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.

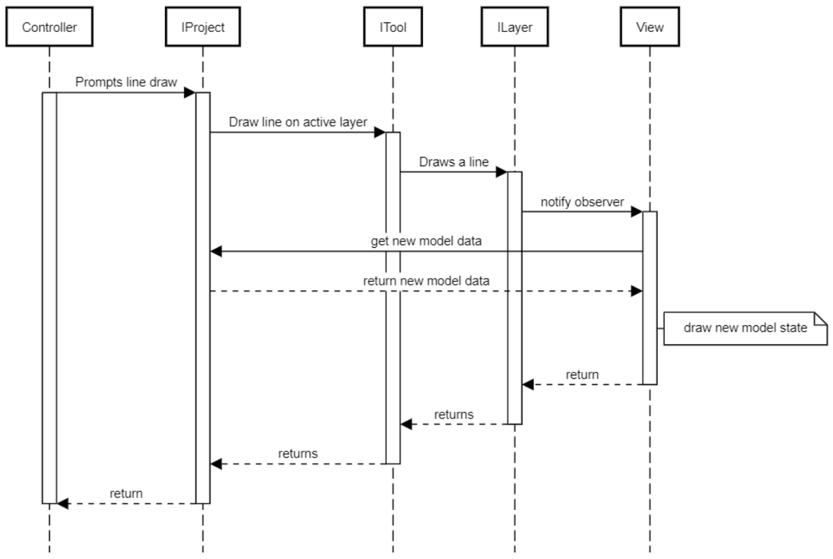


Figure 1: A UML sequence diagram showing the general flow of the application if the user propts the controller to draw a line on the applications primary canvas.

3 System Design

In this chapter the MVC structure is discussed in greater detail. The similarities between the domain and design model is also discussed. Every major portion of the design model is also discussed in greater detail, including design patterns used.

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.

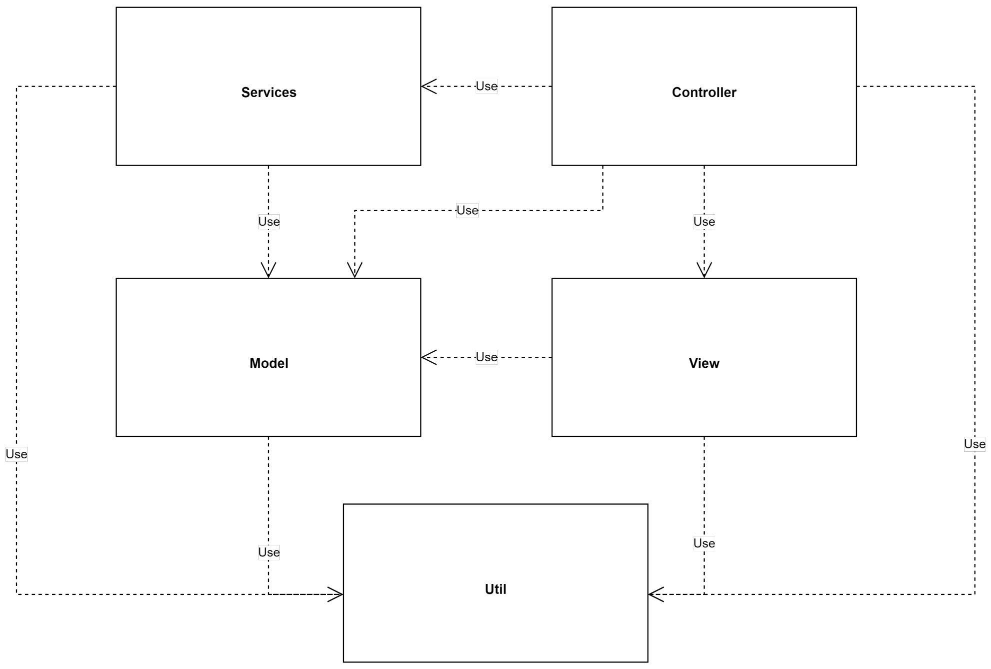


Figure 2: Overhead view of the main packages in this application with arrows indicating the direction of their dependencies on each other.

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 The Model Package

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

3.2.1 Overhead View of the Model Packages

The design model (see figure 4) of C7Paint consists of three key elements: the layer package, the tools package, and the classes and interfaces linking them together. Many of these elements have equivalences in the domain model (see figure 3). such as the *Layer* package which contains both an *ILayer* and an *ILayerManager*. Pixel is the only exception; this object in the domain model represented by a single class or interface. Namely, it is held as array instance variables concrete Layers.

The *IProject* interface can also be found in the design model. Alike the domain model the *IProject* interface functions as a sort of root aggregate to the *Layer* objects.
There exists a few more elements in the design model too: the *IObserver* and *IObservable* interfaces. These are simply linking structures between the *ILayer*, *ILayerManager*, and *IProject*, but also any client objects which want to listen to changes in any

of them. They follow the Observer Pattern, so that the *IProject* must not have a direct dependency to any client view or controller objects which may want to be notified of changes in its internal *ILayers*.

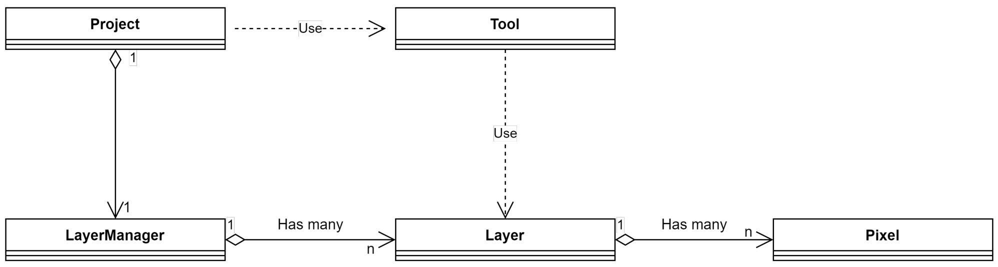
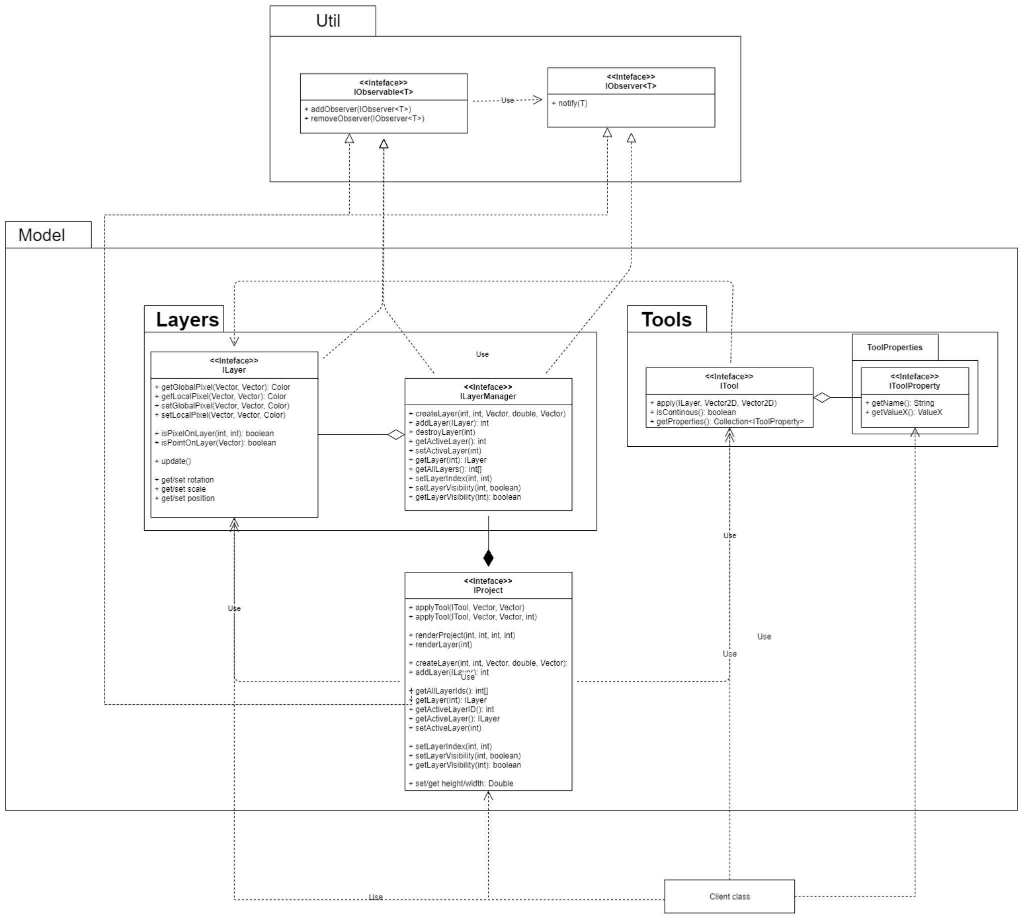


Figure 3: Image of the projects domain model



Util
<<Inteface>>
IObservable<T>
+ addObserver(IObserver<T>)
+ removeObserver(IObserver<T>)
Use
<<Inteface>>
IObserver<T>
+ notify(T)
Model
Layers
<<Inteface>>
ILayer
+ getGlobalPixel(Vector, Vector): Color
+ getLocalPixel(Vector, Vector): Color
+ setGlobalPixel(Vector, Vector, Color)
+ setLocalPixel(Vector, Vector, Color)
+ isPixelOnLayer(int, int): boolean
+ isPointOnLayer(Vector): boolean
+ update()
+ get/set rotation
+ get/set scale
+ get/set position
Use
<<Inteface>>
ILayerManager
+ createLayer(int, int, Vector, double, Vector)
+ addLayer(ILayer): int
+ destroyLayer(int)
+ getActiveLayer(): int
+ setActiveLayer(int)
+ getLayer(int): ILayer
+ getAllLayers(): int[]
+ setLayerIndex(int, int)
+ setLayerVisibility(int, boolean)
+ getLayerVisibility(int): boolean
Tools
ToolProperties
<<Inteface>>
ITool
+ apply(ILayer, Vector2D, Vector2D)
+ isContinous(): boolean
+ getProperties(): Collection<IToolProperty>
<<Inteface>>
IToolProperty
+ getName(): String
+ getValueX(): ValueX
Use
<<Inteface>>
IProject
+ applyTool(ITool, Vector, Vector)
+ applyTool(ITool, Vector, Vector, int)
+ renderProject(int, int, int, int)
+ renderLayer(int)
+ createLayer(int, int, Vector, double, Vector):
+ addLayer(ILayer): int
+ getAllLayerIds(): int[]
+ getLayer(int): ILayer
+ getActiveLayerID(): int
+ getActiveLayer(): ILayer
+ setActiveLayer(int)
+ setLayerIndex(int, int)
+ setLayerVisibility(int, boolean)
+ getLayerVisibility(int): boolean
+ set/get height/width: Double
Use
Use
Use
Use
Use
Client class

Figure 4: Overhead image of the model package containing the two sub-packages Tools and Layers. Also showing which classes are observers and observable.

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 3). 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 which perform some complex action on a point, and lastly transformation tools which apply affine transforms 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. Because of these two aggregates being interfaces and in their own packages, the bridge pattern is utilized. That is, both of these aggregates can be developed independently and thus increase the open-closeness of the *StrokeTool*. Both of these aggregates are used to calculate a collection of pixels in a stroke. What is actually done on these pixels are, however, defined in the inheritors of the abstract *StrokeTool* class. That is, the two concrete classes *BlendBrush* and *OverwriteBrush* decide what is done on the stroke. This allows more functionality from the same code: an eraser which overwrites data with nothing can use the same code base as a brush which blends together colors on a layer and the brush's color.

A concrete example of how a *StrokeTool* uses these aggregates can be seen in sequence diagram in figure 5.

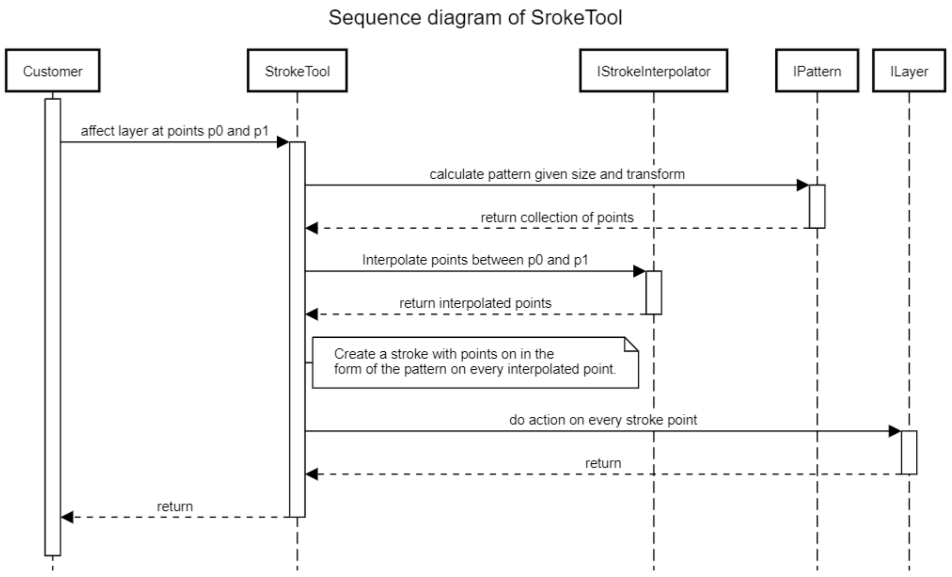

Sequence diagram of SrokeTool
Customer
StrokeTool
IStrokeInterpolator
IPattern
ILayer
affect layer at points p0 and p1
calculate pattern given size and transform
return collection of points
Interpolate points between p0 and p1
return interpolated points
Create a stroke with points on in the form of the pattern on every interpolated point.
do action on every stroke point
return
return

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 affine transformations and tools which perform some other 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 actions to change a property can be seen in figure **??**

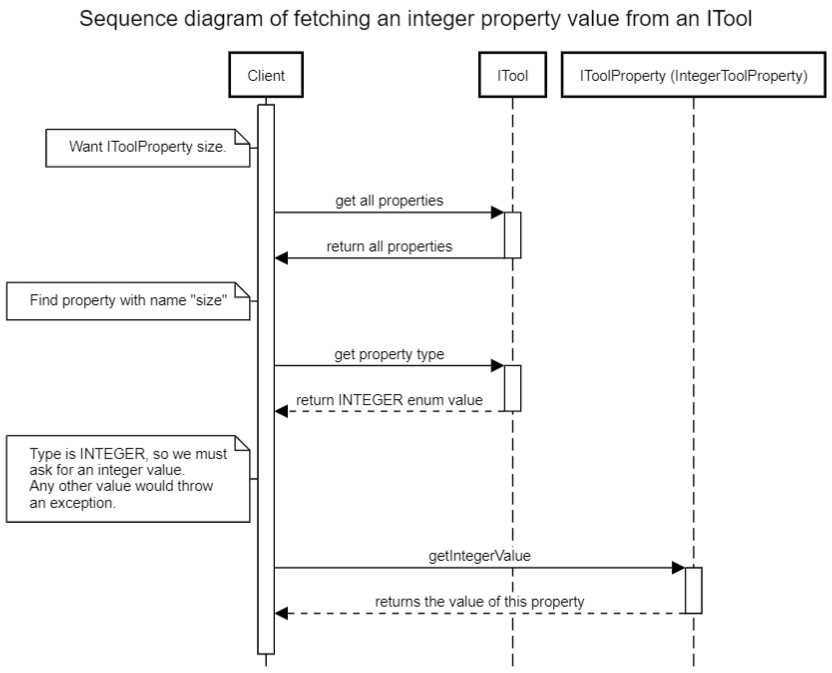

Sequence diagram of fetching an integer property value from an ITool
Client
ITool
IToolProperty (IntegerToolProperty)
Want IToolProperty size.
get all properties
return all properties
Find property with name "size"
get property type
return INTEGER enum value
Type is INTEGER, so we must ask for an integer value. Any other value would throw an exception.
getIntegerValue
returns the value of this property

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

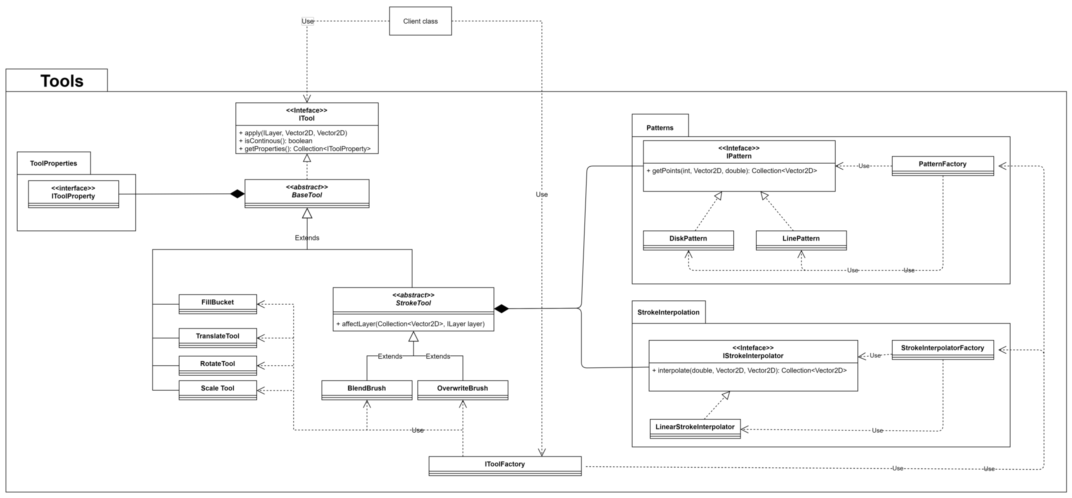


Figure 7: Detailed UML class diagram of the Tools package

3.2.3 The Layer Package

The *Layer* package is part of the *Model* package. It represents the *Layer*, *LayerManager*, and *Pixel* portion of the domain model. That is, the storing of image data in layers, manipulation of this image data, the creation of new layers and management of these layers. 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 The View Package

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. The *IRender* interface is also an observable, which notifies the view whenever a change to its data has been made. In this notification the view receives 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.

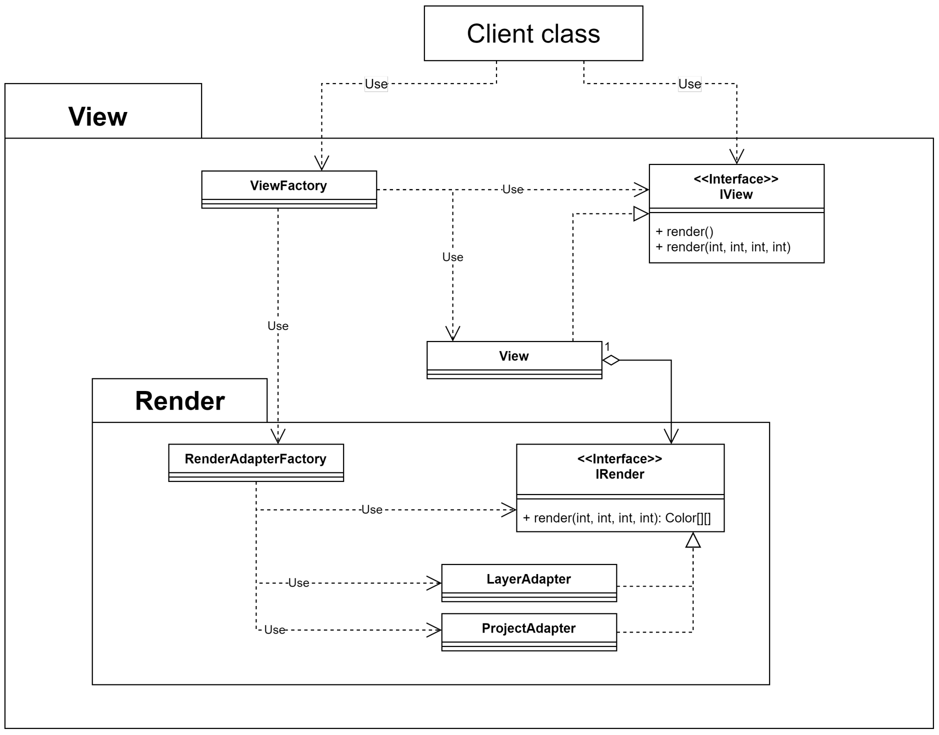


Figure 8: UML class diagram of the View package.

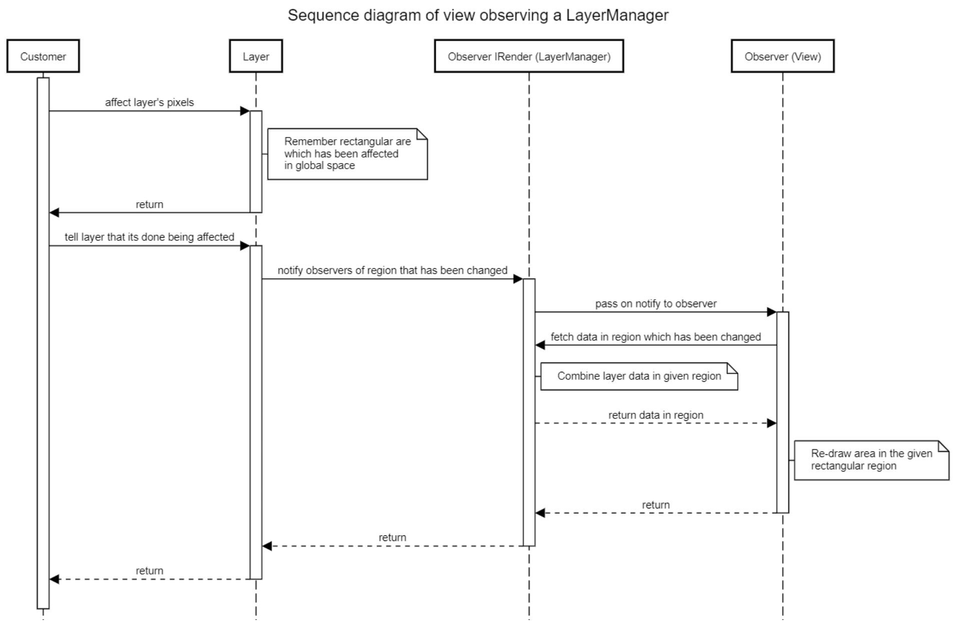

Sequence diagram of view observing a LayerManager
Customer
Layer
Observer IRender (LayerManager)
Observer (View)
affect layer's pixels
Remember rectangular are which has been affected in global space
return
tell layer that its done being affected
notify observers of region that has been changed
pass on notify to observer
fetch data in region which has been changed
Combine layer data in given region
return data in region
Re-draw area in the given rectangular region
return
return
return

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 The Controller Package

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 The Util Package

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 other, such as the Util's observer pattern classes. As well as providing datatype-like classes to streamline communication between packages with classes like Color.

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 an interface, use a factory. For example: it is expected that one can create a *ITool*, but one does not need to know that a *StrokeTool* has an internal interpolation object.

As a result of disallowing the client class to access any internal structures, or event depend on concrete classes, the code in the *Tool* package is more modular. In fact, theoretically 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.

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 on *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 *StrokeTool* class its 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, resulting in code-reuse. 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 all pixels in a stroke can be abstracted. This allows for the actual action performed on a layer by a tool on the calculated stroke to be put in separate classes, such as *BlendBrush* or *OverwriteBrush*. This creates code-reuse and seperation of concern. 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).

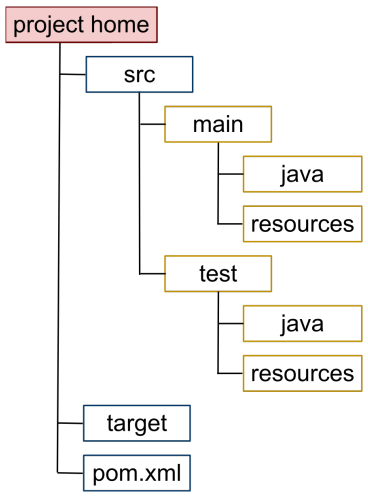


Figure 10: An image of the project structure of a Java Maven project. The directory labled "src" contains the source code for the project. The main directory contains the applications code while the test directory contains the applications test code.

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 (https://app.travis-ci.com/github/Spektria/C$7_TDA367/builds/$23891783).

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.
- The calligraphy brush has gaps in its stroke sometimes. This is especially visible if the layer being drawn on is scaled or rotated.
- Sometimes the view leaves ghost pixels of where a layer used to be if the layer is translated, rotated, or scaled. This seems to be occur when one uses the Fill-Bucket.
- Sometimes when starting launching the application the thumbnails don't load. However, when creating a new project via the file menu it always works.
- Scaling with the ScaleTool does not follow the mouse when the layer has been rotated.

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.

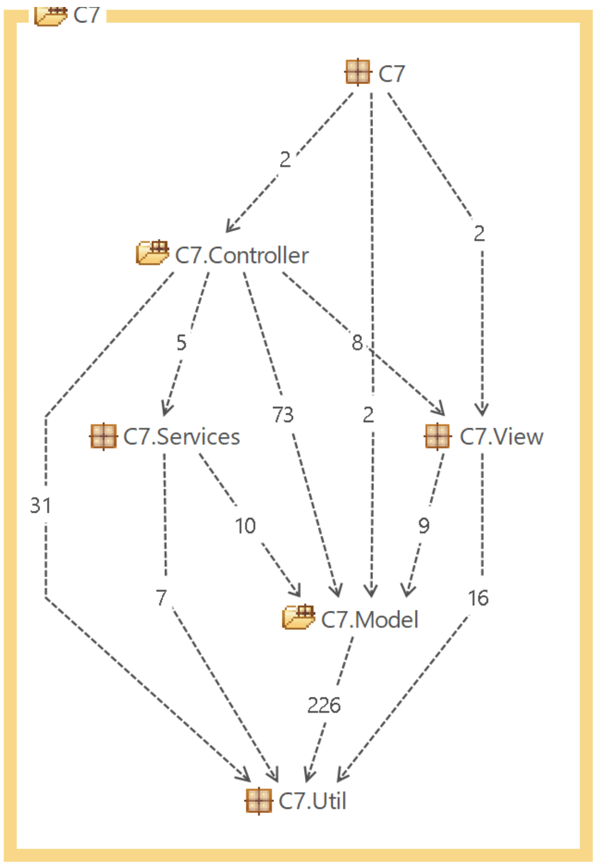


Figure 11: Overview of the project structure from the root folder

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.

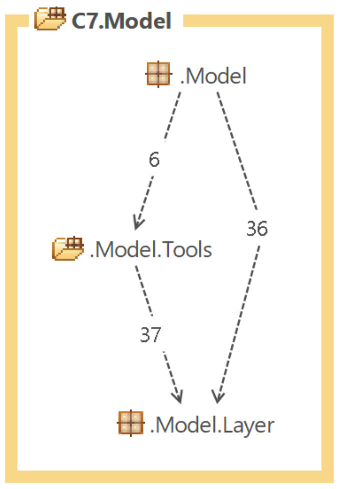


Figure 12: The topmost structure of the model

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.

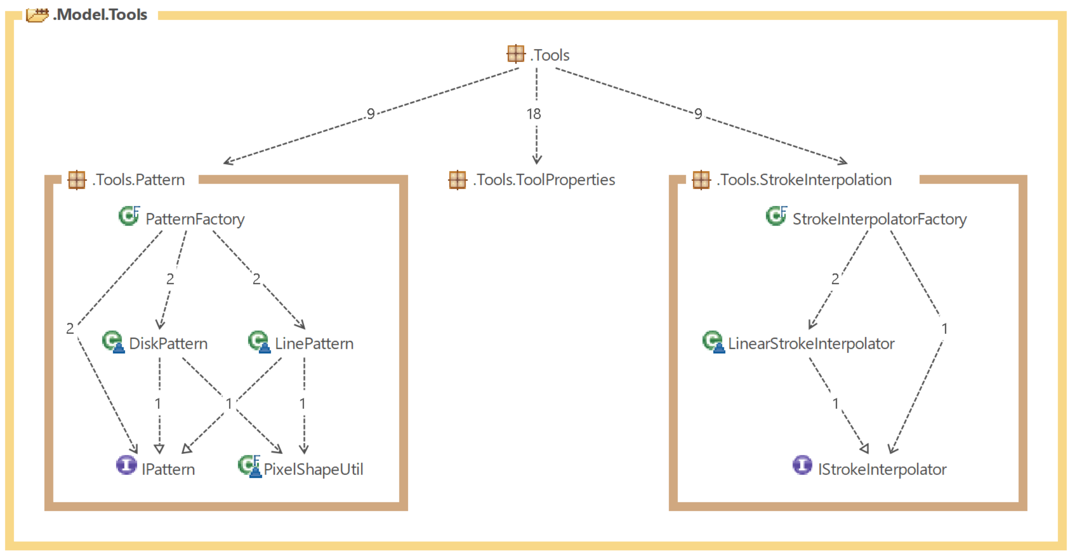


Figure 13: Partially expanded view of the Tools package's structure

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

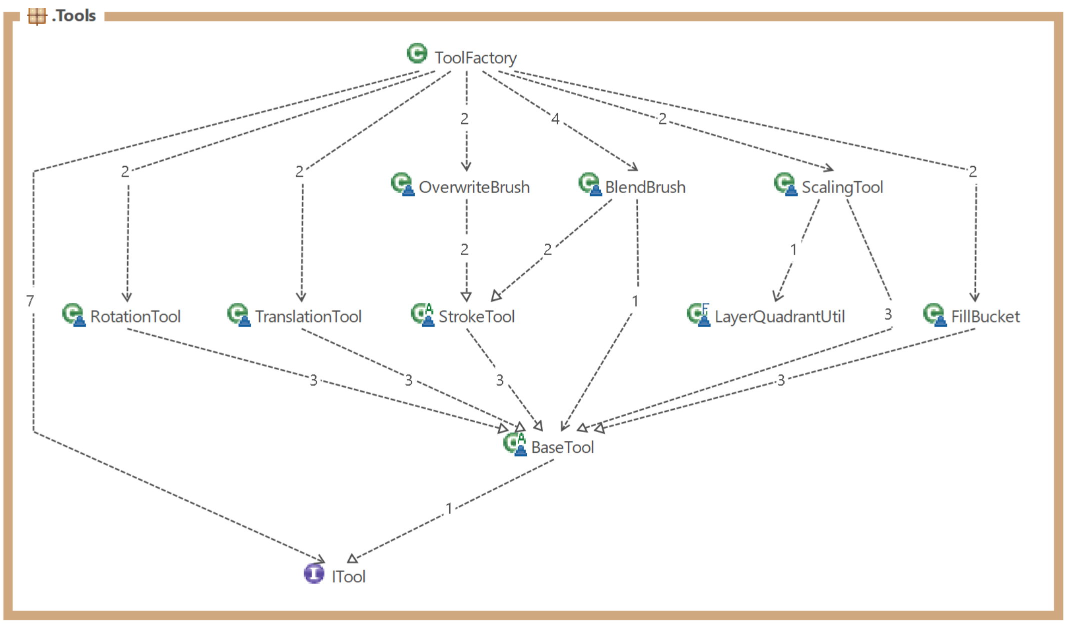


Figure 14: Expanded view of the Tools package

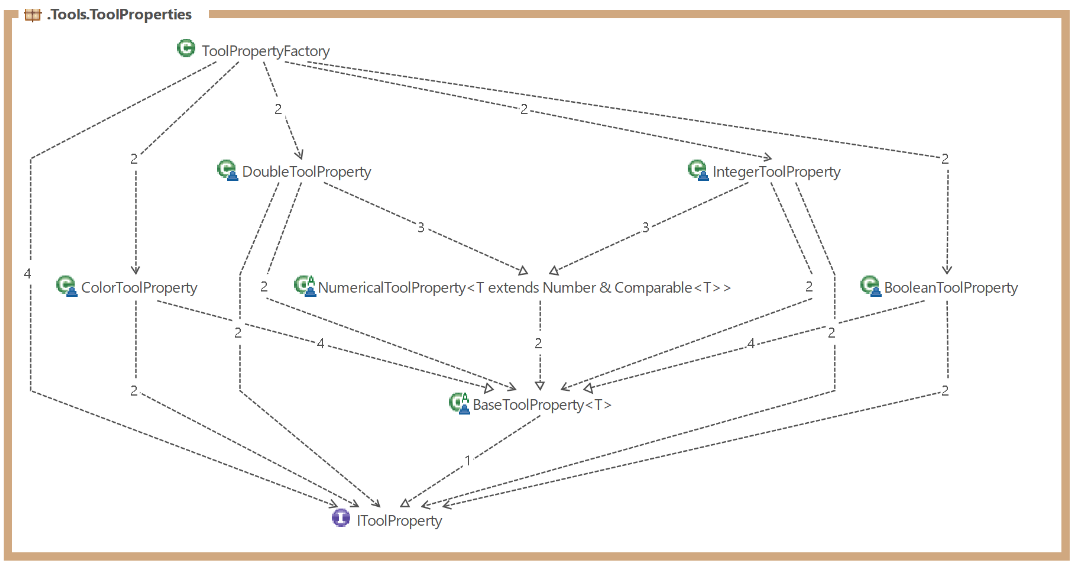


Figure 15: Expanded view of the Tools properties package

9.2.2 Layers

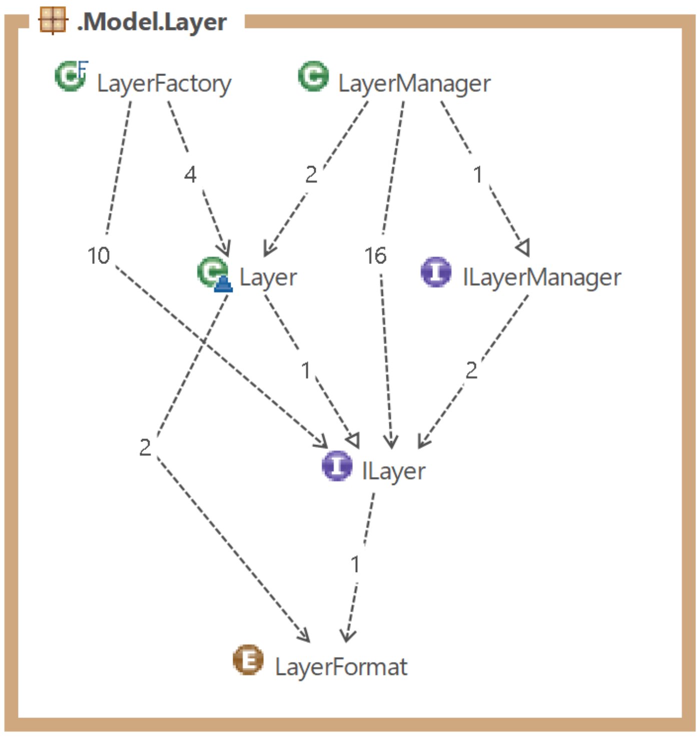


Figure 16: Structure of the Layer package

9.3 Services

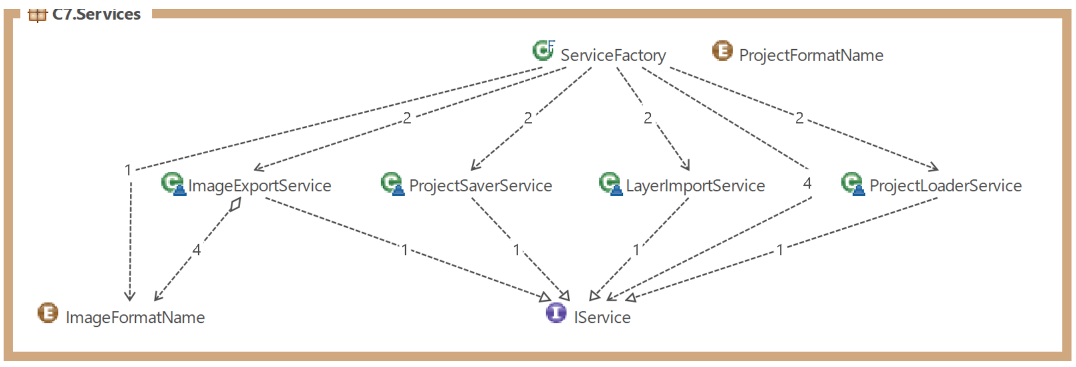


Figure 17: Structure of the Service package

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.

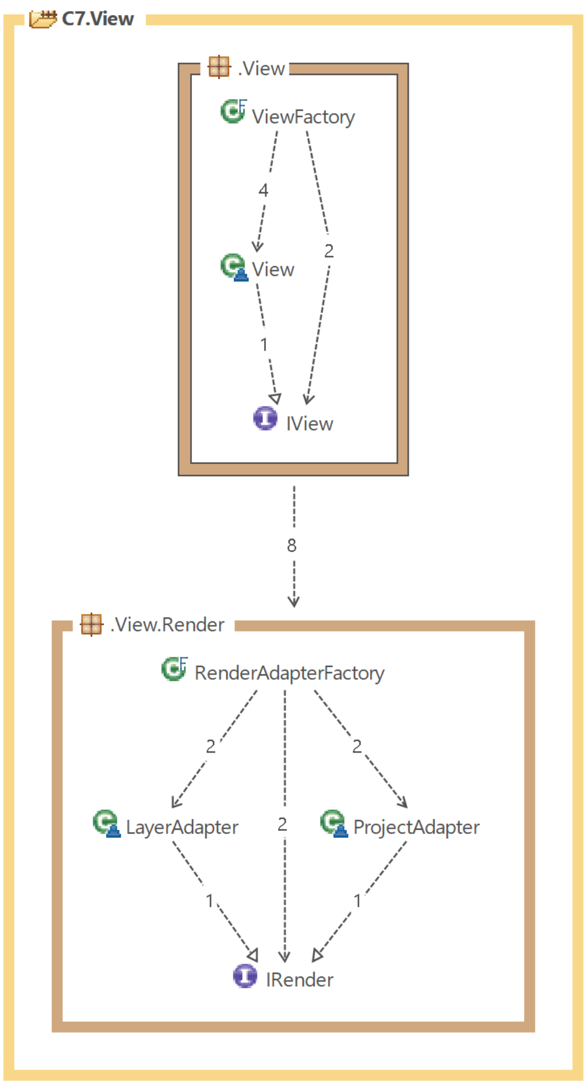


Figure 18: Structure of the View package

9.5 Controller

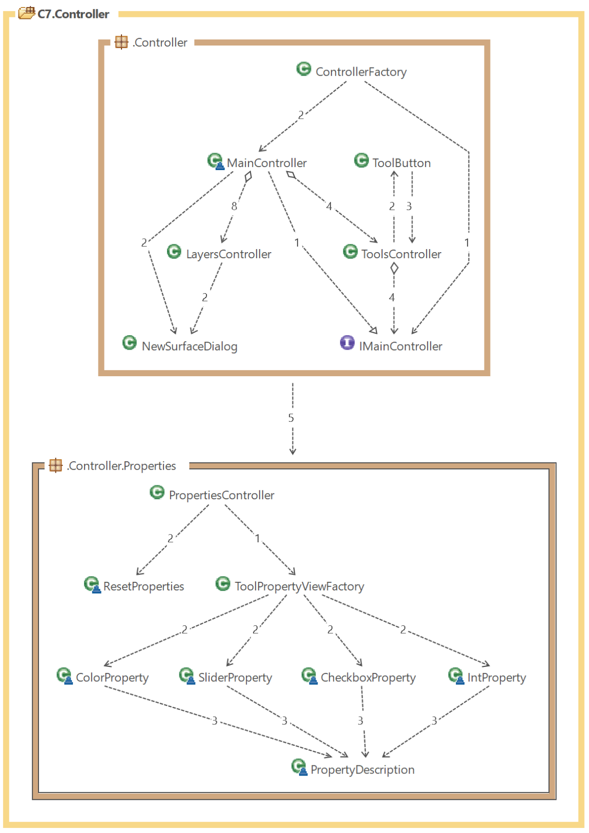


Figure 19: Structure of the Controller package