, I want to have line tools to draw straight and curved lines perfectly.
Confirmation:
Functional requirements:

- Can I select the tool?

- Can I press, drag and then release on the canvas to create a line between the ”press” and ”release” state.

- Can I choose size and color?

Non-functional requirements:

- Performance-wise the line tool, when dragged, should render the line dynamically without latency.

Story Identifier: Story 8
Story Name: Shape tool
Description: As an artist, I want to have a shape tool because it's hard to draw symmetrical shapes manually.
Confirmation:
Functional requirements:

- Can I select the shape-tool?

- Can I choose the desired shape to draw?

- Can I draw the shape on the canvas?

Story Identifier: Story 9
Story Name: Fill tool
Description: As an artist, I want to color certain areas instantly because it's time-consuming to do it manually.
Confirmation:
Functional requirements:

- Can I select the fill tool?

- Can I fill areas on the canvas with colors instantly by clicking on the area?

Story Identifier: Story 10
Story Name: Layers
Description: As an artist, I want to have different layers to avoid ruin previous paint strokes and/or divide into different pieces.
Confirmation:
Functional requirements:

- Can I add multiple layers?

- Can I edit each layers individually?

- Can I rearrange layers to decide the level of the layers?

- Can I draw on one layer without affecting other layers?

- Can I delete a layer without affecting other layers?

- Can I toggle the visibility of the layer on or off?

- Can I name the layers to identify what layer contains what?

Story Identifier: Story 11
Story Name: Undo Buffer
Description: As a user, I want to undo my mistakes so that I am free to try stuff and explore the program.
Confirmation:
Functional requirements:

- Can I undo everything that a tool has changed?

- Can I use a shortcut to undo?

Non-functional requirements:

- The undo function has to be reliable in that it gives an exact replica of the previous canvas state.

Story Identifier: Story 12
Story Name: Select tool
Description: As a user, I want to have a select tool to select an area to work with.
Confirmation:
Functional requirements:

- Can I select the select tool?

- Can I drag on the canvas to create an area to edit?

- User can only edit in the selected area.

- Can I deselect the selected area.

Story Identifier: Story 13
Story Name: Eyedropper tool
Description: As an artist, I want to have a eyedropper tool to easily swap between color nuances.

Confirmation:
Functional requirements:

- Can I select the eyedropper tool?
- Can I set the current color by extracting a color from a specific pixel on the canvas?
- Can I draw with the extracted color?

Story Identifier: Story 14
Story Name: Zoom
Description: As a user, I want to have a zoom feature to view smaller areas on my canvas.
Confirmation:
Functional requirements:

- Can I zoom in and out when using the zoom feature in the view menu?
- Can I zoom in and out when using the magnifying glass in the toolbar?
- Can I zoom in and out to fixed percentages, e.g. 50, 100, 150 and 200 percent?

2.2 Definition of Done

- The code is compilable.
- All tests are successful.
- GUI implemented.

2.3 User interface

Figure 1 is a initial drawing of the user interface. It is inspired by the application Photoshop, where the toolbar is placed on the left side, canvas in the center, menu at the top and layers on the right.

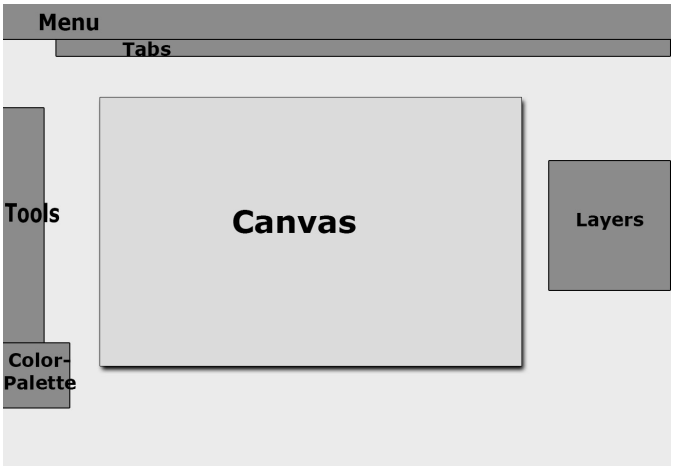

Figure 1: Show our initial sketch of the application

3 Domain model

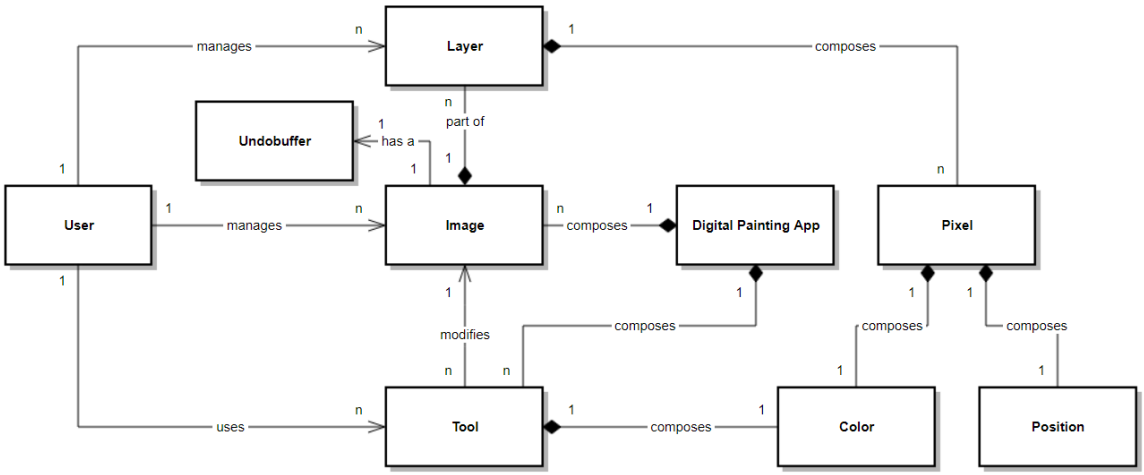

Figure 2: Simplified domain model

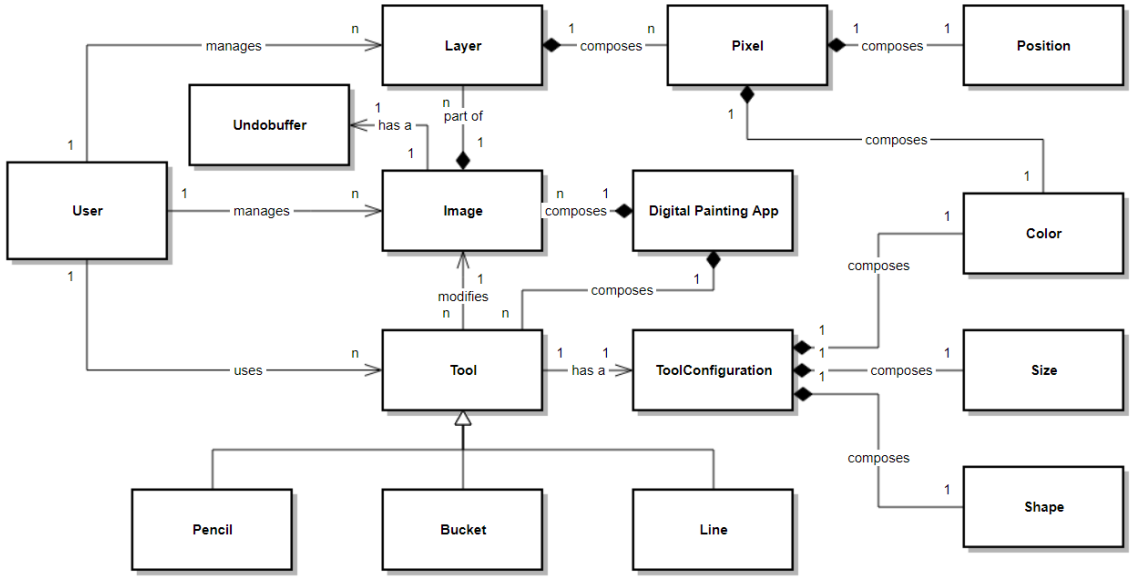

Figure 3: Complex domain model

3.1 Class responsibilities

Class: Digital Painting Application
Responsibility: The raster graphics editing application's responsibility is holding all the classes.

Class: Image
Responsibility: Containing the canvas on which the user edits. The image calculates the rendered canvas which is the result of all layers combined. It is also responsible for allowing the user to undo actions made on the canvas.

Class: Tool
Responsibility: An abstraction of all tools. Used for editing the canvas. Has a tool configuration which specifies the color, shape and size of the tools.

Class: Tool Configuration
Responsibility: Storing the color, size and shape of the tools.

Class: Pencil
Responsibility: Functionality for drawing pixels on the canvas with pixel perfect precision.

Class: Bucket
Responsibility: Functionality for filling an area of pixels on the canvas.

Class: Line
Responsibility: Functionality for drawing a perfectly straight line on the canvas.

Class: Position
Responsibility: Allows the pixels to be arranged in a certain way.

Class: Layer
Responsibility: Allowing the user to edit on several individual layers which together result in a canvas.

Class: Undobuffer
Responsibility: Containing old states of the canvas which the user can revert to by using undo function.

Class: User
Responsibility: The user which gives input to the application to create or edit an

image.

Class: Pixel
Responsibility: Consists of a position and color which allows the canvas to be composed of colors arranged in a certain way.

Class: Color
Responsibility: To allow the user to edit the canvas using different colors and opacity.

4 System architecture

Paint++ is a raster graphics editor, a software application that allows the user to create art and edit images. The overall structure of the application follows the MVC pattern and only requires a single computer to function. The MVC pattern is a software architecture pattern which divides the system into three main components, model, view and controller. The model is responsible for logic and data, and does not have dependencies on components which are not a part of the model. The view is responsible for the graphics of the software, it updates the graphics using logic and data from the model component. The controller is responsible for processing events, such as user interactions, which can result in changes in both the model and view. The software application starts by running the Paint++ executable file and stops by exiting or closing down the application window.

4.1 Subsystem decomposition

The program only runs on one computer with no online functionality and therefore there are no components such as server and client. Therefore, in this project, the MVC packages will be identified as components.

4.2 Model

The application is not a part of a client-server model because it does not communicate with an external server or system. It is a standalone application and therefore, in this case, the model, controller and view packages are the only components.

The application allows the user to create art and edit pictures. It is divided into four packages, model, view, controller and services. The model is responsible for managing data and logic. It is independent of the view, the user interface. The model is again divided into packages which are tools, pixel and utilities, see figure 4.

The tool package is responsible for the tools used in the application such as pencil, brush and fill tool. It contains all the tool classes along with specific methods of each tool. The pixel package contains classes which are necessary for many tools, e.g. a PaintColor class which handles color.

Both the view and controller are responsible for the user interface. The only thing the view does is to listen to the data updates from the model through the observer pattern to thereafter update the view. The controller is responsible for everything that has a connection to the FXML files and shortcuts.

The application follows MVC by letting the controller have instances of both the model and view. The model does not know that the view exists, however the view observes the model through the observer pattern. For the top level UML package diagram, see figure 5.

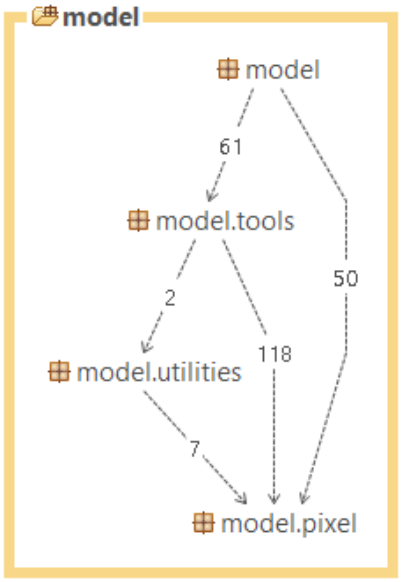

Figure 4: Model - Top level

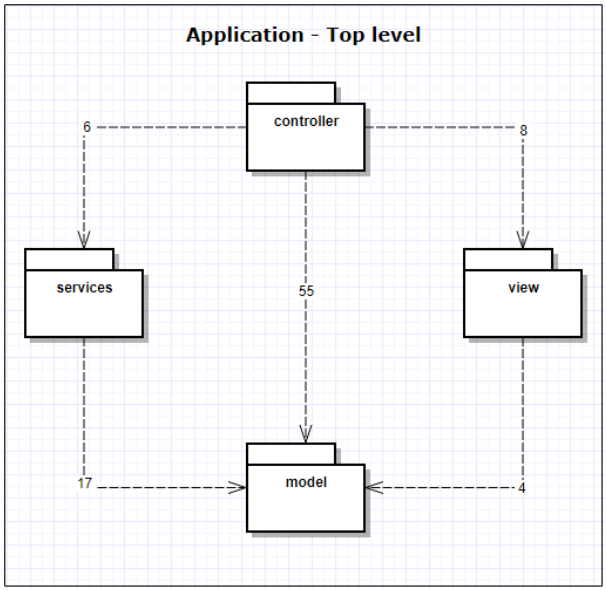

Figure 5: Application - Top level

4.2.1 Design model

Figure 6 shows the contents of the model package and the dependencies between the classes. It also shows which type of dependency each class has to the subpackages. When an external class wants to alter the model it will do so through the ImageModel.

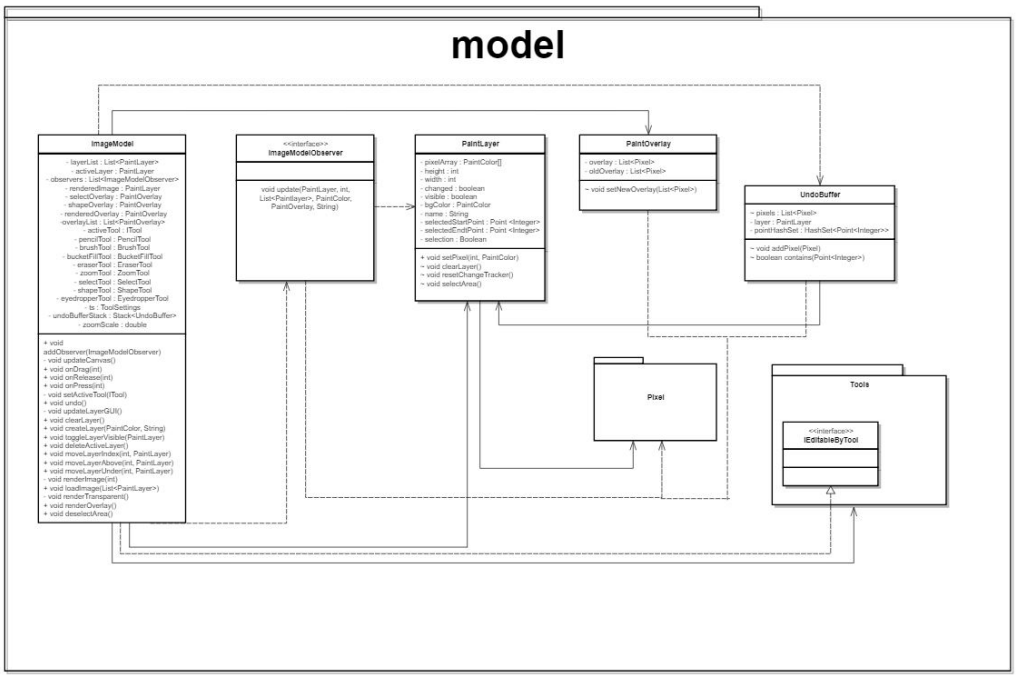

Figure 6: Design model, model

Figure 7 shows the subpackage called tools. There are both internal dependencies and dependencies which stretches to the other subpackage, pixel. This package also contains four strategies which aren't included in the image. They are used to get the algorithm for a line, ellipse, triangle and rectangle. Tools are central within the program since they are the users way of altering an image.

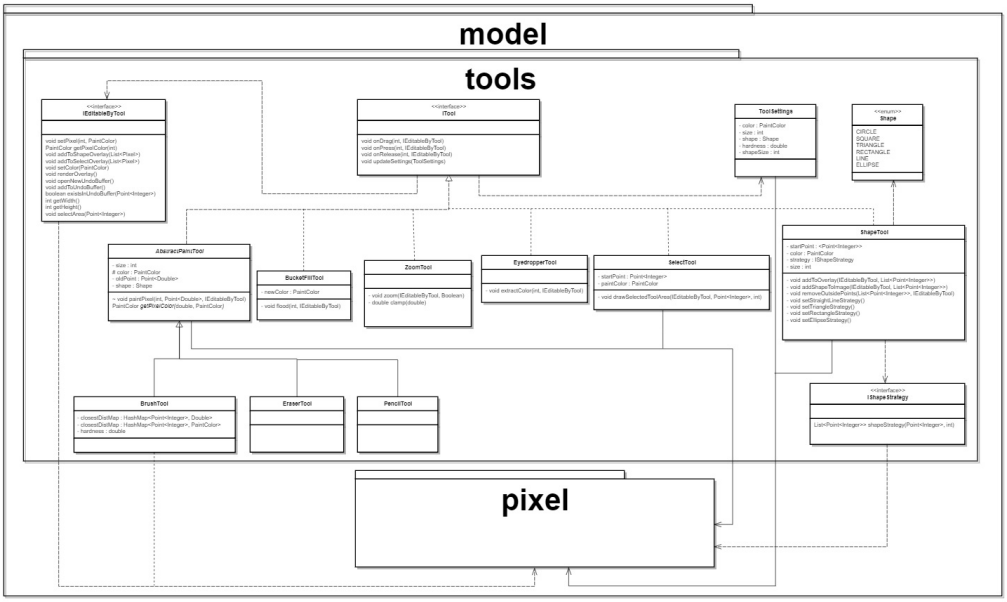

Figure 7: Design model, model.tools

Figure 8 shows the subpackage pixel. These classes have a high usage within the model and are used to represent parts of an image.

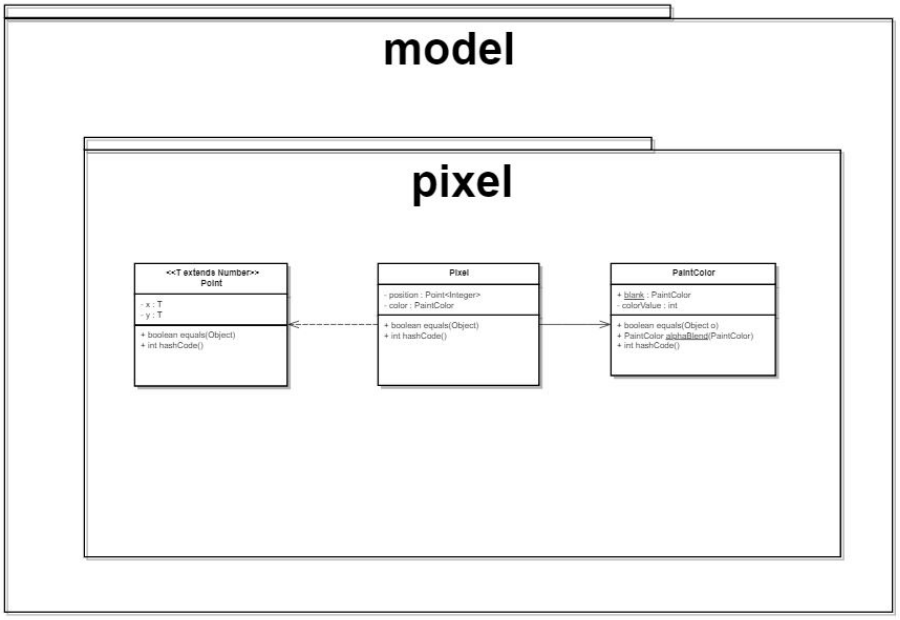

Figure 8: Design model, model.pixel

4.2.2 Model diagrams

In this section UML diagrams of the dependencies in the model package are provided. The diagrams show no circular dependencies, see figure 9, 10, 11 and 12.

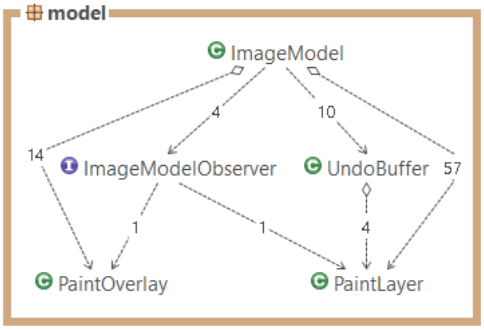

Figure 9: Model package dependencies

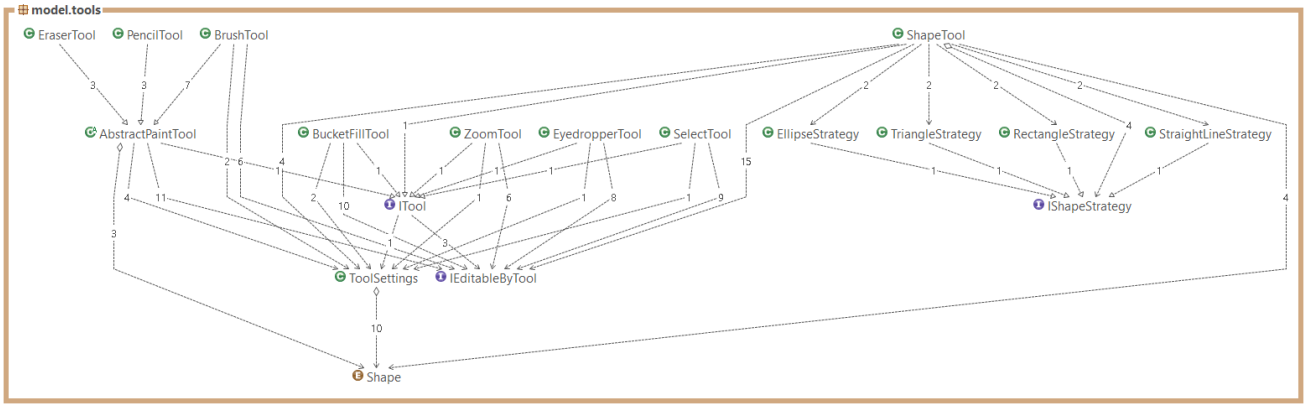

Figure 10: Tool package dependencies

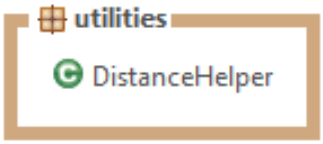

Figure 11: Utility package dependencies

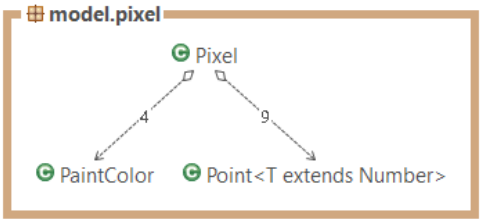

Figure 12: Pixel package dependencies

4.2.3 Sequence Diagrams

Sequence Diagram for creating a layer, see figure 13. The user creates a new layer using the controller, the controller lets the model handle the action. The view is updated and the user can see the layer.

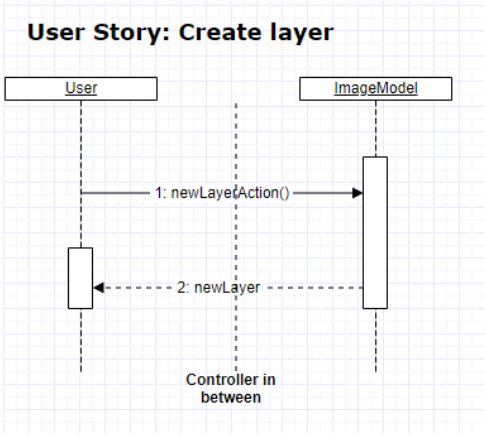

Figure 13: Flow for creating layers

The user selects a tool using the GUI, the controller receives the input and sends it to the model, updating the active tool. The user can then click and drag on the canvas to draw. The controller lets the model handle the onDrag events, which in turn calls the paint method in the active tool. The pixels of the image in the model is changed and updated, the view receives the update and renders it for the user to see. See figure 14.

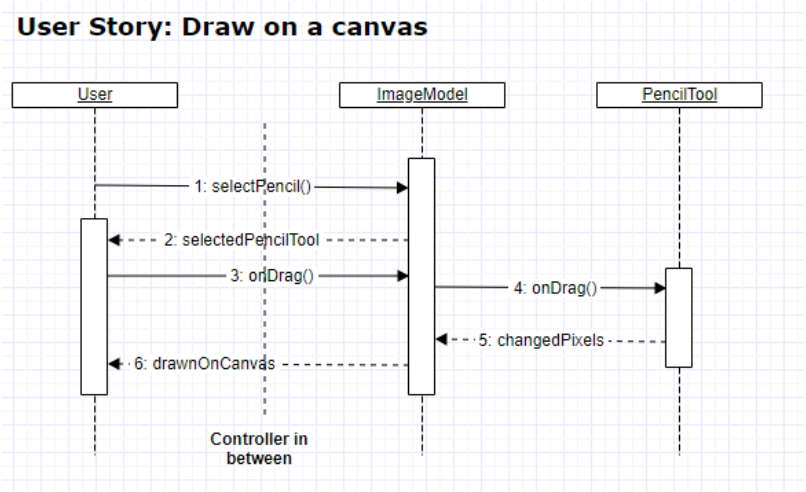

Figure 14: Flow for drawing on the canvas

4.2.4 Design Patterns

- Strategy pattern
- Chain of responsibility
- Observer pattern
- Template method
- Model-View-Controller pattern
- Facade pattern

Strategy pattern - used when creating different shapes with the shape tool.

Chain of Responsibility - used throughout the whole application, for example when the controller needs to draw on a layer which exists in the model.

Observer pattern - the view observes the model. Whenever the model is updated, the view renders the data in the model.

Template method - the abstractPaintTool defines a default base algorithm for painting a pixel, while the brush tool overrides the method to define a concrete behavior.

MVC pattern - the system architecture follows MVC pattern. The application is divided into 3 main components, a controller package, view package and model package.

Facade pattern - the tools alters the image through a facade called IEditableByTool.

4.2.5 Model quality

Property tests on different features in the application are located in the project map in src/test.

Analysis results from PMD:

Project:
Tda367-project
Scope:
Project 'Tda367-project'
Profile:
Default
Results:
Enabled coding rules: 83

- PMD: 83

Problems found: 18

- PMD: 18

Time statistics:

- Started: Wed Oct 10 23:41:56 CEST 2018
  - Initialization: 00:00:00.005
  - PMD: 00:00:01.558
  - Gathering results: 00:00:00.052
- Finished: Wed Oct 10 23:41:57 CEST 2018

Figure 15: First PMD result

Project:
Tda367-project2
Scope:
Project 'Tda367-project2'
Profile:
Default
Results:
Enabled coding rules: 83

- PMD: 83

Problems found: 6

- PMD: 6

Time statistics:

- Started: Sun Oct 28 22:40:43 CET 2018
  - Initialization: 00:00:00.004
  - PMD: 00:00:01.311
  - Gathering results: 00:00:00.041
- Finished: Sun Oct 28 22:40:44 CET 2018

Figure 16: Second PMD result

4.3 View

The view and the controller components share the responsibility of rendering the user interface. The view component is supposed to have most of the responsibility for the user interface, however, because of the project using FXML most of the responsibility of the interface is placed in the controller package.

4.3.1 View diagrams

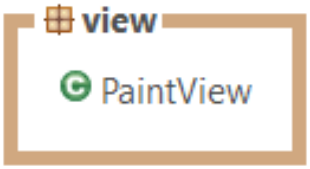

Figure 17: View subsystem dependencies

4.4 Controller

All user input in the application is handled by the controller module. The controller listens to the user and passes the action on to be managed by the model.

The controller module is divided into PaintController, LayerListController, LayerItemController, all of which have corresponding interfaces except for ShortcutController. The PaintController allows the user to choose tools from the toolbar and paint on the canvas. It also initializes the model, view and controllers. The LayerListController is responsible for interaction with the layer list. It allows for rearranging the order of layers and creating or removing layers. Each layer-item has a controller, allowing the user to change layer name, select the layer or toggle visibility. ShortcutController is responsible for the shortcuts in the application.

Controllers which are connected to FXML files act as views(all controllers except for ShortcutController). This means that the controllers, besides allowing the user to interact with the program through the user interface, also listens to the model and updates the views.

4.4.1 Controller diagrams

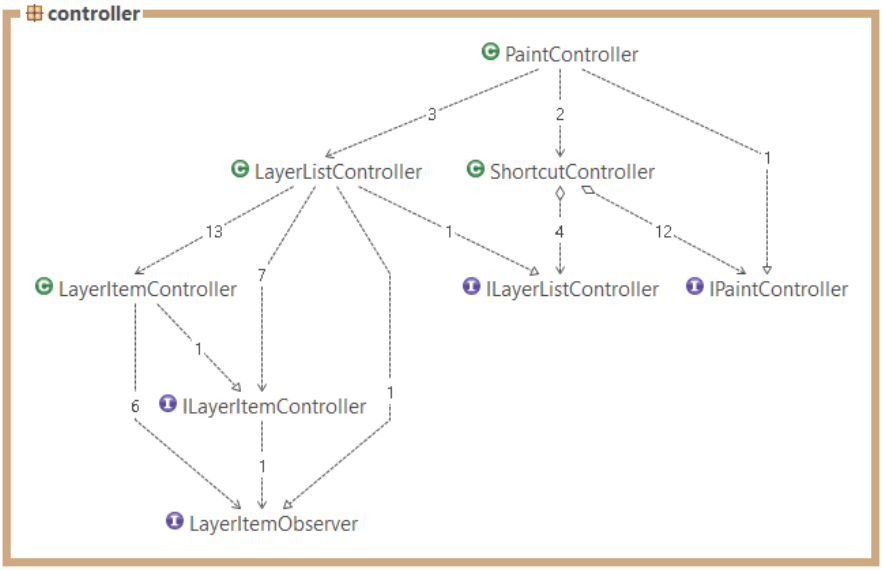

Figure 18: Controller component top level

5 Persistent data management

The only data management that has been implemented is when saving an edited picture. The saving process starts with creating a WritableImage from javaFX and then transfers all of the pixels from the edited picture to the WritableImage. Thereafter, a javaFX function is used to save the WritableImage to PNG format.

6 Access control and security

Paint++ is an application that can be used by anyone. All functionality is available to every role, therefore there is no difference between different roles using the application.

7 Known issues and possible improvements

There are several parts of the program that we had intended to improve. Unfortunately we could not find the time to perfect the current implementation. Although the program runs as it should there are parts of the design which are not optimal.

The main issue lies in the class ImageModel. This class has an undesirable amount of responsibilities and is something we have recognized throughout the process. If we had more time the first change would be to separate everything that has to do with tools from this class, and possibly some layer functionality. A separate class should also act as the administrator. This is not supposed to be done by the ImageModel. By dividing the wide responsibility of the ImageModel into individual classes, Single responsibility principle is achieved. This reduces the complexity of the class and the application becomes more maintainable. It will be easier to find functionality and data when they are separated into their own relevant classes.

8 Peer review (on group 21)

Single Responsibility Principle - There are parts of the program that do not align with this principle. Client controller is one example. This Controller manages the client model, the Service and Controller holds data which belongs in the model (e.g. userName, currentChatName). The method addElementsAfterIndex and the lists, friendList and blockedFriends, also belong in the model.

Open-Closed Principle - There is a lack of abstractions and since the program is heavily dependent on concrete implementations it's hard to extend. This is noticable in ChatLib where the Message class limits the rest of the program. If you want to add a new type of message, you'll have to modify in the MessageFactory and in the Service.read() method (and add in enum).

Dependency Inversion - This could be improved and would probably inspire a lot of other improvements. There are many dependencies on concrete implementations and working around this spawns a lot of the previously named issues. There is not a single interface in any of the model packages.

Does the project use a consistent coding style? - The project uses a consistent coding style which is apparent when looking at the indentations, naming of various variables and white spaces around operators. Furthermore, camel case is used when naming both variables and functions to simplify the process of understanding the code. JavaDoc is used to document the code, however description of parameters in some functions are missing.

Is the code reusable? - There are parts of the program that are reusable. But there are also parts which aren't. There are a few circular dependencies between packages in the code, for example between the server model and the server services, and also between the client controller and client services. Since these packages depend on each other they are hard to reuse in other code bases.

Is it easy to maintain? - A component is separated into packages and the responsibility of packages are divided into distinguishable classes. This makes it easier to pinpoint what functionality lies where and maintain it. However, some of the functionality are confusing, such as the command functionality. Because commands sent to the application are implemented as a type of message, for a new developer, it would be unapparent whether the method sendMessage is used as a command or to send a message to a user. This can be noticed in the login method in CEYMChatClient.Service class. Additionally, the circular dependencies between

controller and service makes some methods hard to maintain. A change in one method may require changes in the other classes methods.

Is it easy to add/remove functionality? - Adding functionality to the application is not difficult and rather straight forward. E.g. adding buttons. However, because of circular dependencies it will affect extensibility of service-related functionality.

Are design patterns used? - We have detected implementation of Factory, Facade, Command and MVC patterns.

Are proper names used? - Almost every variable and function have proper names which simplifies the understanding of the code. By reading the names of the variables or functions programmers get an accurate assumption of what the variables and functions are used for. E.g. saveReceivedMessages and loadSavedMessages.

Is the design modular? - Because of the wide responsibility of the controller, the design is less modular than it could be. The controller containing data and functionality which belongs to the model makes it hard to replace the controller without modifying the model class.

Is the code well tested? - There are tests which test the functionality in the client package, server package and model package. The tests cover the most core features of the application. However, most of the tests in the server package fails.

Are there any performance issues? There are quite a lot of performance issues. E.g. sometimes messages do not get delivered properly, trying to attach and send a file gives a null pointer exception and getting two conversations for every user.

Does it have an MVC structure? - The program has an MVC structure, but perhaps it could be improved a little. For example, the controller in the client is responsible for a few things that might be better suited for the model to handle, such as keeping track of friends, checking if a friend is blocked or not and storing messages.

Can the design or code be improved? Are there better solutions? - There is room for improvements. For example, right now the only way the program stores messages is with the controller appending the messages to an editable textbox(which means you can manually remove all messages by editing the textbox). A better way of storing messages would be if the model held a list of messages in every conversation, and then the view would ask the model for that list of messages when the view needed an update.