Final Report for

PaintIT

Henrik Lagergren, Markus Pettersson, Aron Sjöberg,
Ellen Widerstrand, Robert Zetterlund

28/10/18

v.1.0

Contents

1. Introduction

The project aims to create a desktop version of the popular mobile game "Draw Something" developed by OMGPop[1]. It is a two player game where one player draws a given word and the other player then guesses what word that has been depicted. The game is simple, fun and suitable for all ages. The purpose of the application is to bring people together with this interactive offline experience while enhancing creativity in people. Furthermore the game is very educational for children, both for spelling and drawing.

1.1. General characteristics of application

PaintIT is an offline collaborative desktop application resulting in a gameplay consisting of users gathering around one computer and taking turns painting and guessing.

When starting the game, two players create a team. One of the players (from now on referred to as the guesser) is advised to look away whilst the other player (from now on referred to as the painter) is presented with three words, each with a different level of difficulty attached to them. The painter's task is to first choose one word out of the three that are presented and the painter is then supposed to paint the chosen word with enough accuracy that the guesser will be able to correctly guess the word. When the painter has finished the painting, it is time for the guesser to guess what the painting portrays using a set amount of tiles containing letters. The amount of points given at the end of each round depends on the difficulty level of the chosen word. Easier words are worth fewer points while harder words are worth more points, ranging from 1-3 points each.

The goal of the game is to collect as many points as possible during a game session. In order to keep gathering points, the players need to paint and guess the word within the given time constraint. The painter has no time limit to finish their painting while the guesser has 30 seconds to guess the correct word. Failing to guess the correct word within the set time will result in the end of a game session and the players receive the score of the points which they have accumulated throughout the session. The best scores are saved on the high score which can be viewed at any time from the main menu.

1.2. Definitions, acronyms, and abbreviations

Below follow words that are used throughout the working process.

1.2.1. General words used throughout document and development

- **UI** - User Interface.
- **Design pattern** - Design patterns as it is referred in programming.
- **Visual design pattern** - Visual design pattern as it is refereed in ”Designing Interfaces” [2].

1.2.2. Words explaining the game

- **Painter** - The player that is being presented with a word which he is supposed to paint on the canvas.
- **Guesser** - The player that is supposed to guess the word that the other player/the painter has painted.
- **Canvas** - The canvas contains the painting painted by the painter, which is also shown to the guesser.
- **Round** - A round consists of a word being chosen, painted and guessed, either incorrectly or correctly.
- **Game session** - A game session starts when 2 players have entered their names and ends either when they decide to quit or when they fail to guess the correct word in time.
- **Tiles** - There are eight tiles in total, each representing one letter. The guesser uses the tiles to guess the word that is depicted.
- **Streak** - The team's streak is the amount of points gathered from correct guesses throughout a game session.

2. Requirements

In this section the requirements for the application are presented. They are used as a guide throughout the process of creating the application.

2.1. User Stories

The requirements for the application are given from *User Stories* see Appendix A.1.

2.2. Functional Requirements

The user Stories can be summarised into a list containing functional requirements. Functional requirements and the user stories act as a guide of what to functionality implement throughout the project. Below is a summary of what players should be able to do, they should be able to:

1. Navigate around the main menu.
    a. Be able to read the rules.
    b. Look at the highscore
    c. Go to the game setup screen.
        i. Choose the player names
        ii. Choose timelimit.
        iii. Choose difficulty.
2. Play the game.
    a. The Painter should be able to:
        a. Paint on the canvas using the
            i. Brush
            ii. SprayCan
            iii. Eraser
    b. Choosing color to paint with.

c. Choosing the radius to paint with.

d. Clear the entire canvas.

e. Undo the latest stroke of paint.

b. The Guesser should be able to:

a. See the painted canvas.

b. See a selection of eight tiles.

c. See an empty word sequence.

d. Guess the word by:

i. Be able to create a guess from the given tasks.

ii. Removing tiles from guess.

c. At the end of the round select to start a new round or end game session.

3. Exit the application and save game-streak.

2.3. User interface

Navigation throughout the application uses UI Design patterns from "Designing Interfaces" written by Jenifer Tidwell[2]. The interface is created for its simplicity, primarily the interface suits the inexperienced user.

2.3.1. General Navigation

The general navigation of the application makes use of the keyboard and mouse. Keyboard shortcuts are available whenever necessary, for example when guessing the word: users can use the keyboard to add or remove letters from their guess. Given Figure 1 the user is able to press the E-key on the keyboard, or click on an E-tile with the mouse to complete the guess. Further, pressing BACKSPACE on the keyboard removes the rightmost letter from the guess (in Figure 1 that would be the letter T), and clicking any letter in the guess would put it back to the available tiles.

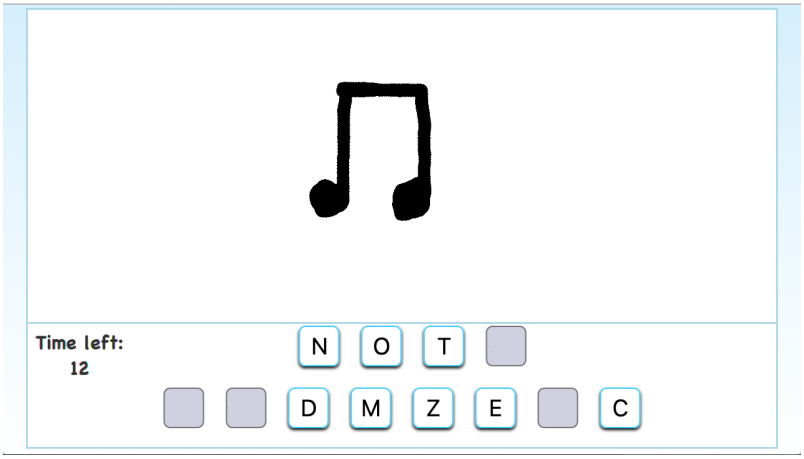


Figure 1: The GuessingView with an almost correctly guessed word

2.3.2. The painting aspect

Painting is one of the main spect of the game, below follows a summary of the iterations and design choices made for the painting view.

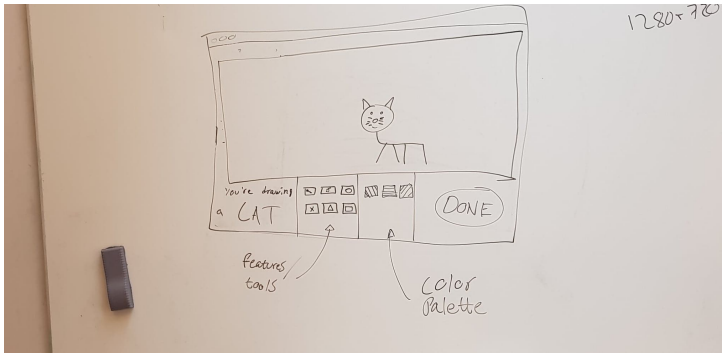


Figure 2: The first sketch of the painting view

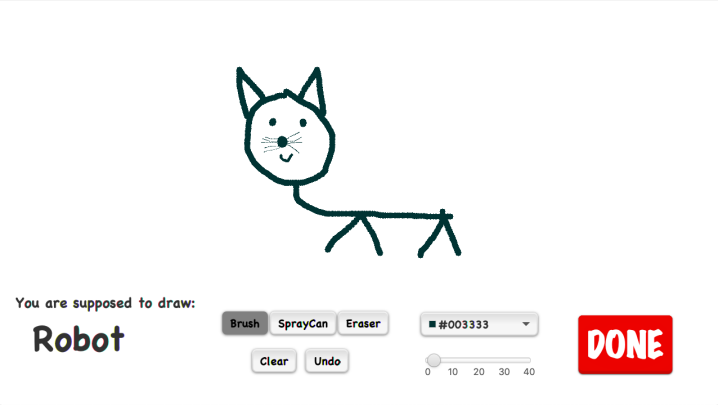


Figure 3: The first iteration of the painting view

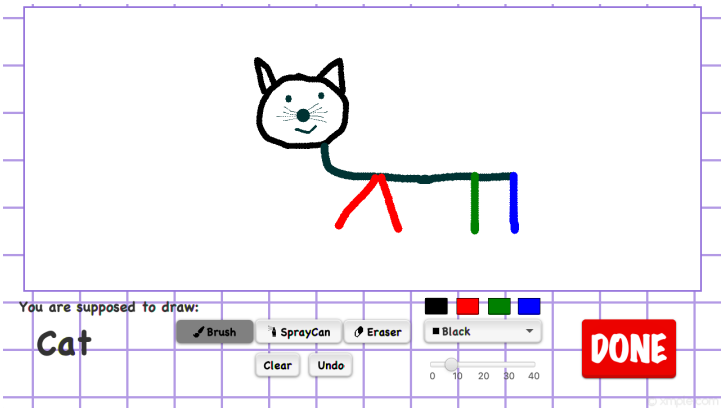


Figure 4: The second iteration of the painting view

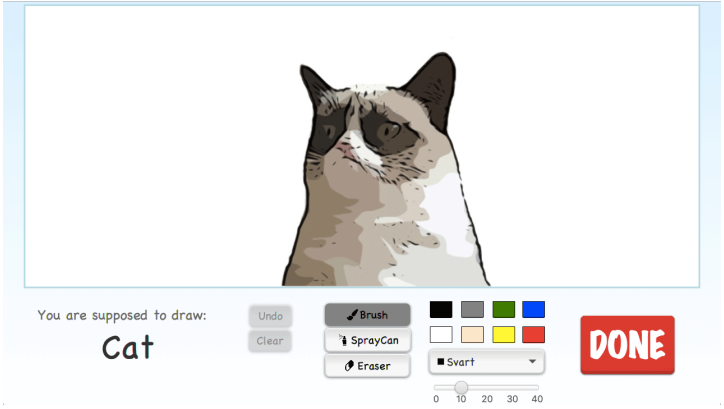


Figure 5: The final iteration of the painting view

The sketches for the painting view was created early on in the working process, see Figure 2, and acted as a template for how the view would be developed. The first iteration of the painting view, see Figure 3, gave us useful feedback, it was noted that it needed

to be more obvious where the canvas was, it needed to be more colour as well as the opportunity to select colours quicker. As seen in Figure 4, the boundaries of the canvas is clearer, it became possible to quickly select basic colors (via "Quick-Colors"), icons was added to the tools (brush, spraycan and eraser) and the view appears to be more coherent. In the final iteration, as seen is Figure 5, more Quick-Colors have been added, groups of button have been more organised and the background was changed to follow the visual framework.

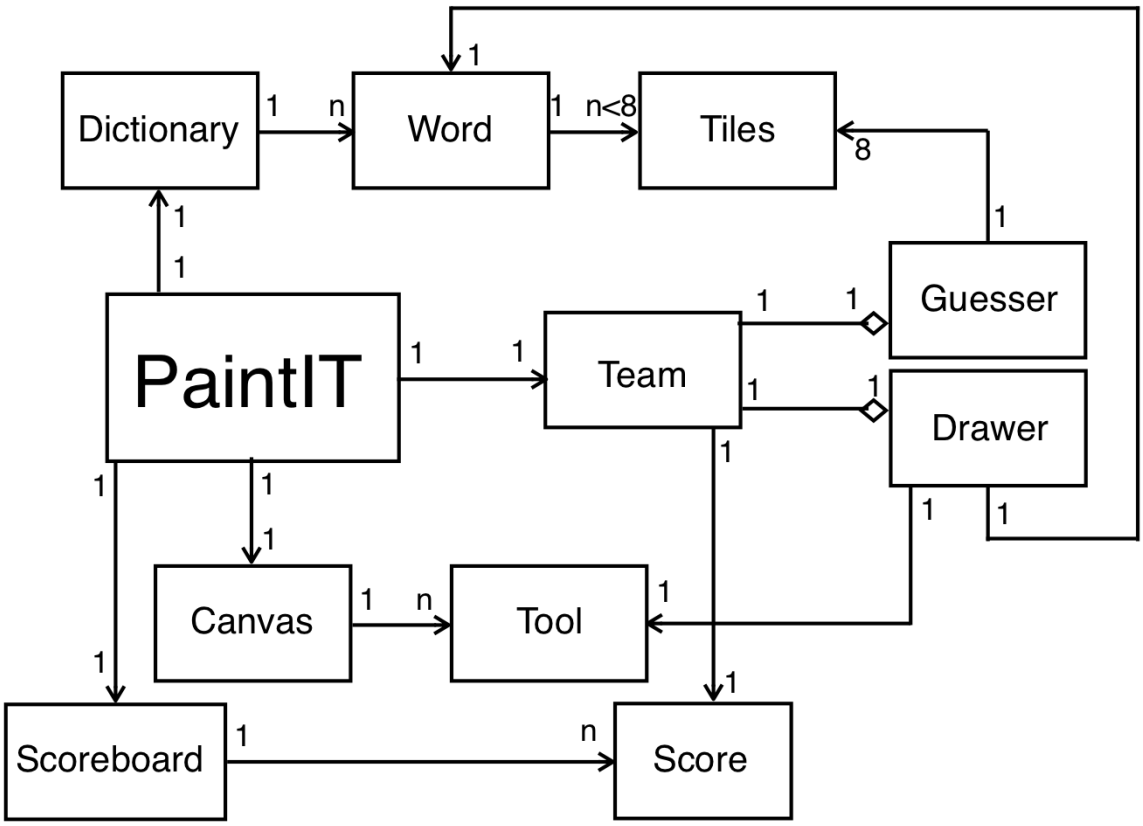
Below follow some of the design patters implemented in the paintingview:

- **Canvas Plus Palette** - Canvas Plus Palette is the basis on the entire view, it is implemented with a scarce amount of tools in an attempt to simplify the painting experience.

- **Prominent Done Button** - A promintent done button is placed in the right bottom corner, following the visual flow of the application, the button stands out and it is clear what will happen when clicked.

- **Button Groups** - Button Groups are used to arrange the different adjustments you can do to the tool and the canvas. Adjusting size, color and tool is placed on the righthand side while buttons for manipulating the canvas directly (by clearing or undoing) is placed on the bottom left, see Figure 4

3. Domain model



3.1. Class responsibilities

Canvas - The canvas is used by the painter to depict whatever the current word is. During a round of gameplay there exists only one instance of a model of the canvas, but there actually exists two different views of that model. One of the views belongs to the painter, on which he/she can paint on. The second view belongs to the guesser, but it has no functionality whatsoever, e.g. it can not be drawn on, only viewed.

Dictionary - Class responsible for supplying words of different difficulty from which the painter can chose from. The dictionary contains words from three distinct difficulties: easy, medium and hard. There exist no formal criteria when a word is being classified as either difficulty, it is totally up to the programmer(s) to decide.

ScoreBoard - The ScoreBoard holds a collection of 10 Scores, representing the 10 highests streaks ever achieved. A Score consists of a team name and a corresponding streak. ScoreBoard presents a collection of Scores as a list, sorted by streak in descending order

(highest - lowest).

Team - There is only one team playing at a time. The team has two players, a team name and a streak supposed to represent how well the team is doing.

4. System architecture

The application is a standalone program contained in one window being run locally on one machine. It does not depend on any external servers/clients or API calls. A main menu exists within the application, which gathers all different parts of the application in one place, so that the users can access them easily. From there the users can access information about the rules of the game, see the results of prior games and start a round of the game. A game starts by registering a team of two players. A game loop is then repeated, which consists of one player choosing a word and painting it and the other player trying to guess what that painting portrays. The game is visualized to the user through *GameScreen*s which are changed continuously throughout the game. After each round the user can decide whether to keep playing or exit the game loop.

4.1. MVC

PaintIT is an application that is heavily driven through interacting with a GUI and therefore a MVC-structure has been implemented in order to separate the data layer from the presentation layer and keep business logic away from the presentation layer. Due to the JavaFX library being somewhat problematic at abstracting logic from the view, it does exist some dependencies from the view to the controller.

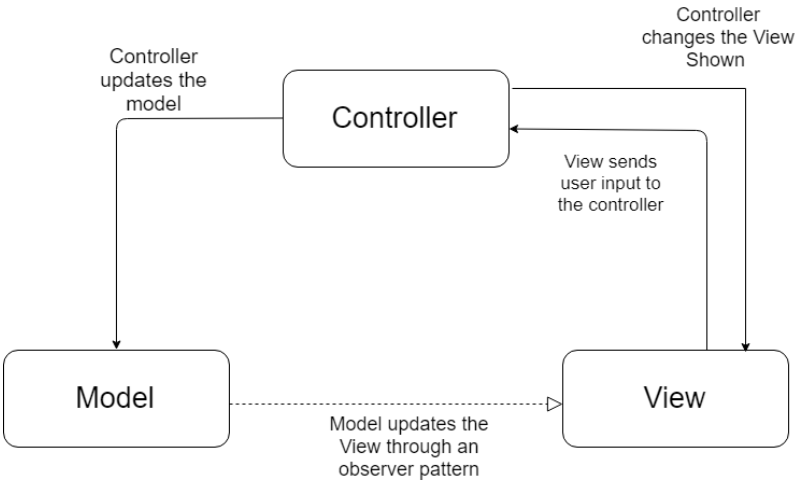


Figure 6: MVC implementation

4.1.1. Model

The Model package contains all classes representing a model in the game, e.g. it has a state and methods to modify that state. There are some models that contains data that

needs to be presented to the user via different views. There are three distinct types of models in this application:

- **Team** - Which holds data for the two players.
- **Canvas** - That stores and handles a 2D array of pixels - the painting.
- **WordAndGuess** - Which generates words each round and then handles process of guessing the depicted word.

All models are self contained in the sense that their internal state is free of any view and/or controller instance. Some models implements *Observable* and that can be read about in Section 2.1.4.

4.1.2. View

The View package consist of all the different views that are shown in the application. A view can consist of several nodes that are other views, but for the most part they do not. All primary views implements the interface *GameScreen*, which requires a view to have an initialization (*from now on shortened as 'init'*) method, as well as a method for retrieving a reference to the view instance, which is used by the ViewController when it is time to show that specific view. The init method updates the view when it is about to be shown, for example updating labels or starting a timer. All primary views are instantiated when starting the application, but some information needs to be updated during the game and that is where the init method is useful.

The player changes the view during the game by pressing one of the buttons in the current view. The use of prominent done buttons makes it clear to the players that they are going forward in the game [2]. The button knows which view that is supposed to be viewed next by retrieving an URL that points to the next view from the ButtonFactory. This URL is sent as a parameter to TopController's show method.

4.1.3. Controller

The controller-package's purpose is to call the methods in objects of Model package at certain stages of the application. It connects inputs from the user to methods in the model and controls which view that is shown to the user. The application has five different Controllers:

- **TopController** - Controls the state of the game and connects all the parts of the program together. TopController can be accessed from all the *GameScreens* which allows them to call for a change of the current view shown. This also

lets the buttons pressed in the views to alter the game state. For example the DoneView has a button which lets the user quit the game, and this is connected to TopController which initiates the gameOver() method.

- **ViewController** - Contains a list of all the *GameScreen*s and controls which one of them that is shown. The "show" method is called from the TopController.

- **GameLogic** - Serves as a wrapper for the Model, and is how the rest of the application gets access to the model's functionality. It holds the instances of the GuessLogic, CanvasModel and the HighScore which make up the entire Model.

- **TileBoardController and CanvasController** - Handles User inputs in the interactive parts of the game (the painting and guessing part). The JavaFX Panes listen for user inputs and then send the information as a String to the controllers where the input is handled and calls the corresponding methods in the Model package

4.1.4. Observer Pattern

To solve the problem with communicating data between a model and its corresponding view(s), while keeping the model isolated, an Observer pattern has been implemented. That way views can be set up to subscribe to a model and get updates whenever that model changes.

4.1.5. Factory pattern

All classes implementing the *GameScreen* interface is instanciated at runtime, as they need to be accessible as soon as the user starts to interact with the program. TopController is responsible for doing this initilization, but it delegates the responsibility of handling the views to the ViewController. To circumvent TopController's need of knowing of all the implementations of *GameScreen*s and relying on them directly, a Factory Pattern has been used. In combination with dependency injection, all classes implemeting *GameScreen* can receive a refernce to the same instance of TopController while TopController nor ViewController relies on their implementation. All implementations of *GameScreen* are simply passed to ViewController as a list of *GameScreen* objects and put in a hashmap, with their classname as the key, for quick lookup during runtime.

4.2. Design model

In order to improve visibility of the design model, every package's class diagram is displayed separately. The Model, View and Controller- packages (as seen in Figure 7, 8 and 9 respectively) have their responsibilities described above in the MVC section. The last package - the Util Package (as seen in Figure 10) - holds functionality which is used by all the other packages. For example it contains the Observer Interface, which is used by the Model and View Package to communicate via an *observer pattern*.

4.2.1. UML - Class Diagrams

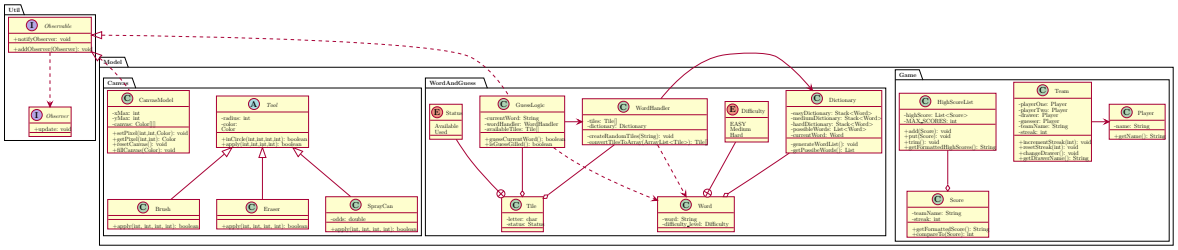


Figure 7: UML for Model package

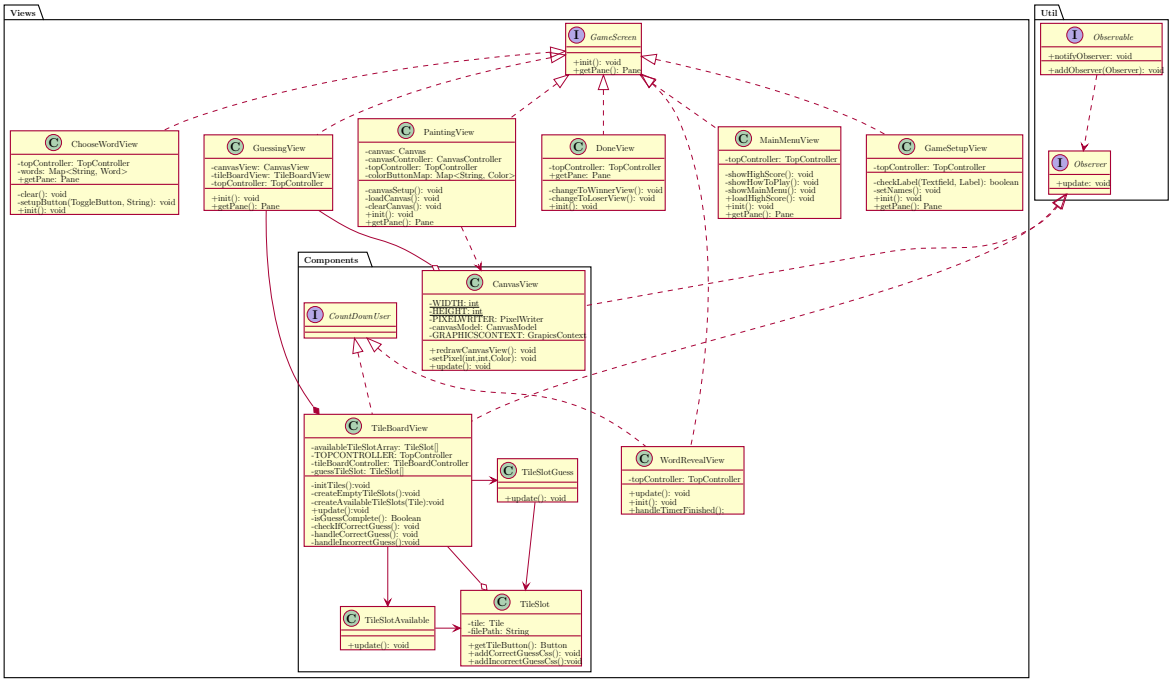


Figure 8: View package

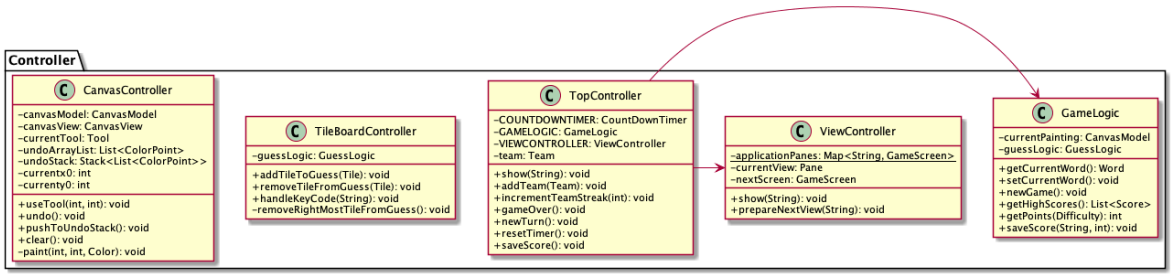


Figure 9: UML Controller Package

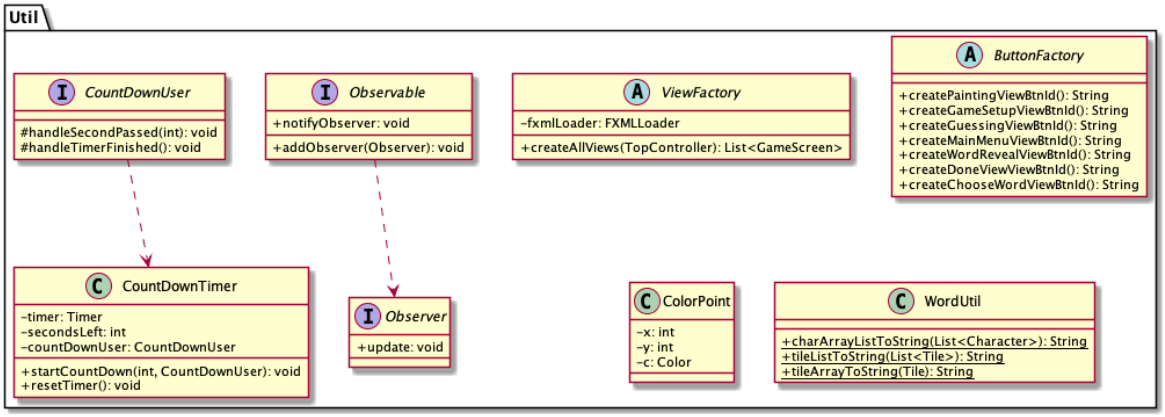


Figure 10: UML Util package

4.3. UI Design Patterns

Different design patterns from the book ”Designing interfaces” [2] has been implemented to improve the communication between application and user. The MainMenuview, GameSetUpView and WordRevealView has a ”prominent done button” to put emphasis on the primary or most important action. Further, Button-Grouping have been used in the paintingview so that buttons modifying the *Tools* are placed in close proximity to each other.

When starting the program, the user is faced with the Main Menu, a simple view with a clean design. The user can get information on how to play the game, view the high score and start a new game, through three different buttons.

4.4. Quality

Various techniques have been used to ensure a robust working process. The following segment covers how tests, version control, continuous integration and documentation

helped maintaining code quality throughout the development of PaintIT.

4.4.1. Tests

Tests have been written continuously during the development process to prevent unwanted changes to behavior of the program to appear unnoticed. The external library JUnit is used to test the classes and their methods. The project is reliant on the visual framework JavaFX which does not provide a built in test suite nor does JUnit provide support for writing test cases for interacting with a GUI. To supplement JUnit the project uses the external library TestFX, which enables testing of the interaction with visual parts of the program. The purpose of testing is to validate individual functions of the application and identify as many bugs as possible before release, but also to facilitate verification of refactored code. Where to find the tests in the project folder is shown in Figure 11.

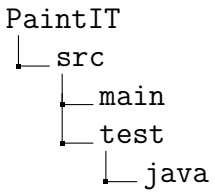


Figure 11: Figure showing where the tests are located

Figure 12 shows a test of some of the functionality of the CanvasModel. The tests of the CanvasModel are based on String comparisons, reliant on a toString()-method that have been implemented in canvasModel which returns a string-representation of the matrix.

4.4.2. Version Control

The VCS used is git, and the code is maintained on a remote repository hosted on GitHub. Using a workflow inspired by a blogpost by JonRohan [3], essentially revolving around branching from the development branch when implementing a feature or fixing a bug, finishing up, then merging the code additions and/or deletions back into the original branch.

```
@Test
public void testSetPixel() {
    // Sets canvasmodel to be a 2x2 matrix filled with color white.
    this.canvasModel = new CanvasModel(2,2,Color.WHITE);

    // Attempts to set pixel [1][1] with the color black.
    canvasModel.setPixel(1,1,Color.BLACK);

    // A comparison String where the array is represented in RGB
    // ranging in decimals from 0 to 1.
    String dummyCanvas =
            ''[ [ 1.0, 1.0, 1.0 ] [ 1.0, 1.0, 1.0 ] ] \n'' +
            ''[ [ 1.0, 1.0, 1.0 ] [ 0.0, 0.0, 0.0 ] ] \n'';

    assertTrue(dummyCanvas.equals(canvasModel.toString()));
}
```

Figure 12: Showing a commented test of the canvasModel.

4.4.3. Continuous Integration

Travis CI is used to verify the correctness of the code continuously throughout the project. Every time a new commit is pushed to the remote repository Travis builds the project and runs through the tests. Travis reports whether there was a problem with the build or not which prevents bugs from entering the code unnoticed.

4.4.4. Documentation

The codebase is documented with JavaDoc to improve understanding of the code while reducing friction when handing over code to another developer. This makes collaborative work easier since the functionality of classes is explained e.g. if you are not a Java compiler, reading about a method is often times easier than figuring out what the code does by yourself.

4.5. Quality tool reports

To complement runtime testing, three static analysis tools; FindBugs, PMD and STAN have been used throughout tbe development.

4.5.1. PMD

PMD is a source code analyzer that is used for finding flaws such as unused variables, empty catch blocks and the like [4]. The usage of PMD has aided us primarily by:

- Revealing when the program relies on implementation rather than abstractions, e.g. declaring an ArrayList instead of List. This is something that follows Dependency Inversion Principle (DIP).
- Showing unnecessary imports and notifying when fields can be made private, final or static.

However, PMD has also notified about violations regarding some coding conventions that was deemed irrelevant, for example, giving us a codestyle violation called "ShortVariable" for using "x0" as a variable name to represent the x-value at the origin of a cartesian coordinate-system, or complaining about violating the Law-of-Demeter when writing the following line of code (which is an example of method chaining) even though toString() will always have an implementation.

```
tile.toString().equals(keyCode)
```

The main takeaway from PMD is that some violations are intended to be suppressed and that the analysis should only be used as a reference.

4.5.2. FindBugs

FindBugs is a static code analyzer [5] that looks for bugs ahead of runtime, e.g. uninitialized fields that could cause a NullPointerException to be thrown. Some parts of the code will not play nicely due to variables being assigned a value at runtime.

The code snippet shown in Figure 13 captures a situation where FindBugs will raise a false alarm. Inside of the "MainMenuView.fxml" document is a declaration of a Button element with the id "play". The field "private Button play" (line 25) in the MainMenuView Java class will thereforee get assigned a reference to an instance of a Button object when fxmlLoader.load (line 48) is invoked at runtime. Findbugs can not know this when doing a static analysis of the code and will mark it as a potential error. Unlike the field "topController" (line 24), which FindBugs can determine will be assigned a reference to an instance of TopController (line 41) as soon as MainMenuView is instantiated.

```
22  public class MainMenuView extends AnchorPane() {
23
24      private TopController topController;
25      @FXML   Button play;
        .
        .
40      public MainMenuView(FXMLloader fxmlLoader, TopController topController) {
41      this.topController = topController;
42
43      fxmlLoader.setLocation(getClass().getResource(''/fxml/MainMenuView.fxml''));
44      fxmlLoader.setRoot(this);
45      fxmlLoader.setController(this);
46
47      try {
48          fxmlLoader.load();
49      } catch (IOException e) {
50          e.printStackTrace();
51        }
        .
        .
        }
    }
```

Figure 13: Showing a snippet of the class MainMenuView, illustrating the possible unitilized variables.

4.5.3. STAN

STAN (Structure Analysis For Java) [6] has been used to visualize structural design of the application. STAN is able to create a neatly arranged dependency graph which makes the application easier to understand. The benefit with a dependency graph is that developers are able to view the application from a bird's-eye perspective and, for example, remove unnecessary dependencies. Figures generated by STAN can be viewed in appendix A.1.

5. Persistent data management

Since the application is offline all of its resources are stored locally on the users computer. When inspecting the project folder almost all resources resides within the ”resource” folder two levels down from the root folder, see Figure 14.

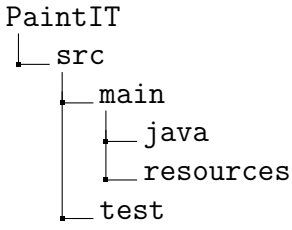


Figure 14: Figure showing where resources are located

Note: In the following paragraphs, ”subfolder” will refer to a folder within the ”resources” folder mentioned earlier.

The words used within the game loop are all stored in a JSON file in the subfolder ”dictionary”. Within the JSON file is a JSON object with an array property called ”dictionary”. Stored in this array are several JSON-objects, each denoting a word with a difficulty assigned to it.

All images used within the game is stored in a sub-folder ”images” with file format ”.PNG”. The images are used for different icons and banners throughout the game.

Highscores and instructions on how to play the game are stored in .txt files called ”highscores.txt” and ”instructions.txt” respecitvely. The ”highscores.txt” file is created if it’s the first time the user launches the application. This file will further be updated whenever a round of gameplay with a new highscore is finished.

6. Access control and security

The application does not have different privileges for different users, there is no data within the application that should only be visible to e.g. an admin.

7. Peer review

This is a peer review of group 13's project *Lagersystem*. Aspects such as how the code is structured as well as if it is easy to understand has been considered, among other things.

7.1. Maintainability

7.1.1. Code documentation

As it stands, the codebase is not very well documented. Only classes in the View package, besides two classes from other packages, contains any sort of JavaDoc. However, much of this documentation is redundant, as it covers things as getters/setters and not actual business logic. A lot of code throughout the codebase is commented out without any explanation as to why.

The classes *Admin*, *Writer* and *Reader* are identical, resulting in redundant inheritance, however, as noted in group 13's SDD, this is the foundation of an inheritence-based design based upon the following inheritance: *Admin* <: *Writer* <: *Reader* <: User. Refactoring core functionality in the User class will transfer to the subtypes, in some sense allowing for self maintaing code. However, this tight coupling makes it hard to refactor logic without introducing unintended side effects. It is mentioned that they strive for LSP, Liskov substitution principle, which is conceptually correct, but different roles should not be inherited, they should be delegated [7]. Therefore it could be argued that they should avoid using inheritance.

The views of the application is initialized using a method *start*, meaning no use of a interface markup language, resulting in a lot of code configuring the layout. This approach makes modifying the views difficult, meaning it reduces maintainability.

7.1.2. Names used

The names of the classes are fairly satisfactory, because of the functionality being based in the real-world. For example, the *Order* class is supposed to represent a real world order and the class does encapsulate what can be expected of an order pretty well, e.g. an order having an order number and data on which that order was placed.

7.2. Design Pattern implementation

The SDD states that the project is using a Factory pattern. The purpose of a factory pattern is to be able to avoid dependencies on specific implementations and instead rely on abstractions [8]. More specifically it should work as a facade to initialize objects of a certain family or sort, which allows for full control of what is shown to the rest of the program as well as decreasing dependencies. As stated in the SDD, the project is trying to implement a Factory Pattern. The implementation however, does not fit the description of the pattern. The only Factory class in the project is the CMSFactory and it does not provide any functionality for instantiating objects. It instead seems like it holds/handles all the observers and controls when their "onAction-methods" are called. This class name makes it more difficult to understand the code.

The Observer pattern implementation is also a bit difficult to get an overview of. The pattern usually indicates that there are two interfaces, an Observer-interface and an Observable-interface[9]. The implementation here only has an Observer interface and the CMSFactory sort of serves as the observable object for all the observers in the project. This makes it difficult to see what observers are called at what time, which could lead to unpredictable behaviour. A possible solution would be to create an Observable interface and have the desired classes in the Model-package implement it. This way an observer could subscribe to a more specific part in the model, which would make the program easier to understand and easier to Maintain/extend.

The code does have an MVC structure and the model is isolated from the other parts. The only way the model can communicate with other parts is via *observer-pattern*.

7.3. Tests

The Model-package is the only package which actually has been tested, with a method-coverage of 71%. The downside is that several of the written tests are for getters/setters, which might be unnecessary. Tests should aim to evaluate behaviour of the code, so instead of write tests for getters/setters should the focus be on real functionality. Tests for both the controller-package and view-package have opportunities for development with a coverage of 0%.

7.4. Security problems

Considering the program is an inventory managing system, where data should be immutable most of the time, variables that are "passed-by-reference" should be sparingly sent around, but it is not. For example, the class *Order* has the String OrderNr which is fetched using:

```
public String getOrderNr() {
   return orderNr;
}
```

Improvements could be made by ensuring immutability of an *Order* object by returning a copy of the variable. However, it is deemed unsafe to keep this implementation.

The application begins with a login-screen which only can get passed with a correct username and password. However, at the moment, it is possible to login with any random name and password. As a user you have the power to delete articles from the inventory. This is a security issue which does not make the application particularly safe to use.

References

[1] Wikipedia. ”Draw Something”. [Online]. Available: https://en.wikipedia.org/wiki/Draw_Something, retrieved: 2018-10-10.

[2] Jennifer Tidwell. *Designing Interfaces*. uppl. 2, Kalifornien, USA, O’Reilly Media, Inc, 2013.

[3] JonRohan. ”Dead Simple Git Workflow”. [Online]. Available: http://jonrohan.codes/fieldnotes/dead-simple-git-workflow-for-agile-teams/, retrieved: 2018-10-16.

[4] PMD. ”About PMD”. [Online]. Available: https://pmd.github.io/, retrieved: 2018-10-19.

[5] FindBugs. ”FindBugs - Find Bugs in Java Programs”. [Online]. Available: http://findbugs.sourceforge.net/, retrieved: 2018-10-19.

[6] Bugan IT Consulting UG. ”STAN - Structure Analysis for java”. [Online]. Available: http://stan4j.com/, retrieved: 2018-10-19.

[7] Alex Gerdes. ”Principles of subclasses”. [Online]. Available: http://www.cse.chalmers.se/edu/course/TDA552/files/lectures/tda552$_{l}ecture2 - 2.pdf, retrieved : 2018 - 10 - 23.$

[8] Tutorialspoint. ”Factory Pattern”. [Online]. Available: https://www.tutorialspoint.com/design$_{p}attern/factory_{p}attern.htm, retrieved : 2018 - 10 - 25.$

[9] Alex Gerdes. ”Observer Pattern och MVC”. [Online]. Available: http://www.cse.chalmers.se/edu/course/TDA552/files/lectures/tda552$_{l}ecture5 - 2.pdf, retrieved : 2018 - 10 - 25.$

A. Appendix

A.1. User Stories

The User Stories are listed with falling priority. The identifier can be found in the subsection numeration for each User Story.

A.1.1. Colour The Canvas

As an Artist, I need to be able to add colour to a canvas so that I can create art digitally.

Confirmation

- If I drag the mouse on the canvas, will there be color?
- If I drag the mouse outside the canvas does it not draw?
- Am I only able to draw on the canvas during the drawing-phase of the game?

A.1.2. Letters

As a guesser in the game I need to receive tiles to make a guess from so that the guess is a bit easier.

Confirmation

- Can I build the correct word with the given letters?
- Are the tiles easy to see and choose from?
- Can I see which Tiles that are included in my guess?
- Can I only use a guess tile once in my guess?

A.1.3. Guess the word

As a player, I need to be able to guess the word corresponding to my friends painting in order for us to proceed with the session.

Confirmation

- Can I use the keyboard to make a guess?
- Can I use the mouse and click to make a guess?
- Do I receive feedback whether the guess was correct or not?
- Can I Guess many times?

A.1.4. Start menu

As a player, I want to be greeted by a startmenu when I start the game so that I can start playing easily.

Confirmation

- Is a startmenu shown when the application starts?
- Does the startmenu direct the player so that he can start the game?

A.1.5. Finishing a painting

As a drawer in the game, I need to be able to finish a painting so that the guesser can make his guess.

Confirmation

- Can I my mark my painting as done?
- Will it be the other players turn if I mark my painting as done?
- Is the painting shown to guesser
- Is the painting unmutable after I mark it done?

A.1.6. See the finished painting

As a guesser, I need to be able to see the painting so that I can guess what it depicts.

Confirmation

- Can I see the painting on the canvas?

- Is the correct word not shown to the guesser?

A.1.7. Advanced drawing (EPIC)

As an artist, I need to be able to use different colors and tools in order to make a detailed drawing.

Confirmation

- Can I change the color which I am using?
- Can I change the drawing tool that i am using?
- Can I change the size of the tool I am using?
- Am I able to find the functionality above?

A.1.8. Erase

As an artist, I need to be able to erase what I have painted so that it is easier for me to correct myself when I have drawn wrong.

Confirmation

- Can I erase color from the canvas?
- Can I choose what parts of the canvas I want to erase?

A.1.9. Choose color

As an artist, I need to be able to choose color when I paint digitally so that I can draw with more detail.

Confirmation

- Can I change color when I am drawing?
- Are the colors easy to choose from?

A.1.10. Undo the canvas

As a painter in the game, I need to be able to undo my last action so that it is easier for me to make a good painting.

Confirmation

- Can I undo my last action?
- Can I undo all my previous actions?
- Can I undo using keyboard shortcuts?

A.1.11. Picking names

As a player, I need to have the ability to choose a name so that it can be shown in the game.

Confirmation

- Can I add name to the players?
- Is the name used to address the players throughout the game?

A.1.12. Countdown

As a player of this game, I need to get a countdown before the guessing-phase so that the guesser has time to prepare.

Confirmation

- Can I see how long time it is left before my turn starts?
- Do I have enough time to turn the computer around for the guesser?
- Is it clear when my turn starts?

A.1.13. Receive relevant words

As a player, I need to receive words that I am able to paint and guess.

Confirmation

- Are users able to paint the words given?
- Are the users enjoying to draw the words given?
- Are the user able to choose the difficulty of the word?

A.1.14. Scoreboard

As a competitive player, I want to be able to see all best scores on a scoreboard so that I can compete with them.

Confirmation

- Can I keep track of my score while the game progresses?
- Can I see other players highscores?
- Can I get on to the Scoreboard if I make a good score?

A.1.15. How to play

As a beginner, I need to get instructions/rules so I can play the game properly.

Confirmation

- Are the instructions easy to find at all stages in the game?
- Can a new player read the rules/instructions and understand how to play?
- Can I reach ”How to play” when I am playing a game?
- Can I go back from ”How to play” without restart the application?