Final Report TDA367

Grupp C7
Elias Ersson, Hugo Ekstrand
Isak Gustafsson, Love Svalby

2021-10-23

Requirements and Analysis Document for C7Paint

Grupp C7
Hugo Ekstrand, Love Gustafsson
Elias Ersson, Isak Gustafsson

2021-10-23
2.0

Contents

1 Introduction

What is the application?

This project aims to develop an application for painting and editing images, much like GIMP or Photoshop but simpler. The application should contain basic tools for editing images and painting. This includes features such as:

- Painting on a canvas with a brush.
- An interface which is similar to most painting software, such as GIMP.
- Changing the brush to a different type, size, and color.
- Loading images into the applications canvas.
- Exporting what has been drawn or edited in the application to an image.

Who is the application for?

The application is aimed at the general populous for an easy and accessible image editing and painting application that is still more advanced than pre-installed applications such as Microsoft Paint.

1.1 Definitions, acronyms, and abbreviations

GUI Graphical user interface

MVC Model view controller

2 Requirements

2.1 User Stories

Done The user story has been completed

Not done The user story has not been completed

Priority has three orders:

1 highest priority.
Should be done.

2 medium priority.
May be done.

3 low priority.
Not expected to be done.

ID: 1Bru
Priority: 1
Status: Done
Name: Brushes
Description:
As a hobbyist illustrator I want to be able to use different types of brushes, such as calligraphy pens, so that I can draw more complex lines in my art.

Confirmation:
Functional:

- Can I select different brushes/pens in the tools box.
- When I have a brush/pen selected can I draw with it?
- Does the drawing of the brush reflect its type? Does a calligraphy draw a slanted line and does a circular brush draw a circle?

ID: 1Prp
Priority: 1
Status: Done
Name: Tool Properties
Description:
As a user I want to be able to configure my brushes and tools, such as changing the size of the brush head so that I can have more fine control over what the application does.

Confirmation:

Functional:

- Can I select a tool in the interface and get a list of settings I can change in a settings window below the toolbox.
- Can I change a setting in a tool by either entering a new value or dragging a slider.
- When I change the setting it should be reflected by the tools effect on the canvas when using it.
- Do Tool settings persist even if I switch tools, but I can clear my settings to reset the tool to its default values.

ID: 1Imi
Priority: 1
Status: Done
Name: Image import
Description:

As a casual user I want to be able to import images into the program so that I can work with them inside the program.

Confirmation:

Functional:

- Can I import all of the common image formats (.JPG, .PNG)?
- When I import an image does it appear on the screen?
- Can I draw and interact with the imported image?

ID: 1Lam
Priority: 1
Status: Done
Name: Layer manager
Description:

As an artist, I want the picture to have different layers so I can edit different parts of the image without changing other parts.

Confirmation:

Functional:

- Can I edit a ”layer” (a portion of an image) independently of other layers?
- When I draw on a layer over another layer does the data of the layer bellow persist when I erase or move the data above.

- Can I change which layer I am editing in the layer controller by pressing on the layer I want to select?

ID: 1Arl
Priority: 1
Status: Done
Name: Add and remove layers
Description:

As a user I want to be able to create and remove layers. I want to be able to create them so that I can create new drawing surfaces, so I mustn't affect something I have drawn on another layer. I also want to be able to remove layers so that I mustn't create a new project every time I create a layer I don't want to keep.

Confirmation:

Functional:

- Does there exist a create and remove button for layers by the layer controller?
- When I press the create button does the application prompt me for the size of the layer I want to create.
- When I create a layer does it get added to the layer controller and can I draw on it?
- When I remove a layer by pressing the remove button does it get removed from the layer controller. Also, does it get removed from the main canvas view?

ID: 1Exi
Priority: 1
Status: Done
Name: Export images
Description:

As a user I want to be able to export my drawings into a standard image format so that I can use my drawings outside the program and share it with others that do not have the program.

Confirmation:

Functional:

- Can I export my drawings as a .JPG, .PNG, and as a .TIF?
- Can I select where the file should be exported and select this destination?

- Does the exported image look the same as what I see in the program.

ID: 1Era
Priority: 1
Status: Done
Name: Eraser
Description:

As a drawer I want to be able to erase what I've drawn so that a I can remove what I've drawn in a particular area. This allows me to change the area or simply remove an area.

Confirmation:

Functional:

- Can I select an eraser in the toolbox?
- When I use the eraser on the canvas does it remove what I've drawn?
- Can I change the size of the eraser?

ID: 1Lai
Priority: 1
Status: Done
Name: Layer visibility
Description:

As a user I want to be able to hide layers in the layer controller so that I can easily either remove a portion of a layer I don't want to export. Alternatively I want to also be able to quickly see how a layer underneath another layer looks in its fullness without modifying the layer above. This allows be to not have to destroy my work just to view separate portions of my work.

Confirmation:

Functional:

- Does there exist a button to hide or show a layer in the layer controller?
- When I toggle the button does the selected layer change its visibility?
- If I export a project with a layer that is hidden, does the hidden layer not show up on the exported image?
- When I save a project with a hidden layer, does it still persist when I load the said project.

ID: 1Lat
Priority: 1
Status: Done
Name: Layer thumbnails
Description:

As a user I want to be able to quickly see what I've drawn on each layer so that I can quickly know which layer I want to remove, select, or modify.

Confirmation:

Functional:

- Can I see each layer separately as a thumbnail in the layer manager view?

ID: 2Cnp
Priority: 2
Status: Done
Name: Create new projects
Description:

As a user I would like to create a new project/clear my current project when the application is running. This allows me to clear or create a new project without having to close the application.

Confirmation:

Functional:

- Can I select a "new" item in the menu for creating a new project.
- Does a window prompt be for the size of the new created project when I press on "new".
- Is a new project created when I have selected a size? And does the old project get removed, including all of its layers?

ID: 2Exp
Priority: 2
Status: Done
Name: Export project
Description:

As a user I want to be able to save and later reload my work so that I can continue my work even after I've closed my computer.

Confirmation:

Functional:

- Can I save the project I am working on?
- Can I choose were it will be saved?
- Can I load a project, given a selected project file.
- Does the loaded project contain the same layers data as when the project was saved.

ID: 2Prb
Priority: 2
Status: Not done
Name: Project background
Description:

As a drawer I want to be able to easily see where my canvas ends so that I do not draw outside of the canvas. It also helps be to see how the exported image will look.

Confirmation:

Functional:

- Does the canvas in the application have a background which is not the same color as the canvas?
- Does this background work for any size of a canvas. That is, if I create a new project with a different size, does this background adher to the size.

ID: 2Cai
Priority: 2
Status: Not done
Name: Canvas info
Description:

As a user I want to be able to quickly see the size of my canvas, the location of my cursor on the canvas, and other general information about the canvas I am drawing on. This helps me for example align lines, since I can check that the x-coordinate matches on two points.

Confirmation:

Functional:

- Is there a bar at the bottom of the window containing information of where the cursor is and the size of the window?
- Does the cursor position change with the cursor?
- Is the cursor position relative to the drawable canvas?

ID: 2Ucp
Priority: 2
Status: Not done
Name: User created brush patterns
Description:

As an advanced user I want to be able to use customize patterns with my brushes, so that I can create my own unique brushes that does exactly what I want.

Confirmation:

Functional:

- I can create a new brush with an empty brush head pattern.
- I can select a brush head pattern out of a list of predefined ones.
- I can enter my own patterns with a bitmap file.
- The pattern I've added is selectable in the list of predefined ones.
- When I select a pattern on a newly created brush and draw with it, the selected pattern is drawn.

ID: 2Sbf
Priority: 2
Status: Not done
Name: Sharpening and bluring filters
Description:

As a user I want to be able to blur or sharpen a selected region of a drawing or image so that I have greater control over how my art looks.

Confirmation:

Functional:

- Can I select a region and either increase or decrease its sharpness?

ID: 2Und
Priority: 2
Status: Not done
Name: Undo
Description:

As a user I want to be able to undo the last couple of strokes I've painted or any other effect I've done to the canvas so that I can reverse a stroke I am not

satisfied with.

Confirmation:

Functional:

- Can I click a back button to undo a stroke with a brush?
- Can I click the button multiple times so that I can undo at least the 5 last strokes?
- Is any potentially overwritten data by the stroke which will be undone preserved when I press undo?

ID: 3Mob
Priority: 3
Status: Not done
Name: Momentum brushes
Description:

As an advanced user I want to be able to user brushes with momentum which affects the stroke's color and shape so that I can best emulate a real life stroke in the application.

Confirmation:

Functional:

- Does stroke opacity decreases when the stroke is curving.
- Does stroke opacity decreases with speed.
- Does stroke size decreases with speed.
- Does stroke size increases in curves.

ID: 3Tab
Priority: 3
Status: Not done
Name: Tabs
Description:

As a multitasker user I want to be able to work on multiple projects at the same time. I want to be able to switch between them by pressing on a tab with their name. I want all of my currently open projects to be visible in this tab view. All of this so that I can better work on multiple projects simultaneously without having to open multiple instances of the application since I don't have room for multiple applications on my small screen.

Confirmation:

Functional:

- Does there exist a tab view above the canvas?
- When I load a project does it appear in this tab view?
- WHen I press one of the tabs, which is not the one I am currently on, does the canvas change to that one.
- Does my selected tools, and their settings, persist even when I change project?
- Can I close an opened project by pressing a close button on its tab?

ID: 3Onc
Priority: 3
Status: Not done
Name: Online colaboration
Description:

As an artist with many colaborator in my drawings I want to be able to work with my colaborators remotely so that I mustn't be at the same place to work together.

Confirmation:

Functional:

- Can I start the program and connect to a remote endpoint or open for connections to my endpoint.
- Can I get the project of the remote endpoint I connect to.
- Can I draw or edit the project I have connected to and that the changes update on the other host user's canvas.

2.2 Definition of Done

Any given user story is considered done once the following criteria have been met:

- The code passes the user story's acceptance criteria
- The code passes every unit test
- All of the codes public methods are documented
- The code is briefly reviewed by the other project members
- The code has been integrated into the development branch
- Any bugs discovered via merging to the development branch has been mended

2.3 User interface

The graphical user interface (GUI) for the application consists of a primary view (see 2). In this view the user may use tools on the canvas. Tools can be found in the tool window, as can be seen in the top right corner, and consist of different brushes, a zooming tool, and much more. The GUI also features a window for configuring the selected tool from the tool window, which is located below the tool selection window. Under the selection window there is a layer window, which is where the user can select which layer they want to draw on. Additionally, the visibility of layers can be toggled.
In addition to these primary windows there also exists a bar for general info about the current canvas, such as canvas size, located at the bottom of the window.

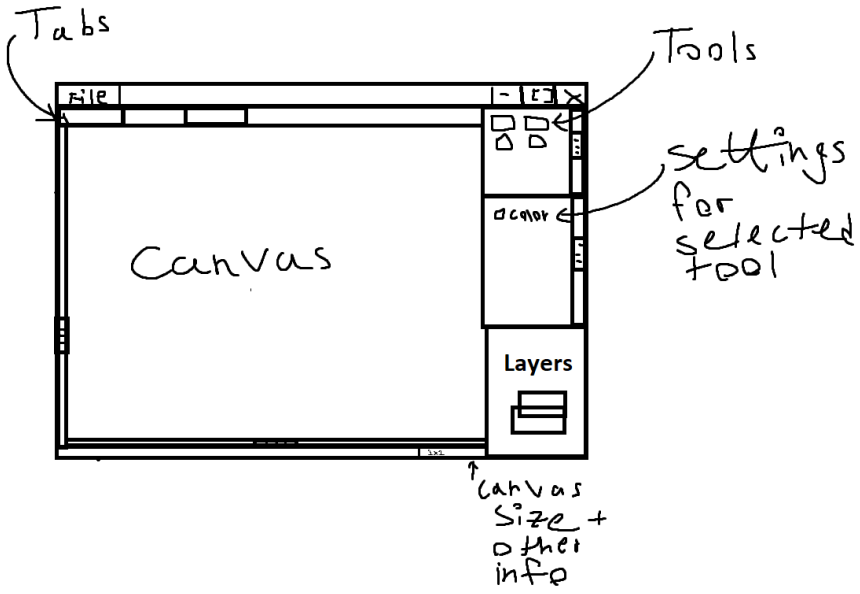

Figure 1: The first gui sketch

3 Domain model

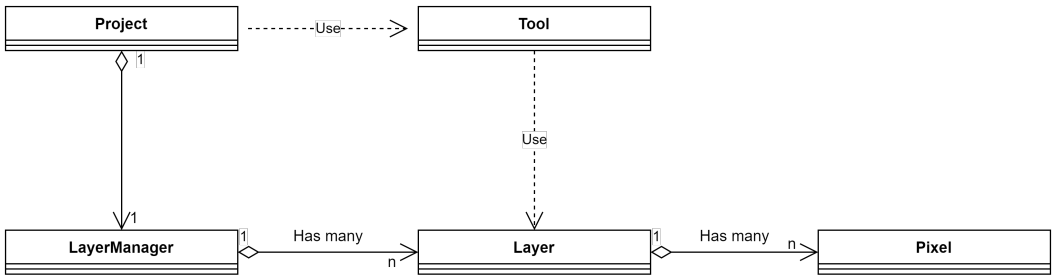

Figure 2: Domain model of C7Paint

3.1 Class responsibilities

Project

The project is the highest level of data for a piece of work in C7Paint. A project contain all the data for a project (such as a LayerManager) and the metadata (such as the project name). This forms a larger aggregate of image data and miscellaneous metadata which views and controllers interact with.

LayerManager

The LayerManager manages a collection of Layers. Its role is to allow the creation of new Layers and manipulation of existing Layers, such as reordering them. It is also responsible for combining Layers into a singular image which can then be saved or displayed on the view.

Layer

A Layer represents a drawable surface that can be rotated, translated, or scaled. It stores pixel data which can be manipulated by a Tool object. That is, a layer stores the raw pixel data of a particular portion of an image

Tool

The Tool class represents tools such as brushes, fill buckets, or anything else which would manipulate a layer. That is, a Tool is responsible for larger scale manipulation of layers, such as drawing shapes on a layer or transforming a layer. As can be deduced, the the Tool section is very broad.

Pixel

The pixel class represents an RGB color. It also handles color comparison, but for the most part it is a data class.

4 References

5 Tools

Project continous integration tool:
Travis, https://www.travis-ci.com/

Build automation tool:
Maven, https://maven.apache.org/

Integrated development environment (IDE):
Intellij IDEA, https://www.jetbrains.com/idea/

Version control:
Git, https://git-scm.com/

Remote repository hosting site:
Github, https://github.com/

UML class diagram tool:
Diagrams.net, https://app.diagrams.net/

UML sequence diagram tool:
Sequencediagram.org: https://sequencediagram.org/

Structureal analysis tool:
Stan, http://stan4j.com/

6 Frameworks and Libraries

Project unit test framework:
Junit, https://junit.org/junit5/

Project graphical framework:
Javafx, https://openjfx.io/

System Design Document for C7Paint

Grupp C7
Elias Ersson, Hugo Ekstrand
Isak Gustafsson, Love Svalby

2021-10-22
v.2.0

Contents

1 Introduction

This documents describes the design of the C7Paint application. It explains the general structure of the application via UML diagrams of the application's design model and domain model. The application, C7Paint, is a drawing and image editing software where users can import images, work on them with brushes and other drawing tools, and export created and edited images.

1.1 Definitions, acronyms, and abbreviations

GUI Graphical user interface

MVC Model-View-Controller

Layer A layer is a drawable surface. In many programs, includingt in this application, a layer can be moved around in a stack of many layers. This allows one layer, containing e.g. a drawing, can be ontop or below another layer containing a drawing.

Global space Global space represents a coordinate system in the same plane and location as the applications GUI. That is, 1 in the x-axis in the system represents one pixel on the screen.

Local space Local space represents a coordinate system local. Often this is used in relation to a layer. The local space of a layer would be the coordinate system which is transformed and translated to where a layer is. For example, if a layer were to be scaled so that it is twice as wide in the x-axis, 1 in the x-axis would represent two pixels in global space.

IO Input-output. Reading and writing to the computers secondary storage, such as its hard drive.

Mutate Change the state of a class.

Javafx This projects graphical framework. See Tools section for more info.

Stan This projects structural analysis tool. See Tools section for more info.

Junit This projects unit testing framework. See Tools section for more info.

2 System Architecture

The project is divided into three parts. The **Model**, which contains all logic and data not directly related to the ways a user can interact with the application; The **View**, which is responsible for displaying information about the model and available options to the user; The **Controller**, which interacts with both of the other parts, sending instructions to the model based on the user's interactions with the view.

2.1 Application flow

The application is launched through an entry point class which instantiates and connects the other components. When the view is initialized a window is created on the screen using the Javafx library.

When a user interacts with control elements such as buttons or sliders, or uses the cursor to draw on the canvas, events are triggered and handled by the controller which uses information about the interaction to send the appropriate instructions to the model. After updating accordingly, the model may indirectly notify the view through the observer pattern, and the view displays the changed model to the user. To see a concrete example refer to figure 1 which shows what happens if the user draws a line on the applications primary canvas.

To close the program, the user is expected to simply terminate the process, most likely through the window's close button. At this point, all data that has not been explicitly saved will be lost.

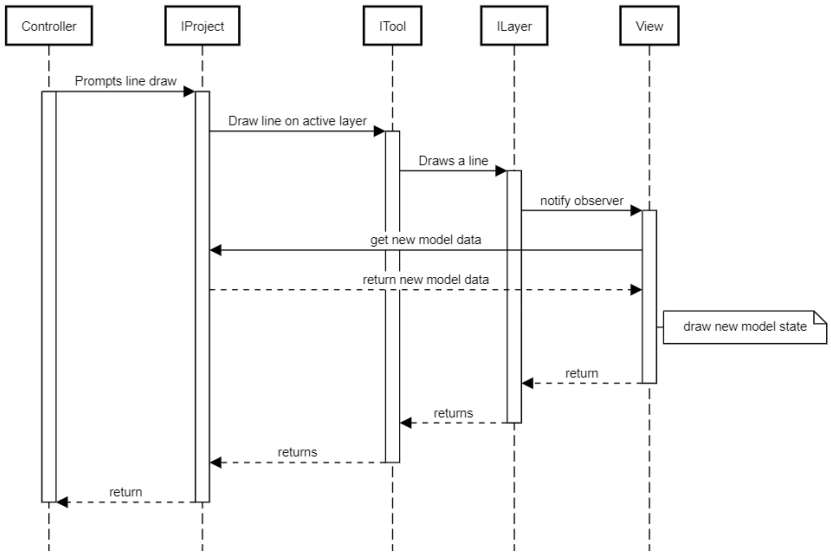

Figure 1: A UML sequence diagram showing the general flow of the application if the user propts the controller to draw a line on the applications primary canvas.

3 System Design

In this chapter the design of each individual package is discussed in greater detail than in the system architecture chapter.

3.1 MVC structure

The application follows the MVC structure. As can be seen in figure 2, the application has five main packages: Model, View, Controller, Services, and Util.

- The Model package contains the domain logic of the application.
- The View displays the model's current state through the applications graphical framework, Javafx.
- The Controller package translates input from the graphical framework to the model.
- The Services package contains operations done with model classes which do not fit inside the model, such as IO operations.
- Lastly, the Util class contains utility classes which have no dependency on other packages but is used by multiple packages.

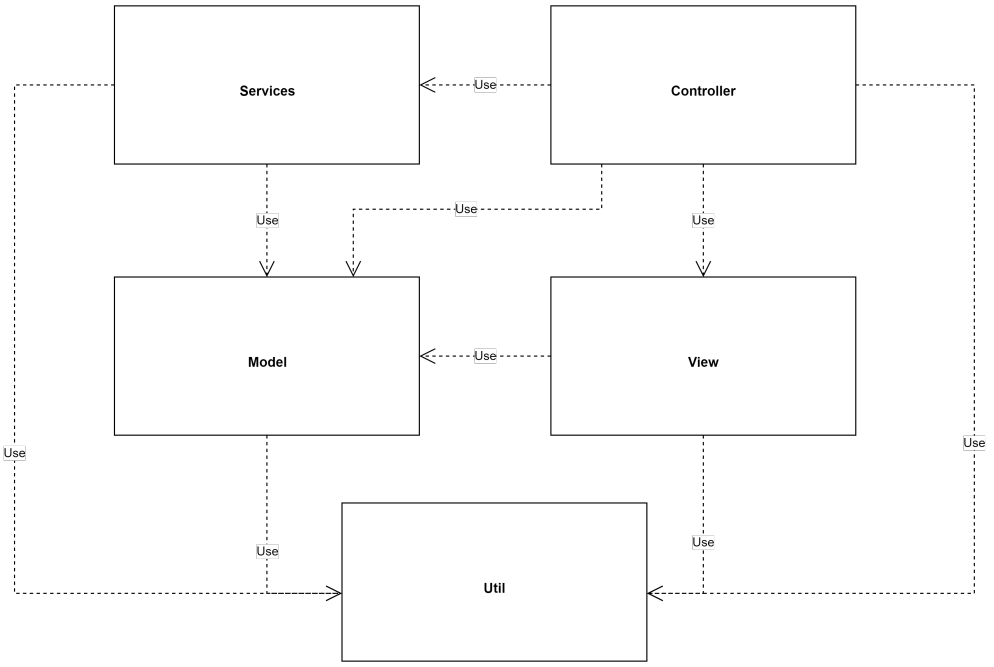

Figure 2: Overhead view of the main packages in this application with arrows indicating the direction of their dependencies on each other.

3.1.1 Inter-package Dependencies

As is illustrated in the package diagram in figure 2 the dependencies between the packages can be described as:

- Model is independent from every package. That is, it has no outwards dependencies except for to Util. This independence is, of course, an important part of the MVC structure.
- Controller interacts with View, Services, and Util. It is the common controller for connecting the program's discrete parts and thus has many dependencies.
- View depends on Model and Util. Notably, it has a read-only dependency on the model. That is, it does not mutate the model but only polls for the its state.
- Services depend on Model and Util.
- Util is truely independent and has no outwards dependencies. This is important since the model has a dependency on Util. To ensure that Model is truely independent of the graphical framework and View and Controller. Util must have no outwards dependencies.

The only outwards dependencies by Services, Controller, Model, and View are of interfaces. This enforces the principle of depending on abstraction rather than concrete classes. The Util package is relived from this rule since its classes have very varied responsibilities and are not expected to be changed.

3.2 Model

The model represents the domain logic of the application. It handles the actual internal drawing and image processing operations, such as combining figuring blending layers together.

3.2.1 Overhead View of the Model Packages

The design model (see figure 3) of the project consists of three key elements: the layer and tools package and the classes and interfaces linking them together. Similarly the domain model also contains these two key elements, the LayerManager and Layer class andthe Tool class, (see figure 4), however in the design model they are more complex and thus divided into their own respective packages.

Notably the Project class can also be found in the design model, as a single class. Alike the domain model the Project class functions as a sort of root aggregate to the Layer package. It functions as an intermediate step in between the client (the view and controller) and the underlying system in the Layer package.

There exists a few more elements in the design model too: the IObserver and IObservable interfaces. These are simply linking structures between the Layer and LayerManager, but also any client objects which want to listen to changes in the LayerManager. They follow the Observable Pattern, so that the ILayerManager must not have a direct dependency to any client view or controller objects which may want to be notified of changes in Layers.

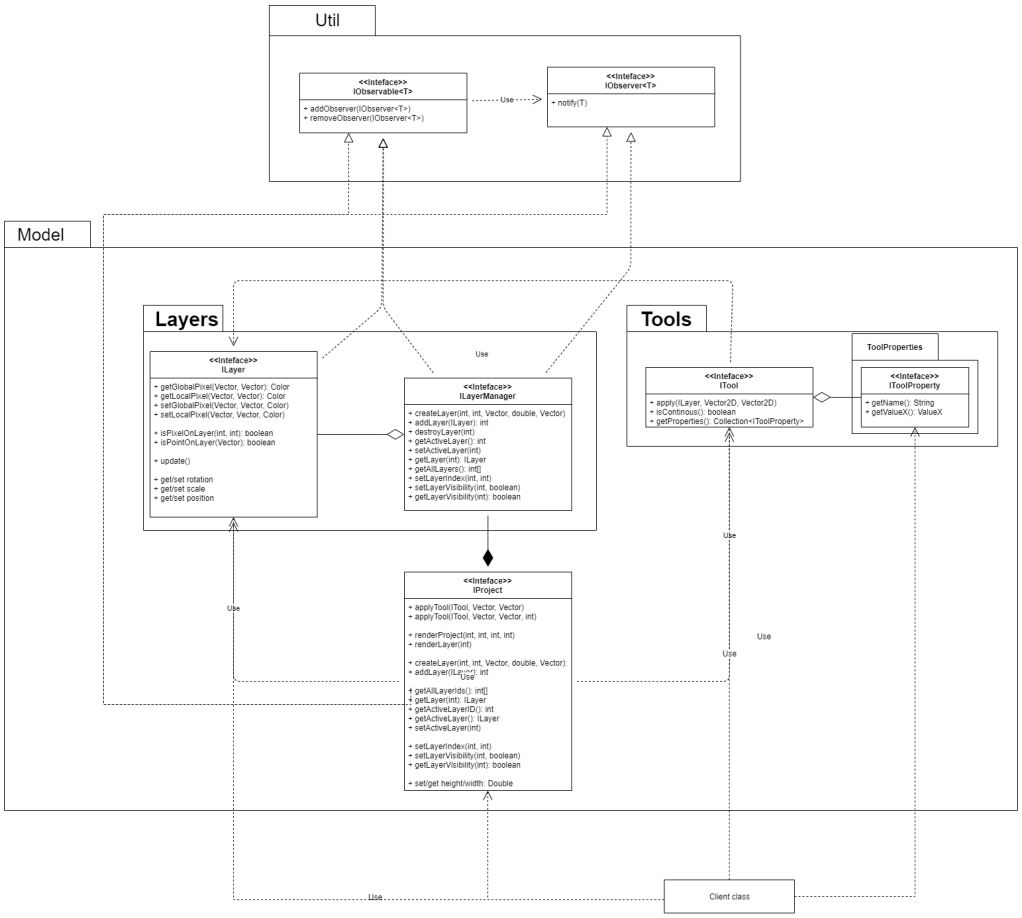

Figure 3: Overhead image of the model package containing the two sub-packages Tools and Layers. Also showing which classes are observers and observable.

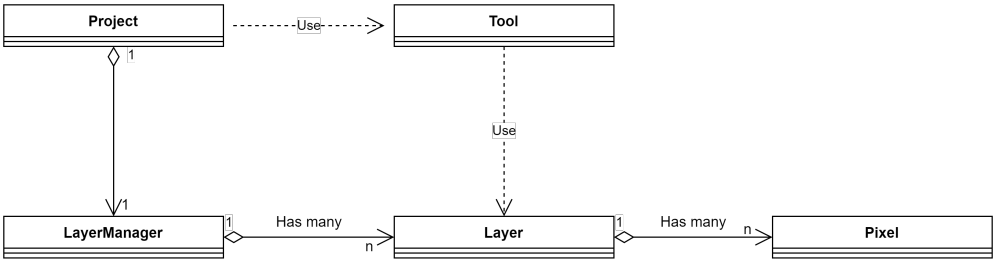

Figure 4: Image of the projects domain model

3.2.2 The Tools Package

The *Tool* package is a portion of the model package. It represents the *Tool* portion of the domain model (see figure 4). In the design model a tool performs some form of an action, given two points in global space and a layer to perform the action on. As implies by the name in both the design and domain model, a tool is very generic and can represent a lot of different concrete tools. However, one could divide the tools into three categories: tools which affect pixels in a stroke, tools perform some complex action on a point, and lastly transformation tools which transform layers.

3.2.2.1 Stroke Tools

Stroke tools are most structurally complex tool type. They contain two aggregates an *IPattern* and *IStrokeInterpolator* as can be seen in figure 7 in the *StrokeTool* class. The *IPattern* generates a collection of points which represent a 2d shape, such as a square or disk. It is this pattern which represents the shape of points which will be drawn. To create a stroke from these points and the two coordinate points given to an *ITool* extra points in-between the two given coordinate points. This is done with an *IStrokeInterpolator*. Essentially, a collection of points are create between the two given coordinate points. On each of these interpolated points the tools pattern is drawn on the layer given by the *ITool affect* method. This sequence of actions can also be seen in figure 5.

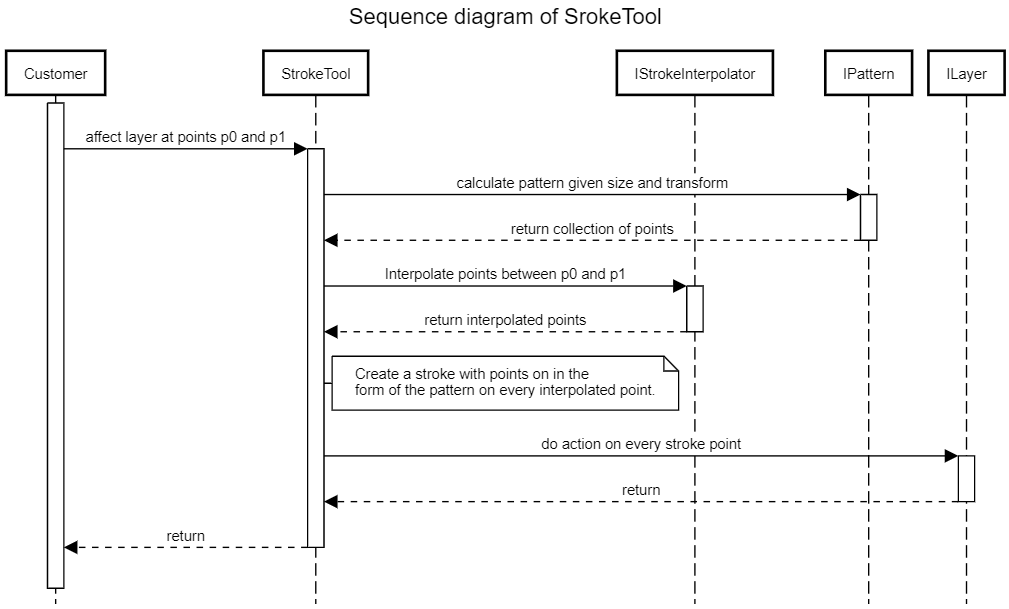

Figure 5: Sequence diagram displaying customer usage of a *StrokeTool* on a layer.

3.2.2.2 Point Tools and Transform Tools

Both tools which perform transformations and tools which perform some form of complex action on a point are often very specialized concrete implementations of *ITool*. Because of this there is very little abstraction that can be done. For example: the *FillBucket*, which would be considered a point tool, simply performs a specialized algorithm on a point to fill an area with a color. Similarly, the transform tools, such as *ScaleTool*, simply perform a mathematical transformation operation on the given layer.

3.2.2.3 IToolProperty

*ITool*s contain a collection of *IToolPropery*. These properties represent changeable data which can be changed in a tool. For example, a *FillBucket* has a threshold property for determining what is should fill. To allow access to these properties, which are often unique to every tool, while still having a common interface for all ITools, the *IToolProperty* system is used. There exists a few base data types which a property can be, e.g. an *IntegerToolProperty*. The IToolProperty interface contains a getter and setter for all of these base types and also a method for checking which type the property is. By doing this one can circumvent the need to type-cast properties, which would be needed if the properties were generic. This sequence of actiosn to change a property can be seen in figure **??**

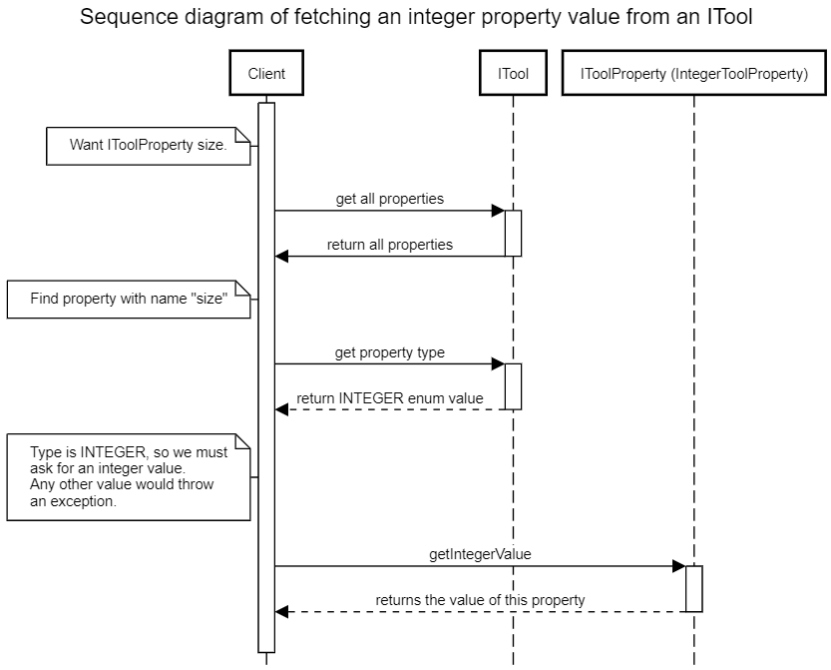

Figure 6: Sequence diagram show the actions required from the client to get the value of an *IntegerToolProperty* from an *ITool*

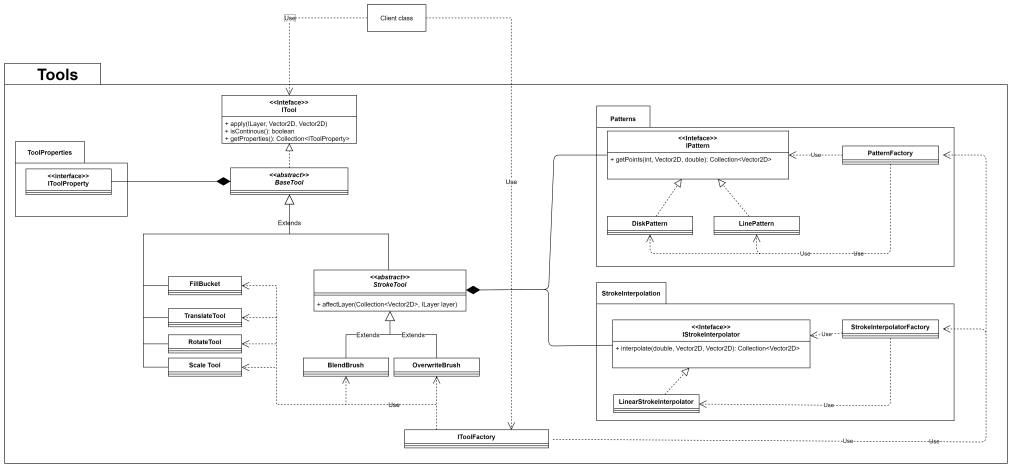

Figure 7: Detailed UML class diagram of the Tools package

3.2.3 The Layer Package

The *Layer* package is part of the *Model* package. It represents the part of the domain model that handles the creation and management of layers in an image project. That is, it represents the *Layer*, *LayerManager*, and *Pixel* portion of the domain model. The *Layer* package contains an interface for layers, an interface for a layer manager, and a factory class for creating layers with a specific implementation. This means that new

types of layers, such as layers that store colors in different data formats, can be added without changing the outwards interface of the package.

The *ILayer* interface is used for representing a single layer in a project that can be manipulated. It holds some sort of image data, which is the collection of *Pixel* in the domain model, that can be changed and retrieved.

The *LayerManager* is a helper class that acts as a collection of layers. It is responsible for any operation involving multiple layers, such as getting the aggregated color of all layers at a single point in space, keeping track of what order layers are to be drawn in, and storing whether or not a layer is currently visible.

3.3 View

The *View* package represents the view portion in the MVC structure. It handles the actual drawing of layers. It is sort of a middleman between the model and the graphical framework, javafx. That is, it translates the model so that it can be interpreted by javafx.

In order to increase the modularity of the view it reads model data from an interface, *IRender*, which only has the methods required by the view for requesting data to draw. The model classes which are supposed to be read are adapted to fit this interface, as can be seen in the Render package in figure 8. Notably, the *IRender* interface is also an observable, which notifies the view whenever a change to its data has been made. It also returns a rectangle encapsulating the area which has been changed. Because of this the view does not need to re-draw unnecessarily which improves performance drastically.
To give a concrete example: If we were to assume the view listens to a *LayerManager* adapted to an *IRender*. Then if we were to modify a *Layer* in the *LayerManager* then the LayerManager would be notified of the change and send it forth to the primary view (see figure 9). The view would receive an area to update and ask the adapted LayerManager for data in the received area and draw it.

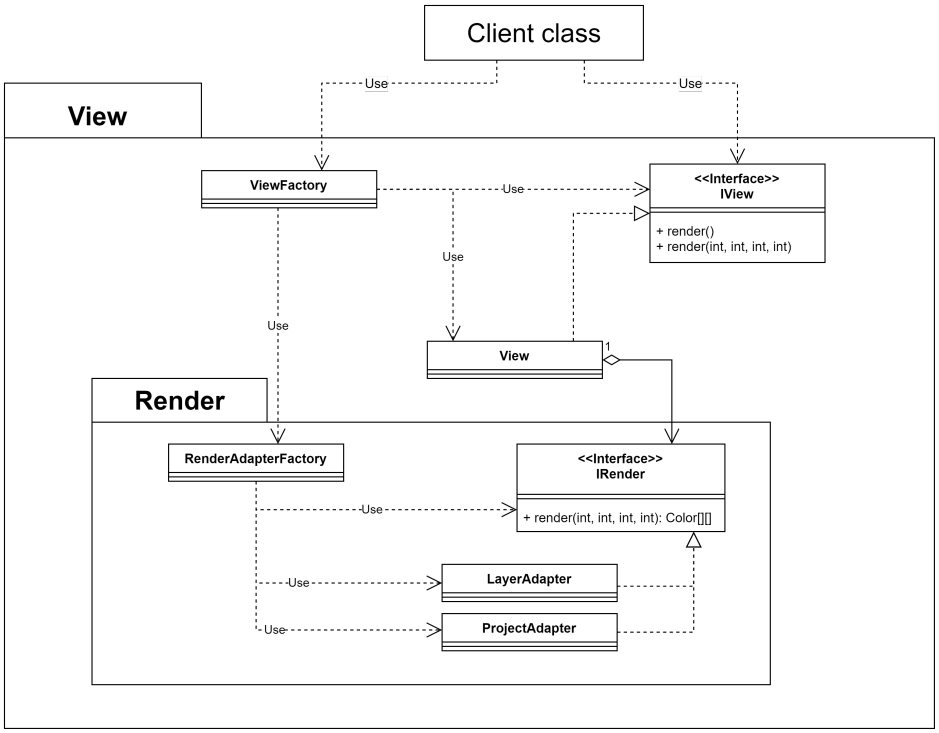

Figure 8: UML class diagram of the View package.

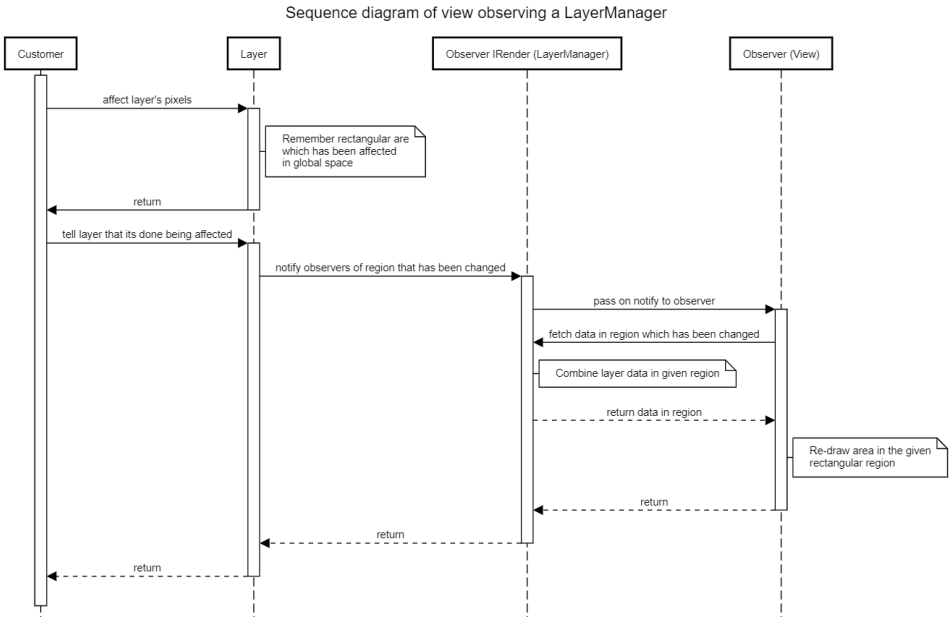

Figure 9: Sequence diagram showing a layer being drawn on and it leading to the view being notified of a change to model, resulting in the view re-drawing only the altered region on the layer.

3.4 Controller

The *Controller* package represents the controller portion of the MVC structure. Aside from factories for creating the other classes, all of its classes are implementations of Javafx's own internal controller classes, the instances of each of which connect to individual Javafx visual elements. The controller of a Javafx element can modify its features and handle events caused by users' interactions with it. The subpackage *Properties* contains classes made to translate user input to change the values of a *Tool*'s *IToolProperty*s so that tools can dynamically be changed by the user.

3.5 The Services Package

The *Service* package contains *IService*s which can perform actions with classes from the model. For example, there exists services for doing IO operations such as loading images from files to layers. IO operations should obviously not be in the model, and thus they are put in the *Service* package as an *IService*.

3.6 Util

Util is a tiny and shallow package containing classes without any outwards dependencies, but are depended on a lot. It lays the basic foundation for how the packages communicate with each others, such as the Util's observer pattern classes.

3.7 Design Patterns used

The application utilizes a few design patterns. The usage of these will and their benefits will be described in this section.

3.7.1 Factory Pattern

The factory pattern can be found in many places in the application. Most packages which want to hide their concrete classes, but still allow for external usage via a interface, use a factory. For example: it is expected that one can create a *ITool*, but one does not need to know that a *Brush ITool* has an internal interpolation object or that every type of *Brush* is the same except for having different *IPattern*s. As a result of disallowing the client class to acess any internal structures, or event depend on concrete classes, the code in the *Tool* package is more modular. In fact, theoratically the whole of the Tool package could be changed, except for the factory and its interface, and the client class would not need any change. Because of this the code is more open-closed; we mustn't do shotgun surgery just to change the *Tool*s package.
The same idea is applied to the *Layer* package. Different types of layers can be implemented, and the implementation of a specific type of layer can be changed, without the rest of the application needing to be changed.

3.7.2 Observer Pattern

The observer pattern is used heavily in the *Layer* and *View* packages. In order to allow the *Layer* package to notify the view of changes to itself without a dependency from the model to the view the observer pattern is used. Similarly, in the *Layer* package a *Layer* class instance notifies a *LayerManager* instance that it has changed via an observer notify call. This removes the dependency Layer would otherwise have to have to *LayerManager*, if it were to not use the observer pattern.

3.7.3 Bridge Pattern

The bridge pattern is mainly used in the *StrokeTool* class. In order to separate responsibility in the relatively complex *StrokeTool* class, major components have been divided into their own packages. A *StrokeTool* has a *IPattern* and an *IStrokeInterpolator*, both of which can be developed independently. This creates modularity too, and a good example of this is in the *ToolFactory*: the only diffrence between a brush with a brush with a circular draw shape and a calligraphy is their respective *IPattern* aggregates.

3.7.4 Adapter Pattern

The adapter pattern is used in the *View* package to adapt *ILayer*s and *IProject*s to the same *IRender* interface. Because of this adaptation both thumbnails and the primary canvas view can use the same view implementation. These adapters can be found in the *Render* package inside the *View* package, or in the UML class diagram in figure 8.

3.7.5 Template Pattern

The template pattern is used in the *StrokeTool* class. Because of its usage calculating where which pixels are affected in a stroke with a tool can be abstracted. This allows for the actual action performed on a layer by a tool to be put in separate classes, such as *BlendBrush* or *OverwriteBrush*, where a *BlendBrush* blends a color with a layer and a *OverwriteBrush* overwrites the color on a layer. This structure can be seen in figure 7.

4 Persistent Data Management

The application stores and loads a few sets of data. However, some of this data is dynamically stored and loaded during runtime and some are static and immutable during runtime.

- Dynamic
  - Saved projects
  - Exported images
- Static
  - Javafx fxml files

Broadly speaking, any persistent data is done locally on the machine. Similarly, static resources such as fxml files are stored locally.

4.1 Dynamic

All of the logic of getting the path for saving/loading is handled by the controller. This path is then used by a service for saving/loading.

4.1.1 Loading and Saving Projects

C7Paint makes use of java standard serialization for saving and loading Project to/from disc during runtime. C7Paint allows for the user to store and later retrieve projects in the form of .C7 files. These files are stored as regular files through the default file manager.

4.1.2 Loading and Saving Images

C7Paint allows for importing images into the currently open project. This can be done in two ways: dragging and dropping inside of project, and by a dedicated button under the menu bar File-¿Import. The actual saving to/from file formats are handled by the imported classes javax.imageIO and javafx.scene.image.

4.2 Static

The static files are all saved inside of the standard java resource folder. The primary static resources is javafx fxml files. These .fxml files are for defining what the windows contain and how they interact with our controller's code.

5 Quality

5.1 Tests

The project follows the standard Maven project structure, which infers that the project contains separate folders for source code for tests and the application source code (see figure 10).

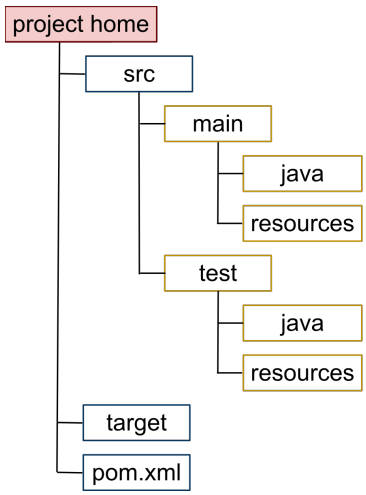

Figure 10: An image of the project structure of a Java Maven project. The directory labled ”src” contains the source code for the project. The main directory contains the applications code while the test directory contains the applications test code.

Tests are run with the Junit testing framework. Tests are created for each class, testing its public methods. The tests try to cover most of the functionality of the applications. In addition to Junit travis is used for continuous integration. The link to travis is here.

The test-coverage of both the Model and Service package are intended to have a high test-coverage of about 85% lines. This is due to especially the Model package being the most important part of the application; it handles all of the domain logic. Often domain logic is easier to test than the code for the view too.

5.2 Known Issues

- *Brushe*s draw twice at the end end start of stroke segments. This is especially notable when the *Brush*'s color has an alpha value less than 1. This is due to the Brush not having data about its previous stroke segment, causing it to ”redraw” overlapped stroke segments.
-

- Translating, rotating, and scaling larger layers takes a lot of performance.
- The StrokeTool doesn't perfectly scale with layers if they are heavily scaled. That is, the StrokeTool's stroke size is not perfectly consistent if the scale of the layer being affect is very large.
- There is a delay or hitch when drawing the first stroke on a layer.

5.3 Analysis Tools

5.3.1 STAN

STAN has been used for dependency analysis of this application. The projects general structure can be seen in figure 11 in appendix 1. More detailed images of the individual packages can also be found in appendix 1.
Notably one can see that there exists no circular dependencies between packages; that is, there are no arrows in the figure which goes, by proxy, in a circle or back and forth. Additionally, one can see that the *Model* package is completely independent of any other package except for the *Util* package. This is of course important if the code is supposed to follow the MVC structure.

5.4 Access control and security

N/A

6 References

References

[1] R. C. Martin, Objectmentor, "Design Principles and Design Patterns", [Online], Retrived 2021-10-23 from `https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf`

7 Tools

Project continous integration tool:
Travis, https://www.travis-ci.com/

Build automation tool:
Maven, https://maven.apache.org/

Integrated development environment (IDE):
Intellij IDEA, https://www.jetbrains.com/idea/

Version control:
Git, https://git-scm.com/

Remote repository hosting site:
Github, https://github.com/

UML class diagram tool:
Diagrams.net, https://app.diagrams.net/

UML sequence diagram tool:
Sequencediagram.org: https://sequencediagram.org/

Structureal analysis tool:
Stan, http://stan4j.com/

8 Frameworks and Libraries

Project unit test framework:
Junit, https://junit.org/junit5/

Project graphical framework:
Javafx, https://openjfx.io/

9 Appendix 1: Stan

This appendix contains images generated from the structural analysis tool Stan and some short explanations for why the structure is good or bad on some of the figures. It is divided into sections, representing the package that is being analyzed. It also contains the topmost strucute of the project.

One general comment to most of the figures is that there are not circular dependencies, which is considered good since circular dependencies can introduce a lot of issues [1, s.18].

9.1 Topmost Structure

The topmost structure of the view (see figure 11) contains a logical flow from the initalization package (*C7*) which nothing depends on to the *Util* package with a lot of dependencies. The most important parts of this structure is that the *Model* package only depends on the *Util* package which accommodates the MVC structure this project is supposed to have.

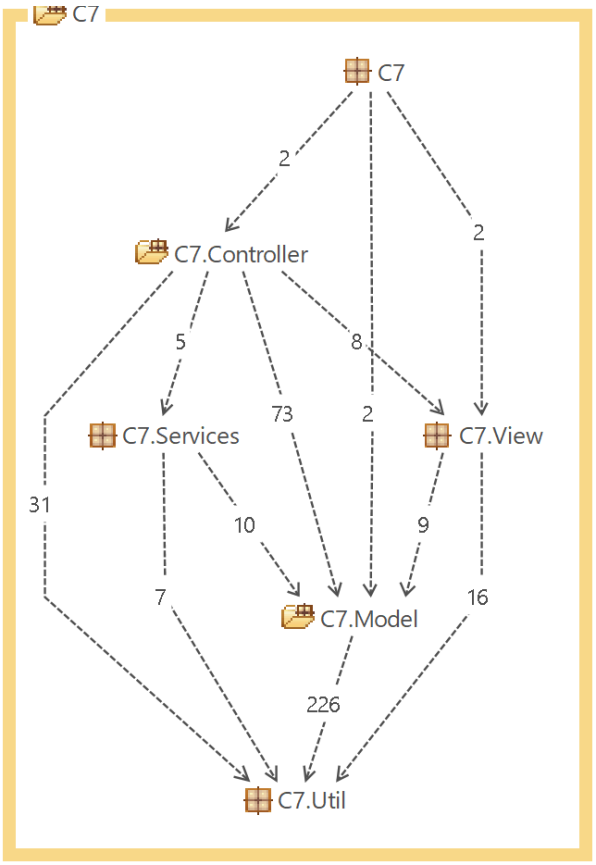

Figure 11: Overview of the project structure from the root folder

9.2 Model

The model has no circular dependencies between internal packages which is the for the most part the only important thing in figure 12.

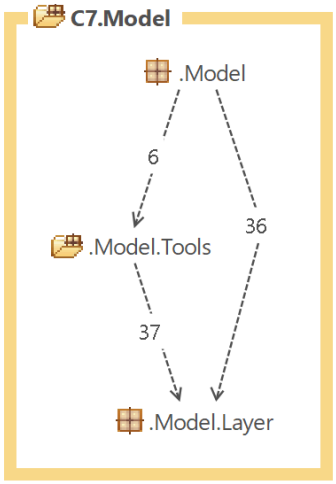

Figure 12: The topmost structure of the model

9.2.1 Tools

The *Tool* package has low coupling between internal packages (see figure 13), which infers that they are open-closed and that it is easy to extend one of them without having to modify all of them.

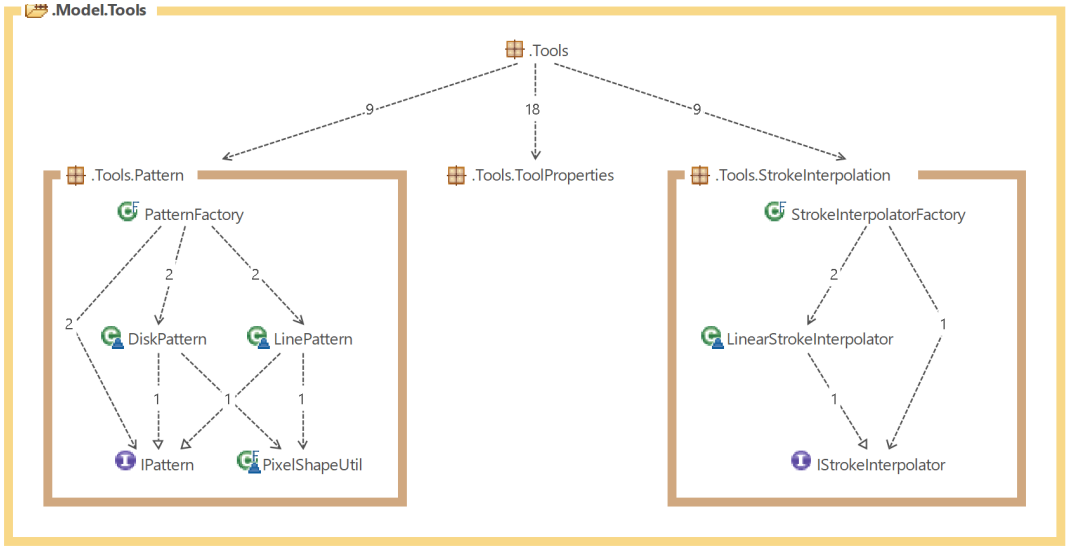

Figure 13: Partially expanded view of the Tools package's structure

Figure 14 show the general inheritance structure of an *ITool*.

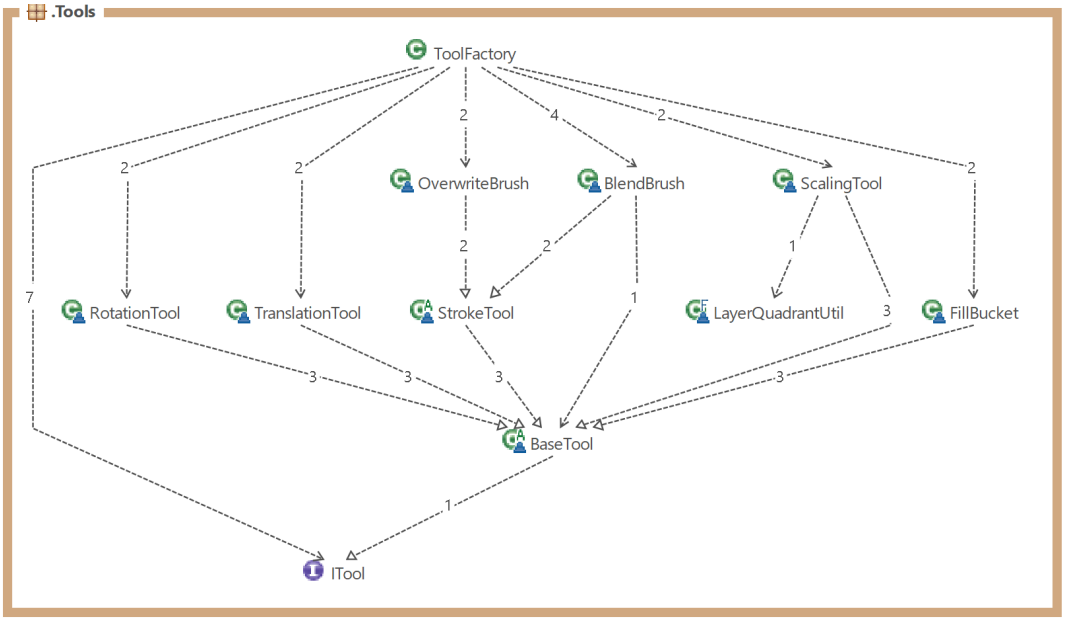

Figure 14: Expanded view of the Tools package

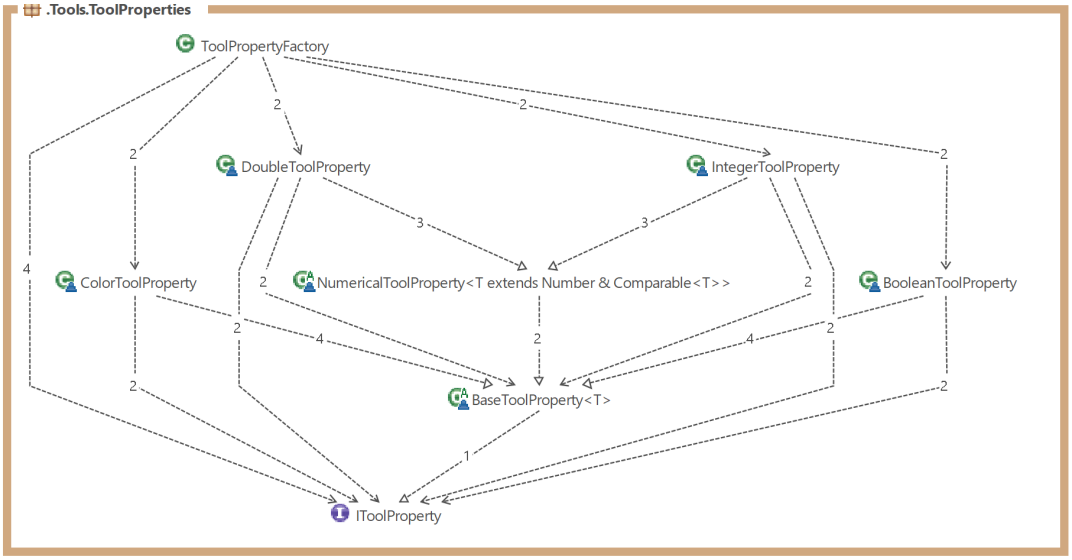

Figure 15: Expanded view of the Tools properties package

9.2.2 Layers

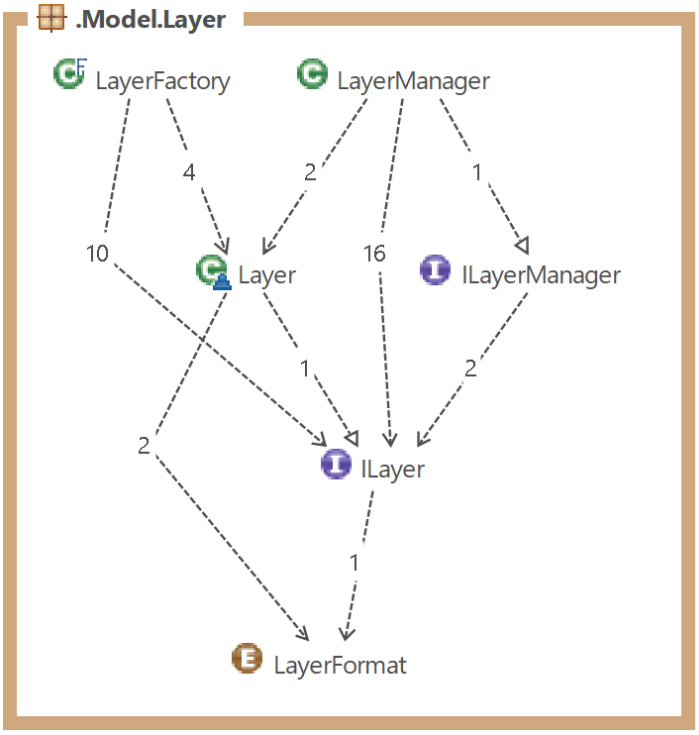

Figure 16: Structure of the Layer package

9.3 Services

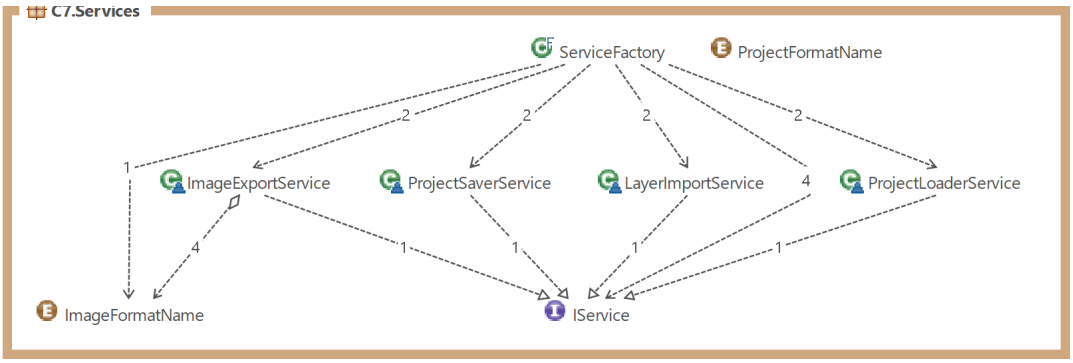

Figure 17: Structure of the Service package

9.4 View

The View package has an internal package, Render, which the primary package View has good separation from. The coupling is low as can be seen in figure 18 *View* only has 8 references to the package.

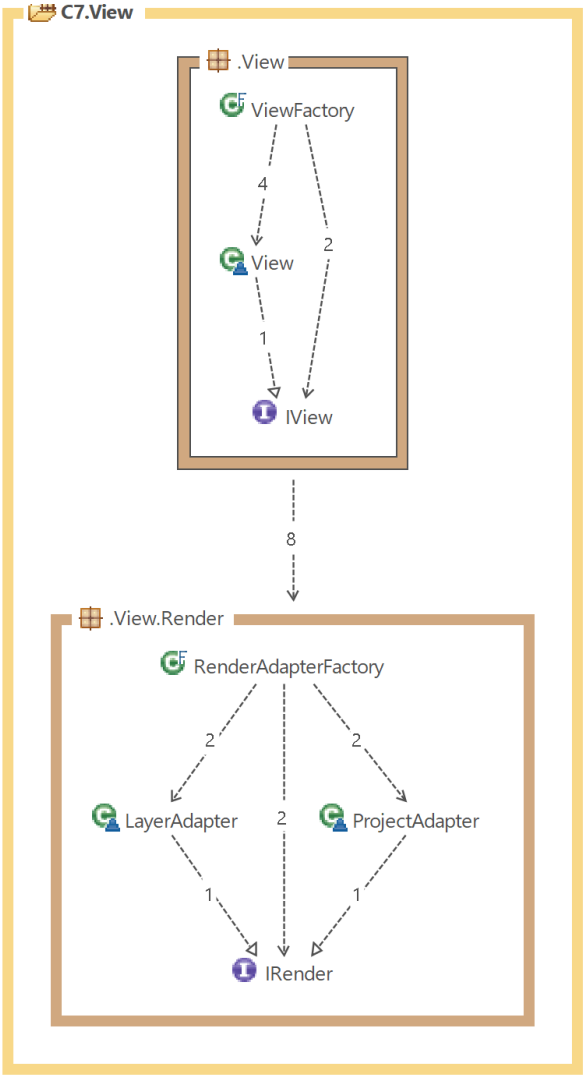

Figure 18: Structure of the View package

9.5 Controller

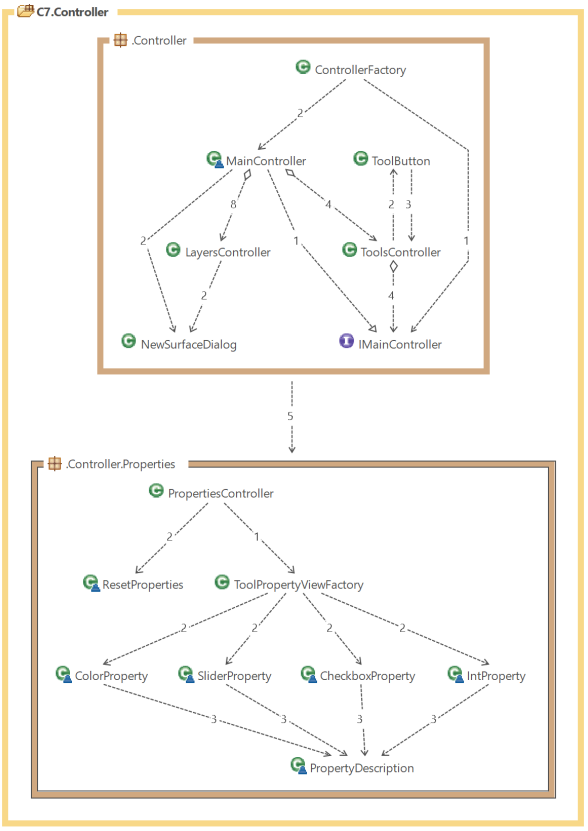

Figure 19: Structure of the Controller package

TDA367 Peer Review of Bestprogrammers_eu

By Group C7

Code Reusability, Open-closed, and Design Principles

Most of the code is single-use only and created on a case-by-case basis. However, model contains some elements of reuse. One example of this is how an Attack's effect is created by stringing *IEffects* together. This makes new Attacks easy to implement due to their effect being a reuse of *IEffects*.

The project attempts to use the factory pattern in multiple places but lacks one critical part of the pattern. The *Attack*- and *ItemFactory* returns a concrete class, *Attack* and *Item* respectively, instead of an interface. Both classes are thus not encapsulated since they have public access modifiers. A factory is supposed to hide the internal parts of a package, but if every class in the package is public the factory loses some of its meaning. Likewise, if the dependency the factory returns is concrete instead of an interface, the code is not as open to modification as it could be if the factory returned an interface.

The observer pattern is also used in this project in the *EffectAnimationHandler* inside the view package. This class notifies any listener that an effect should be displayed. However, the class is problematic in other ways which will be discussed in the MVC portion.

However, one relatively modular portion of the application is the controller. The controller functions as an aggregate with its root in the *InputController*. Every sub-controller implements the *IController* interface. Because of this use of an interface, one could easily add a new controller to the aggregate root.

Documentation, Readability, and Consistency

The code lacks almost any documentation or comments, except for a few TODO:s and a comment or exceptionally rare java doc here and there. Notably, the few comments that do exists are, however, inconsistent in its usage of language; some are in Swedish (e.g. in IEffect) and most are in English. The lack of comments affects the readability of the code since some classes, such as the *alterCurrentStats* method in the class *Puckemon* (see figure 3, appendix). The readability is even worse in classes containing magic numbers, such as the *EffectHelper's typeChart*. This 2d double array filled of magic constants and has no description for what its purpose is.

Class names are generally sufficiently intuitive to make their purpose comprehensible despite the lack of documentation, but there are a few examples of poor naming, such as the implementation of the graphics library's *Game* being named "*Boot"* and the player's opponent in the *Combat* class being simply named "*fighter*".

The consistency of the code in the project could be improved. Some classes have package-private modifiers for no apparent reasons while others have only private modifiers. Some methods use boxed Booleans while others use primitives, and again for no apparent reason. To see an example of this one could compare *HealAmount* and *ModifyAttackPower* which reside in the same package. Similarly, some interfaces have explicit public methods (e.g. *IFighter*) while other interfaces have implicit public methods (e.g. *ITrainer*).

MVC Structure

The most obvious violation of the MVC pattern is that the model package has dependencies on the view package. The *IEffects* in the *effects.effectTypes* package uses the singleton *EffectAnimationsHandler* for telling the view when to display an effect (see figure 1, appendix). Secondly, in the *ListItem* class in the model uses the imports images and stores them. This implies that the model is not independent from the view. All of this implies that the model has a hard

dependency on the graphics library and view. In addition to this, the *PartyScreen* view class modifies the model by switching *Puckemons* in the *switchPuckemon* method. Preferably the view should not mutate the model and just observe it.

In the view class *InvertoryView* model data (inventory items) is saved in instance variables. Optimally one would not save any model data in any view class.

The controller follows the MVC principles for the most part. It is separated form the view and model and has a clear responsibility of sending input and data to the model.

Dependencies and Abstractions

The code has circular dependencies. For example: the *Boot* class depends on the *InputController* and the *InputController* depends on the *Boot* class via the *setController* and *setView* methods. However, there are many more on the package level, and these can be seen in figure 1 and 2 in the appendix.

The concept of depending on abstraction rather than concrete implementation could be followed better. While the interface List is used in favor of ArrayList in most places and most instance variables and return types are interfaces. The project lacks interfaces for some relatively large components such as the class Attack or the Item class. Because of this concrete dependencies are used instead of dependencies on interfaces.

How is the performance?

There does not seem to be any performance issues with the application. However, the performance and stability of the application could be improved by only loading the excel file in *ExcelReader* once and then caching it instead of loading the excel file on every method call to the class. It is best to minimize IO operation during runtime.

Tests

The codebase has a low test-coverage in the model. Only the *ExcelReader* and *MonRegisterInterperator* have tests. Preferably all of the model would have tests so that bugs such as the "*GoldenNuggie*" *Item* crashing the application when used.

Improvements

Here is a list of improvements in no particular order which would increase the quality of the code.

- Use interfaces more so that concrete dependencies are minimized, especially for factories since that is a core part of the design pattern.
- Use an observer pattern or some other form of event bus the view can tap into instead of *EffectAnimationsHandler* for notifying the view of effects from the model. This removes the model's dependency on the view.
- Remove code which modifies the model from view. This makes the model independent.
- Use package-private modifiers for classes which should not be accessed outside a package. This modularizes packages more and forces customers to not depend on internal parts of the package.
- Document and use functional decompositioning more int code. This improves its readability, and thus its maintainability.
- Move IO operations, such as the excel reader, outside the model and into a service or utility package. The model should also preferably not store file paths.
- Make an excel database singleton class instead so that the excel sheet only needs to be read once and then cached. This decreases IO operations.

Appendix

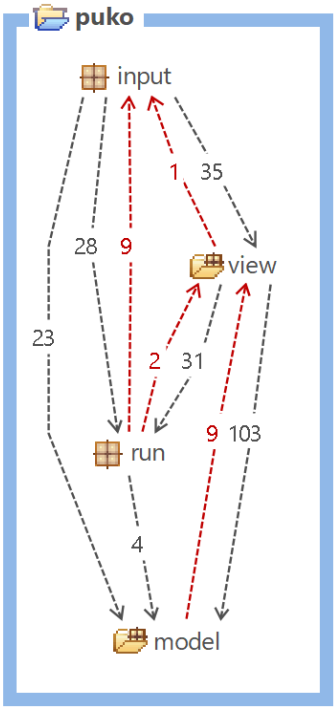

Figure 1: Root folder

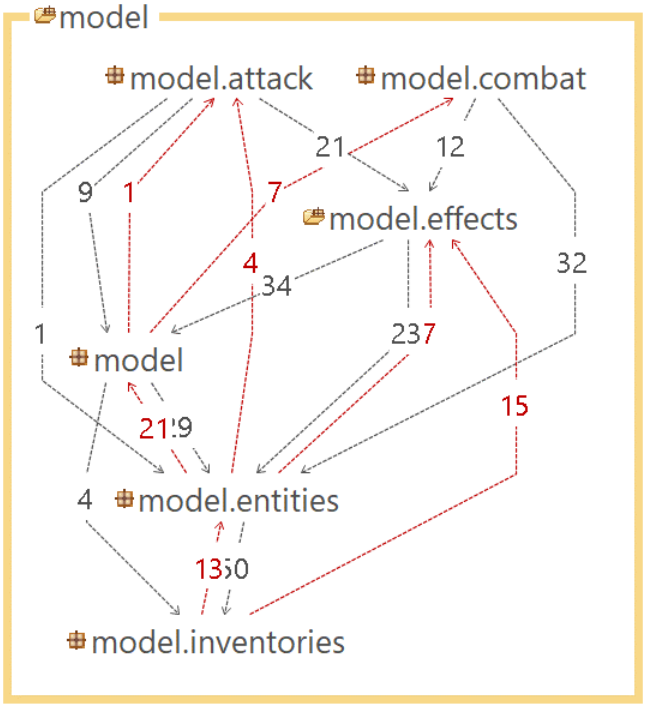

Figure 2: Model package

```
protected void alterCurrentStats(){
    if (attackPowerBuffFactor < 0){
        currentAttackPower = (int) (attackPower) * (2 / (2 + (-1) * attackPowerBuffFactor));
    }else{
        currentAttackPower = (int) (attackPower * (1 + attackPowerBuffFactor * 0.25));
    }
    if (defenceBuffFactor < 0){
        currentDefence = (int) (defence) * (2 / (2 + (-1) * defenceBuffFactor));
    }else{
        currentDefence = (int) (defence * (1 + defenceBuffFactor * 0.25));
    }
    if (speedBuffFactor < 0){
        currentSpeed = (int) (speed) * (2 / (2 + (-1) * speedBuffFactor));
    }else{
        currentSpeed = (int) (speed * (1 + speedBuffFactor * 0.25));
    }
}
```

Figure 3: "alterCurrentStats" method in model class Puckemon. Because of the lack of comments the code is harder to understand.

```
public class EffectHelper {
    private static PTypes[] types= {PTypes.NORMAL, PTypes.FIRE, PTypes.WATER, PTypes.ELECTRIC, PTypes.GRASS,

    private static double[][] typeChart = {
            //1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5, 0.0, 1.0, 1.0, 0.5, 1.0},
            {1, 0.5, 0.5, 1.0, 2.0, 2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 0.5, 1.0, 2.0, 1.0},
            {1, 2.0, 0.5, 1.0, 0.5, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 1.0, 1.0},
            {1, 1.0, 2.0, 0.5, 0.5, 1.0, 1.0, 1.0, 0.0, 2.0, 1.0, 1.0, 1.0, 1.0, 0.5, 1.0, 1.0, 1.0},
            {1, 0.5, 2.0, 1.0, 0.5, 1.0, 1.0, 0.5, 2.0, 0.5, 1.0, 0.5, 2.0, 1.0, 0.5, 1.0, 0.5, 1.0},
            {1, 0.5, 0.5, 1.0, 2.0, 0.5, 1.0, 1.0, 2.0, 2.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0},
            {2, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 0.5, 0.5, 0.5, 2.0, 0.0, 1.0, 2.0, 2.0, 0.5},
            {1, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 0.5, 0.5, 1.0, 1.0, 1.0, 0.5, 0.5, 1.0, 1.0, 0.0, 2.0},
            {1, 2.0, 1.0, 2.0, 0.5, 1.0, 1.0, 2.0, 1.0, 0.0, 1.0, 0.5, 2.0, 1.0, 1.0, 1.0, 2.0, 1.0},
            {1, 1.0, 1.0, 0.5, 2.0, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 1.0, 1.0, 0.5, 1.0},
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 1.0, 1.0, 0.5, 1.0, 1.0, 1.0, 1.0, 0.0, 0.5, 1.0},
            {1, 0.5, 1.0, 1.0, 2.0, 1.0, 0.5, 0.5, 1.0, 0.5, 2.0, 1.0, 1.0, 0.5, 1.0, 2.0, 0.5, 0.5},
            {1, 2.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 0.5, 2.0, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 0.5, 1.0},
            {0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 1.0},
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 0.0},
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 0.5},
            {1, 0.5, 0.5, 0.5, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 0.5, 2.0},
            {1, 0.5, 1.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 0.5, 1.0},

    };
```

Figure 4: double array filled with magic constants. The lack of comments, except for row index, makes the purpose of this array an enigma wrapped in a riddle.