System Design Document for

PaintIT

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

28/10/18

v.4.1

This version overrides all previous versions.

Contents

1. Introduction **1**
1.1. Definitions, acronyms, and abbreviations . . . 1
1.1.1. General words used throughout document and development . . . 1
1.1.2. Words explaining the game . . . 1
1.1.3. Implementation definitions . . . 2

2. System architecture **3**
2.1. MVC . . . 3
2.1.1. Model . . . 3
2.1.2. View . . . 4
2.1.3. Controller . . . 4
2.1.4. Observer Pattern . . . 5
2.1.5. Factory pattern . . . 5
2.2. Design model . . . 6
2.2.1. UML - Class Diagrams . . . 6
2.3. UI Design Patterns . . . 7
2.4. Quality . . . 8
2.4.1. Tests . . . 8
2.4.2. Version Control . . . 8
2.4.3. Continuous Integration . . . 9
2.4.4. Documentation . . . 9
2.5. Quality tool reports . . . 9
2.5.1. PMD . . . 10
2.5.2. FindBugs . . . 10
2.5.3. STAN . . . 11

3. Persistent data management **12**

4. Access control and security **12**

References **13**

A. Appendix **14**
A.1. STAN generated images . . . 14

1. Introduction

PaintIT is an interactive game for two players where one player, the painter, is given a canvas and a word to depict. The other player, the guesser, is shown the finished painting and has to guess which word is depicted. The following document describes the system design of the PaintIT application.

1.1. Definitions, acronyms, and abbreviations

Below follow words that are used throughout the working process.

1.1.1. General words used throughout document and development

- **MVC** - Model View Controller.
- **JavaFX** - GUI library for Java.
- **UI** - User Interface.
- **Design pattern** - General solution to a common programming problem.
- **Visual design pattern** - General solution to common problems concerning GUI development.

1.1.2. Words explaining the game

- **Painter** - The player that is being presented with a word which he/she 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. The game session also ends if they have guessed all the words correctly.

- **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.

1.1.3. Implementation definitions

- **Canvas** - Not an actual class. Umbrella term for the area which you can paint on. The Canvas consists of the CanvasModel, CanvasController and the CanvasView.

  - **CanvasModel** - The data representation of the canvas. The CanvasModel consists of a 2D matrix storing RGB values. CanvasModel implements the interface *Observable* and is subscribed to by the CanvasView.

  - **CanvasController** - The CanvasController recieves coordinates from the user via the CanvasView, and changes the model accordingly with regards to the equipped Tool.

  - **CanvasView** - The visual representation of the canvas. CanvasView extends the JavaFX Class Canvas and uses a pixelWriter to change pixels. Implements the interface *Observer* and is subscribed to a CanvasModel.

- ***GameScreen*** - An interface which allows a view to have a setup routine. Views implementing *GameScreen* occasionally contains other, smaller view components and acts as a 'main view' which can be accessed via pressing buttons throughout the game.

- **ViewController** - A controller that prepares and shows views. It stores an instance of all *GameScreen*s in a hashmap.

- **Tool** - An Interface which is implemented by classes who wish to be able to modify the canvas. A class implementing *Tool* has methods for modifying its radius, color and for applying its pattern which can be applied to the canvas.

- **Dictionary** - A class that generates and holds all words that the painter can choose from.

2. 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.

2.1. 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.

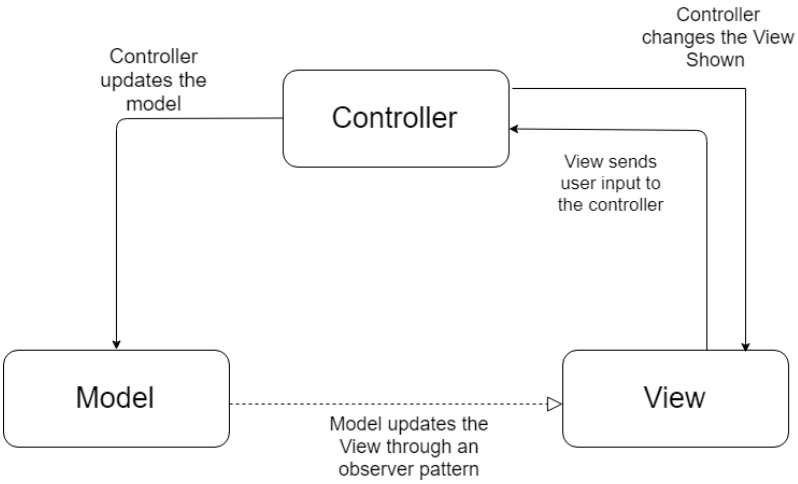

Figure 1: MVC implementation

2.1.1. 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.

2.1.2. 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 [1]. 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.

2.1.3. 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

2.1.4. 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.

2.1.5. 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.

2.2. 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 2, 3 and 4 respectively) have their responsibilities described above in the MVC section. The last package - the Util Package (as seen in Figure 5) - 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*.

2.2.1. UML - Class Diagrams

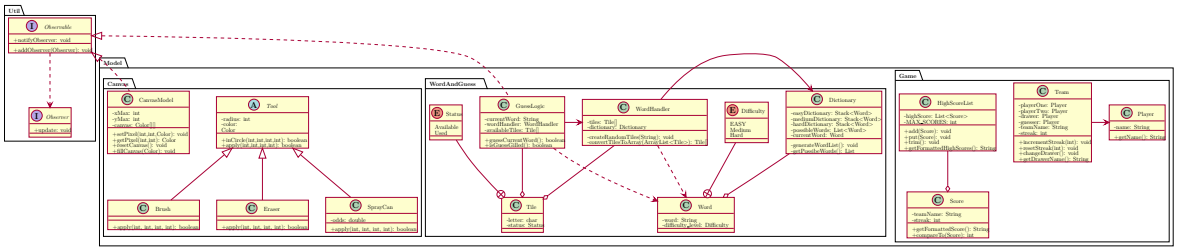

Figure 2: UML for Model package

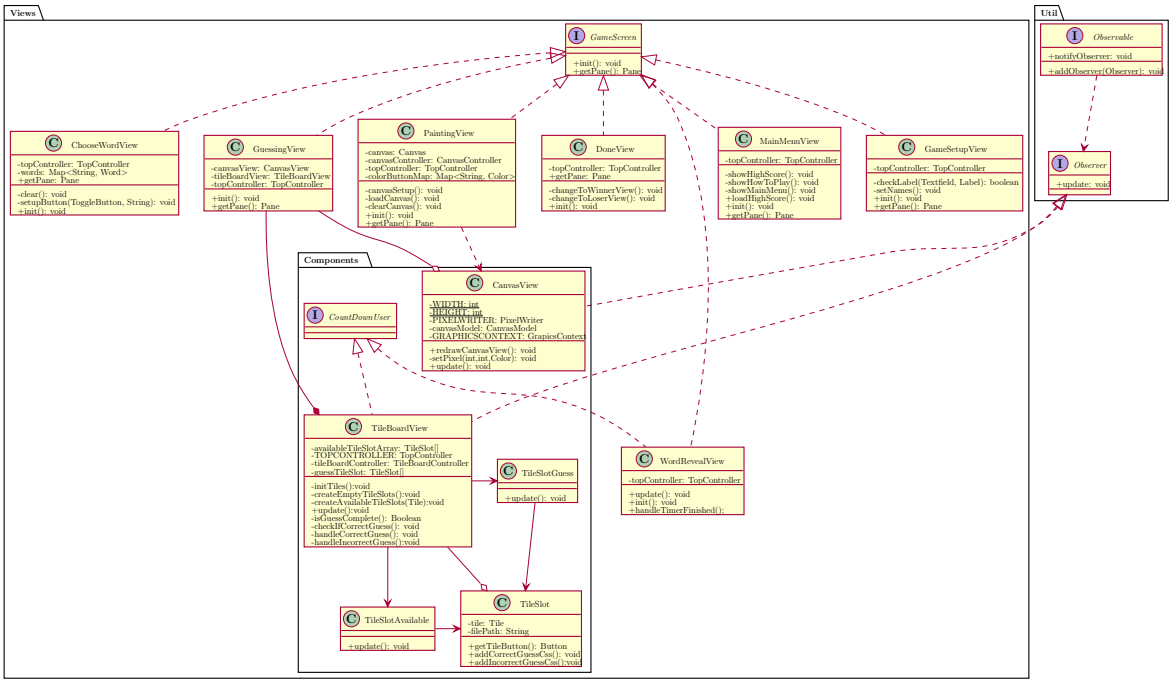

Figure 3: View package

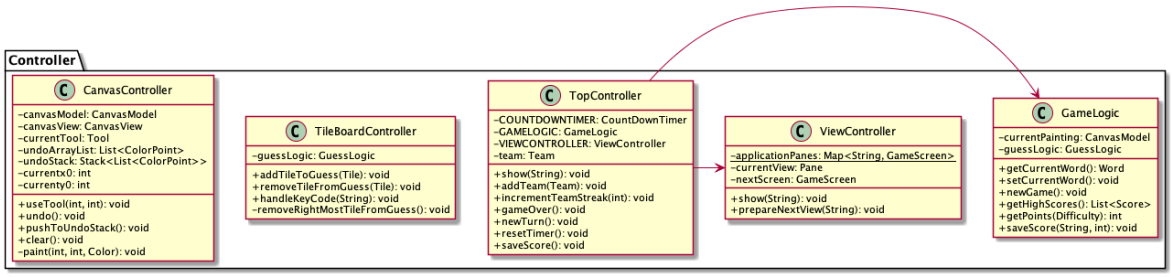

Figure 4: UML Controller Package

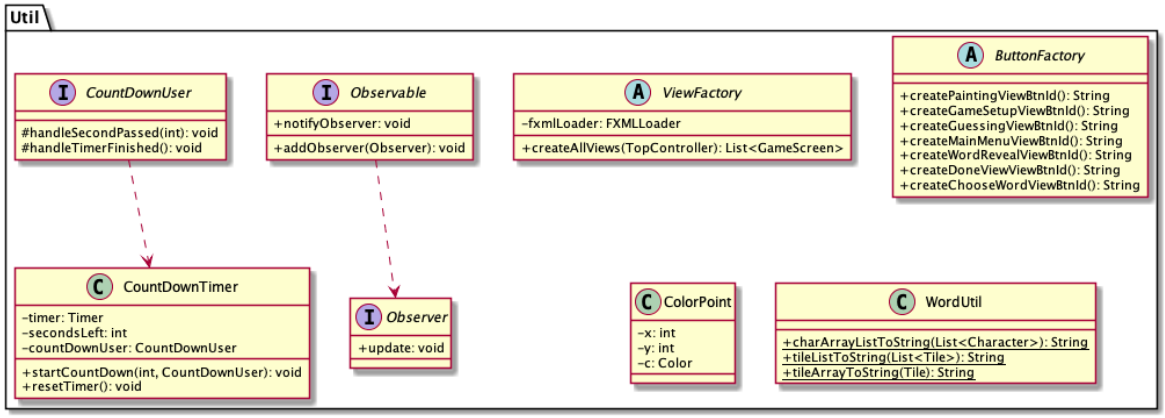

Figure 5: UML Util package

2.3. UI Design Patterns

Different design patterns from the book ”Designing interfaces” [1] 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.

2.4. 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.

2.4.1. 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 6.

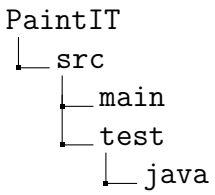

Figure 6: Figure showing where the tests are located

Figure 7 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.

2.4.2. 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 [2], 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 7: Showing a commented test of the canvasModel.

2.4.3. 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.

2.4.4. 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.

2.5. Quality tool reports

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

2.5.1. PMD

PMD is a source code analyzer that is used for finding flaws such as unused variables, empty catch blocks and the like [3]. 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.

2.5.2. FindBugs

FindBugs is a static code analyzer [4] 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 8 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't 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 8: Showing a snippet of the class MainMenuView, illustrating the possible unitilized variables.

2.5.3. STAN

STAN (Structure Analysis For Java) [5] 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.

3. 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 9.

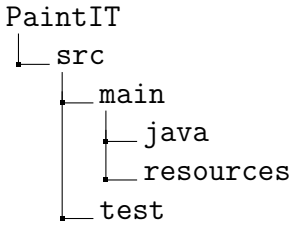

Figure 9: Figure showing where resources are located

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.

4. 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.

References

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

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

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

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

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

A. Appendix

A.1. STAN generated images

Generated STAN images.

Figure 10: View

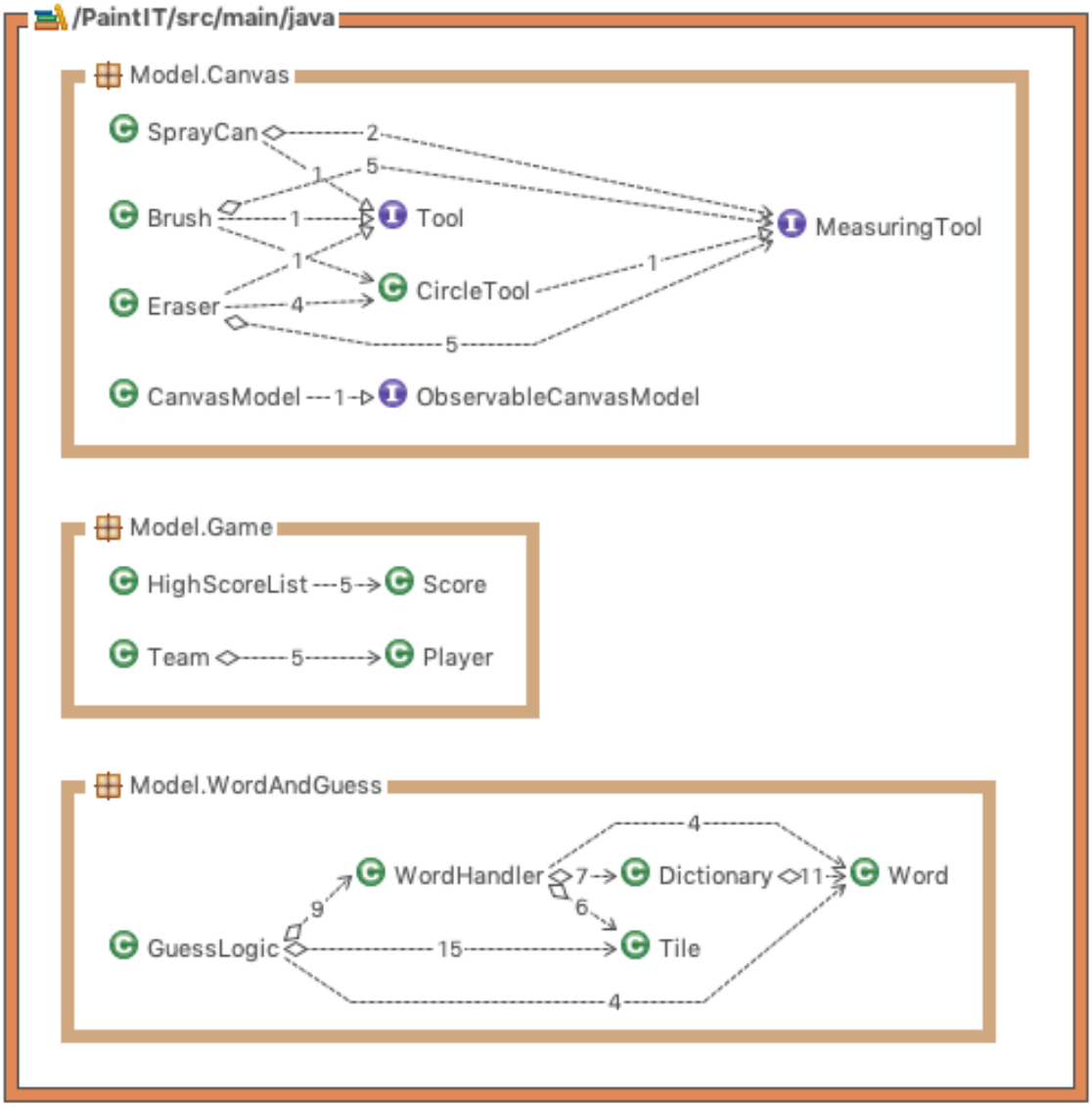

Figure 11: MODEL

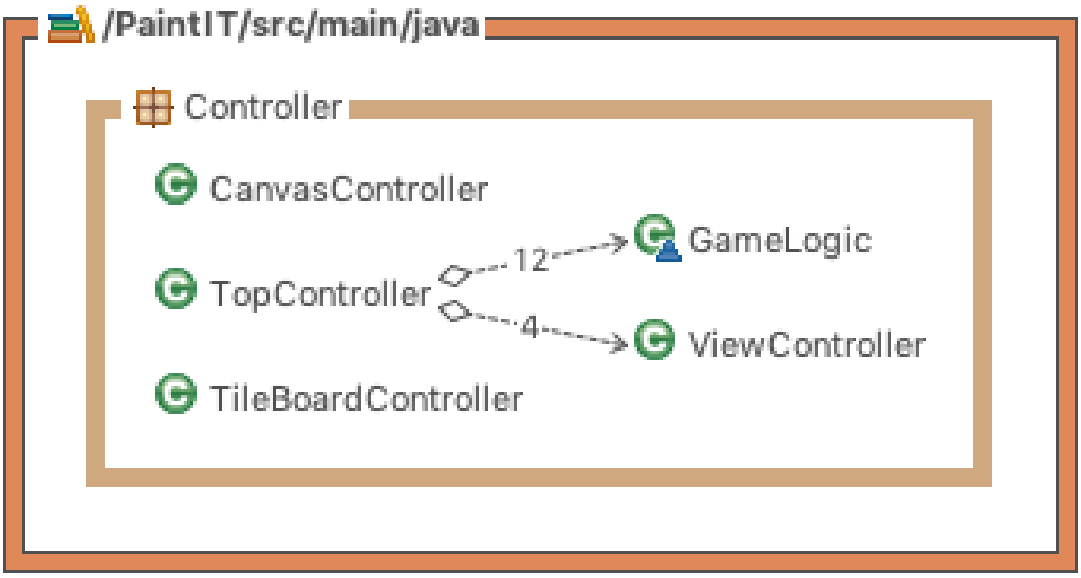

Figure 12: CONTROLLER

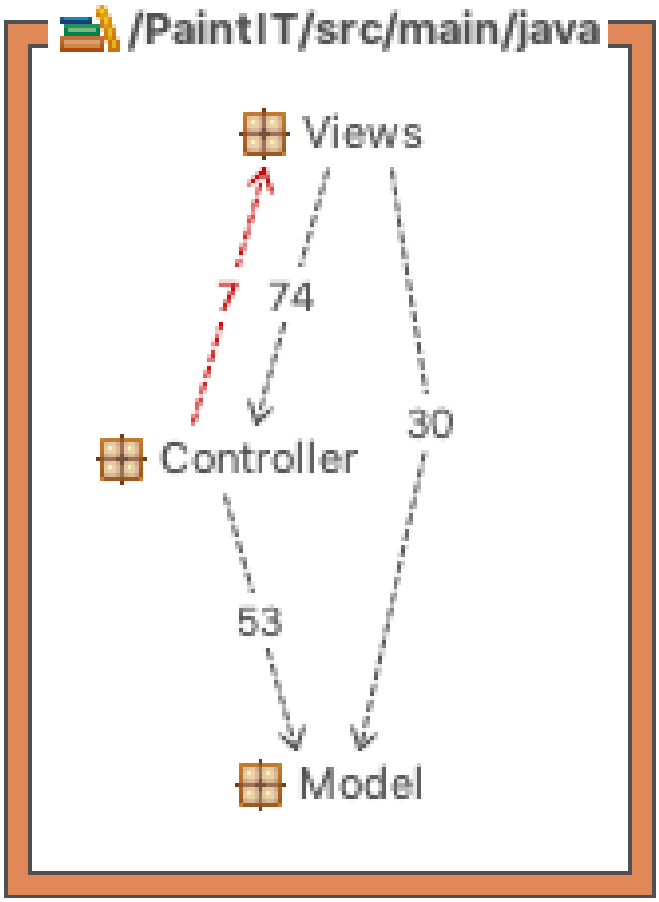

Figure 13: MVC