System design document for Paint++

Anthony Tao, Hampus Ekberg, Henrik Tao, August S’o’lveland

21/10-2018
1.1

1 Introduction

This is a system design document which describes the system architecture of Paint++. It also describes how the application manages data, access control and security.

1.1 Definitions, acronyms, and abbreviations

MVC - Model-View-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 System architecture

Paint++ is a raster graphics editor, which is a software program that allows the user to create art and edit images. The overall structure of the program follows the MVC pattern and only requires one computer. 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. The model 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. Since it is a software application it starts by running the Paint++ executable file and stops by exiting or closing down the application window.

2.1 Subsystem decomposition

The program only runs on one computer with no online functionality, as such there are no other system components than the application itself.

2.2 Application

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 the application is the only component. The application allows the user to create art and edit pictures. The application 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.

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 file and shortcuts.

The developers are aware of that the controller has a too wide area of responsibility. The classes that qualifies as a part of the controller package are current not gathered in a package because of errors caused by unknown reasons.

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, look at figure **??**. Currently, there are no interfaces for the controller, the view, nor between them.

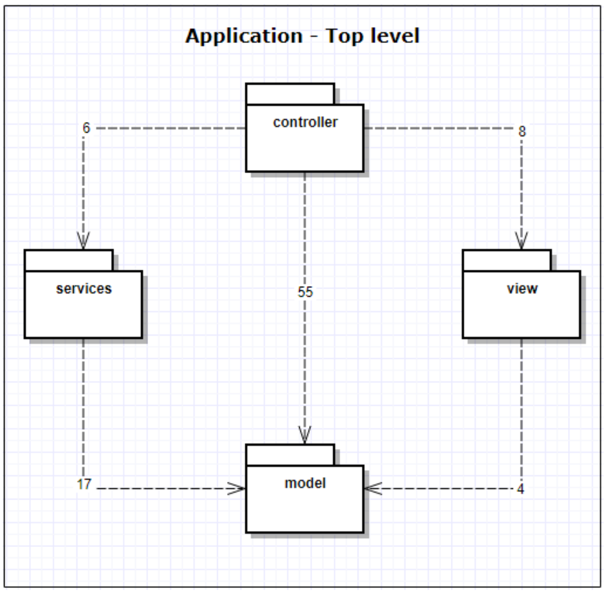


Figure 1: Application - Top level

2.2.1 Application diagrams

In this section UML diagrams of the application dependencies in each package are provided. The diagrams show no circular dependencies, see figure 2, 3, 4, 5, 6, **??** and 8.

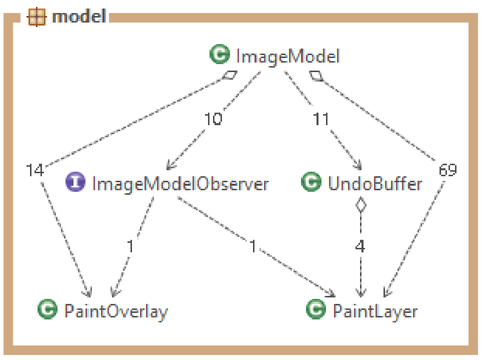


Figure 2: Model package dependencies

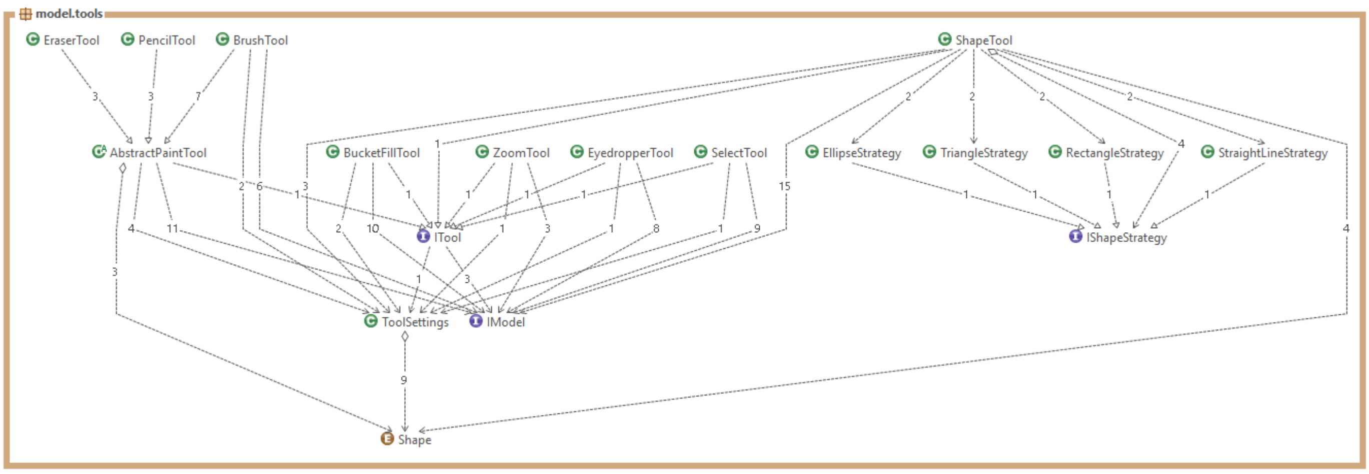


Figure 3: Tool package dependencies

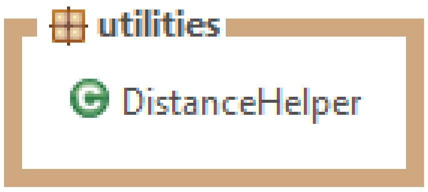


Figure 4: Utility package dependencies

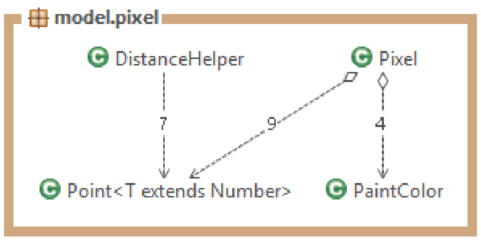


Figure 5: Pixel package dependencies

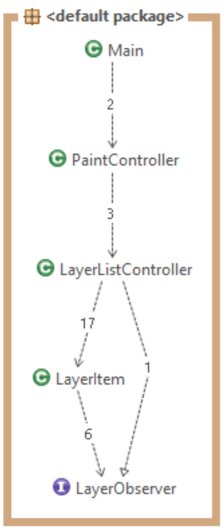


Figure 6: Controller package dependencies

view
PaintView

Figure 7: View package dependencies

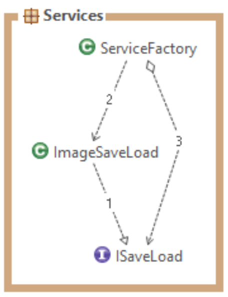


Figure 8: Services package dependencies

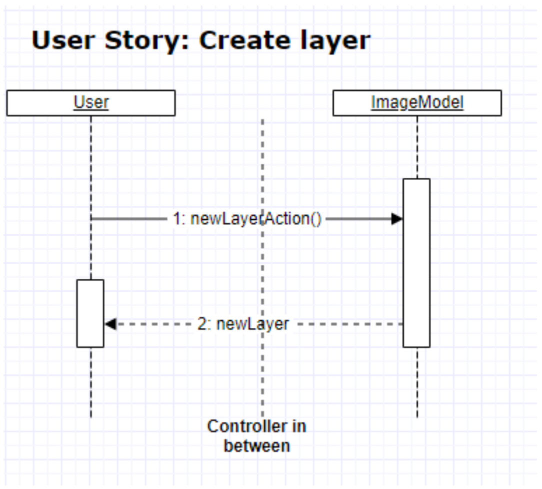


Figure 9: Flow for creating layers

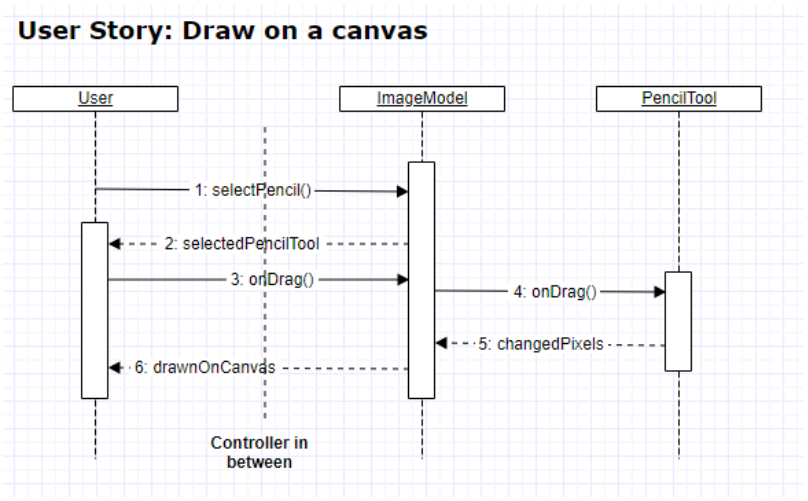


Figure 10: Flow for drawing on the canvas

2.2.2 Design Patterns

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

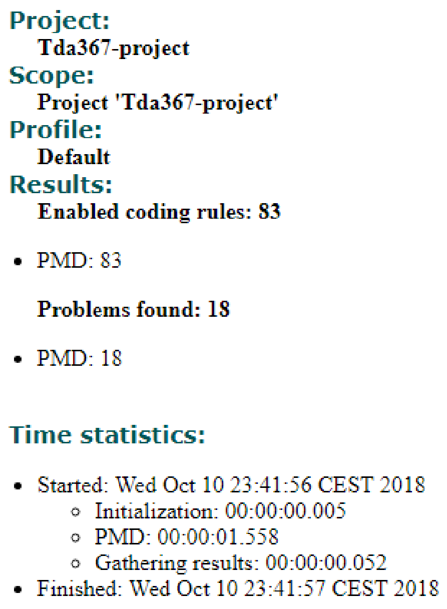
Observer pattern - the view observers the model. Whenever the model is updated, the view renders the image in the model.

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.

2.2.3 Model quality

The tests are located in the project map in src/test.



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 11: First PMD result

3 Persistent data management

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

4 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.

5 References