Software Engineering 2

g2018w_se3_0403

SUPD Report

A simple graphics editor

Team-Members:

Klaus Bareis 01501513

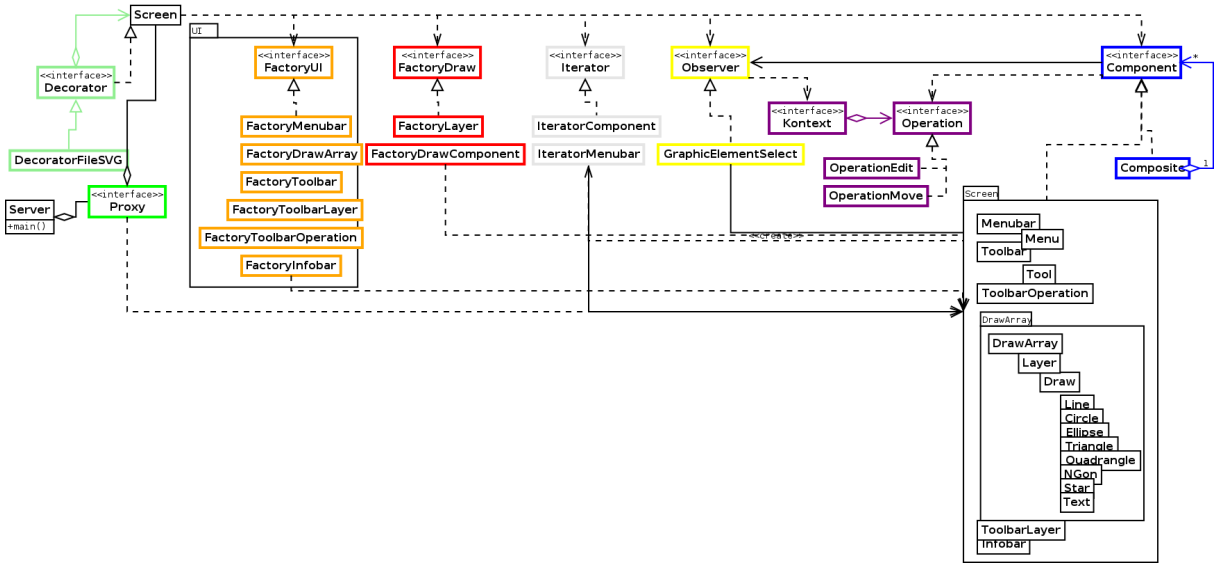
Fabian Schmon 01568351

Margaryta Simkina 01446530

Universität Wien, WS2018

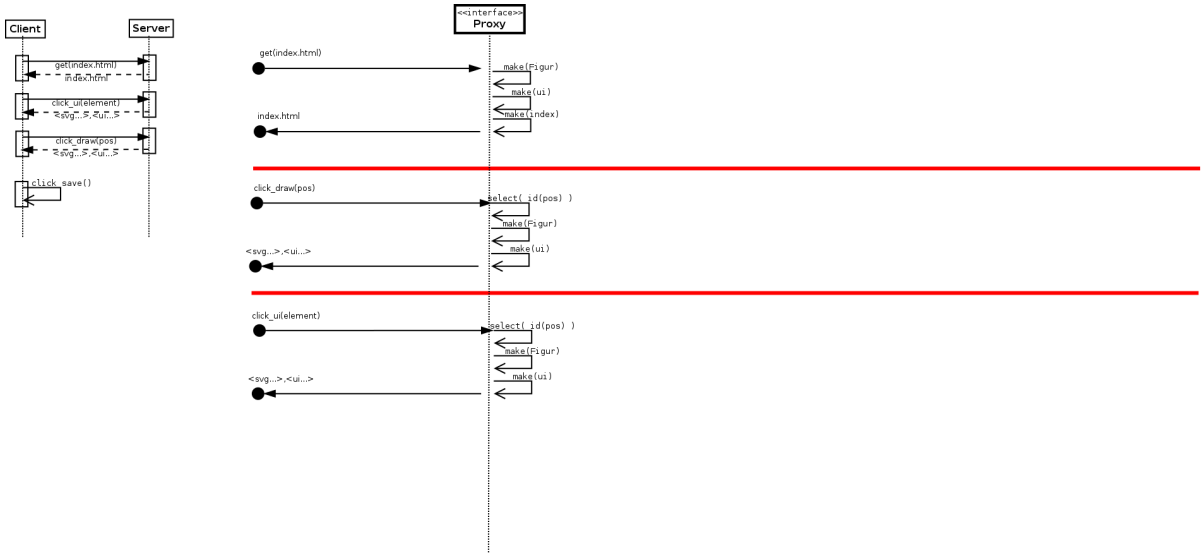
Design Draft 1

Class Diagram



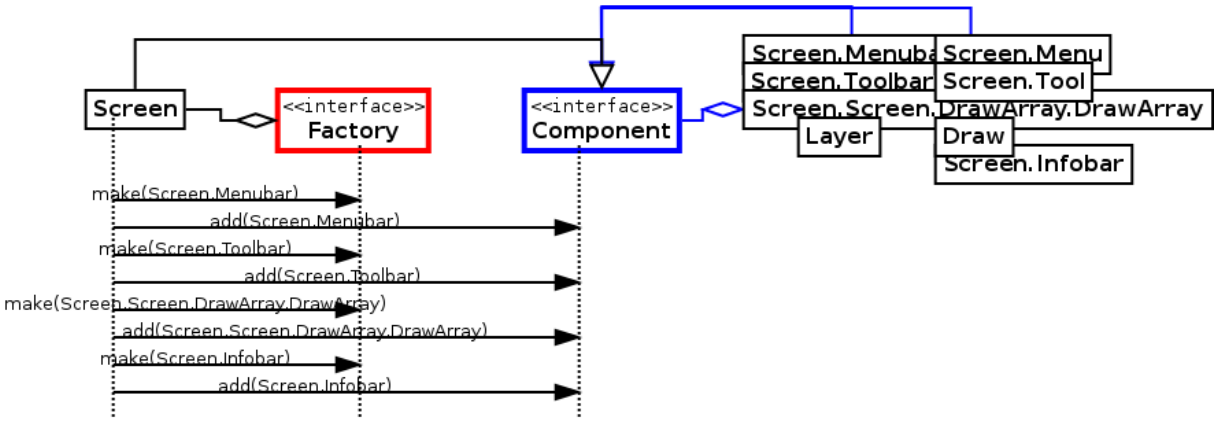
The above class diagram gives a conception view over the main part of this project, the structure of the server. This design follows multiple pattern guidelines, which are explained in detail with the use of sequence diagrams on the following pages. Each of the implemented pattern is illustrated with its own graphical representation.

Sequence Diagram: Client & Server



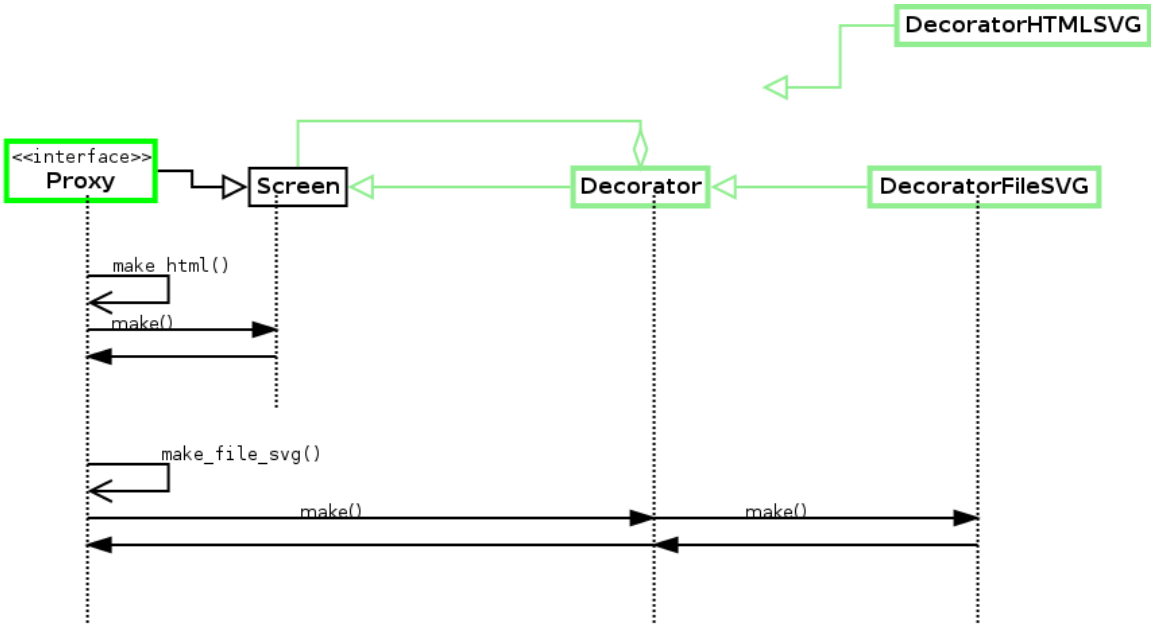
The client – server information flow represents the user inputs and their results. In this design, all operations are processed server-side, the client just sends commands and gets the results back in form of a html file.

Sequence Diagram: Composite



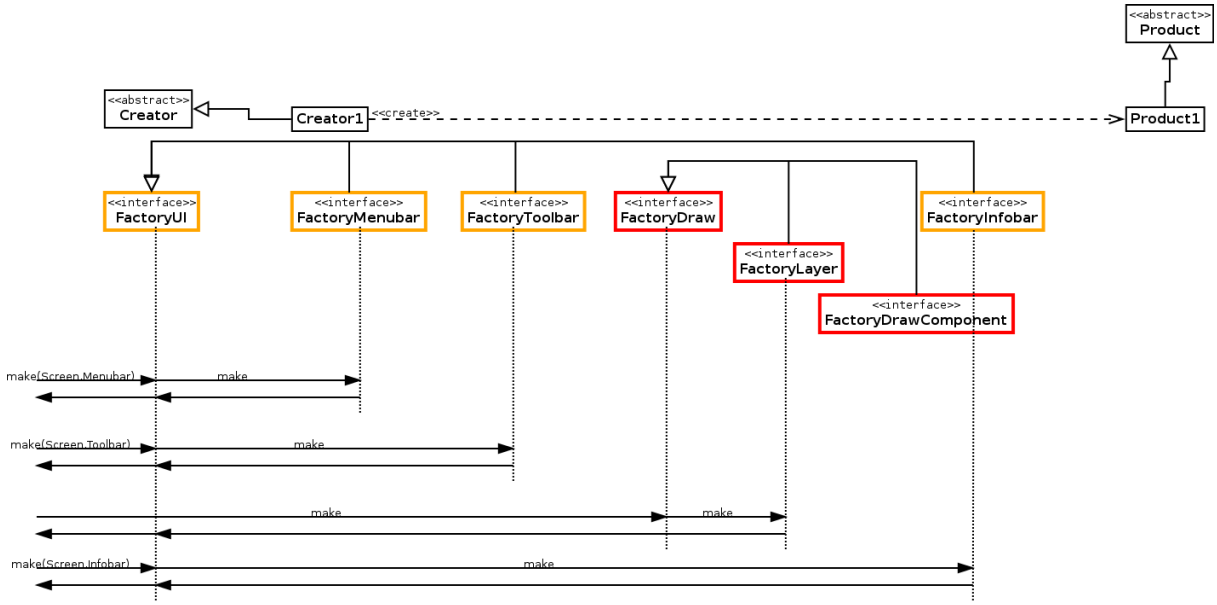
The structure of the “Component” class reflects the composite pattern. One “Component” consists of multiple screen elements which can consist of multiple components as well. This provides the ability to easily perform an action on multiple composites of screen elements.

Sequence Diagram: Decorator



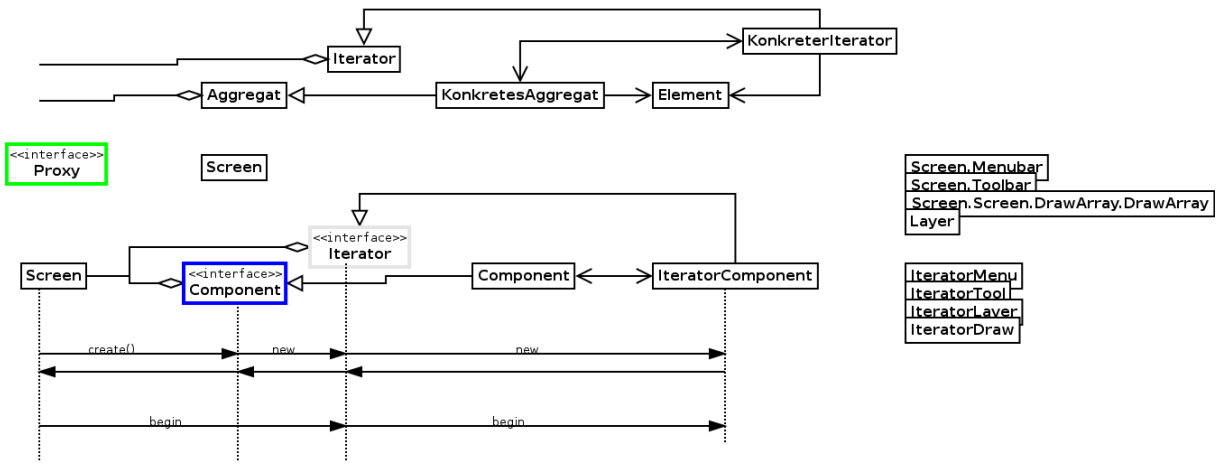
The use of the decorator pattern enables the possibility to change the visualization of different “Screen” elements, without modifying additional code. In this sequence diagram, the SVG-file creation is outlined.

Sequence Diagram: Factory



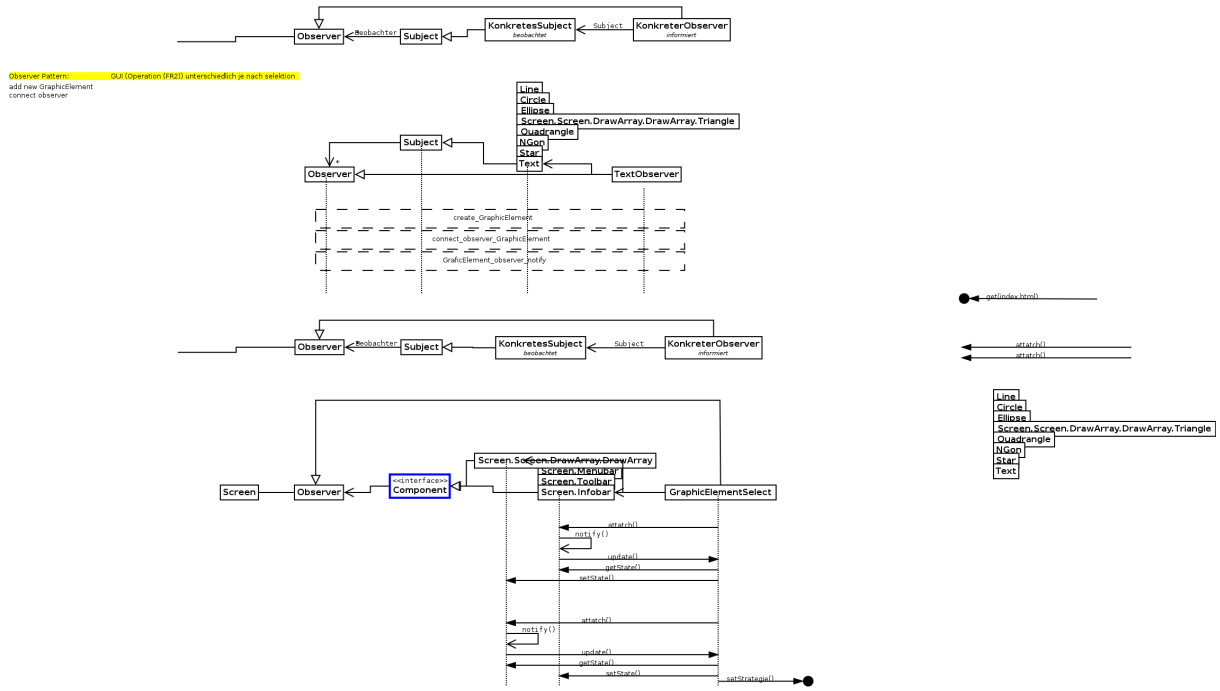
This diagram shows the information flow when different UI elements are created. The "FactoryUI" makes use of the abstract factory pattern, the use of the illustrated interfaces enables easy handling of all UI parts.

Sequence Diagram: Iterator



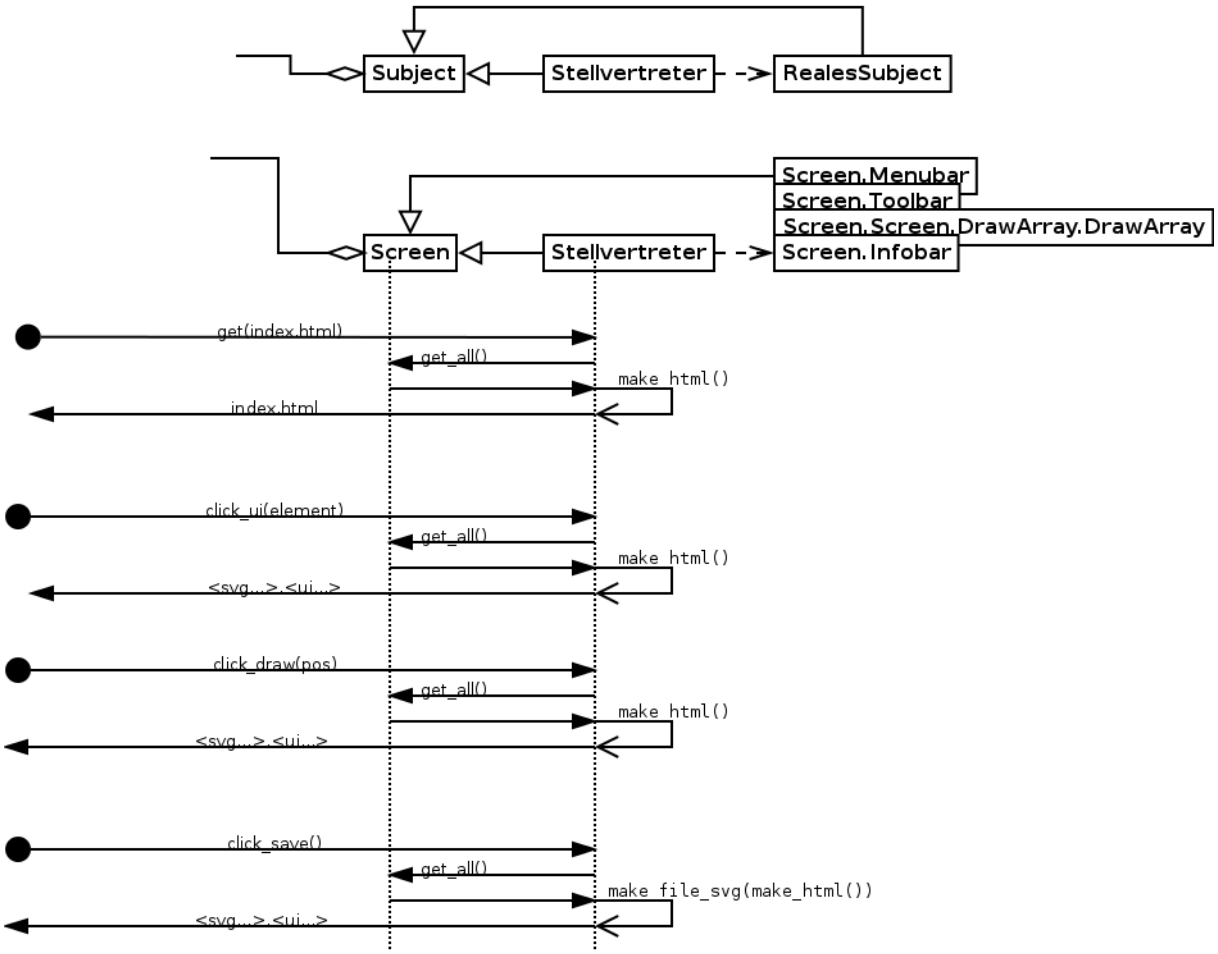
The use of the iterator pattern enables efficient access and navigation through the various screen objects. The information flows from "Screen" to other objects as outlined below.

Sequence Diagram: Observer



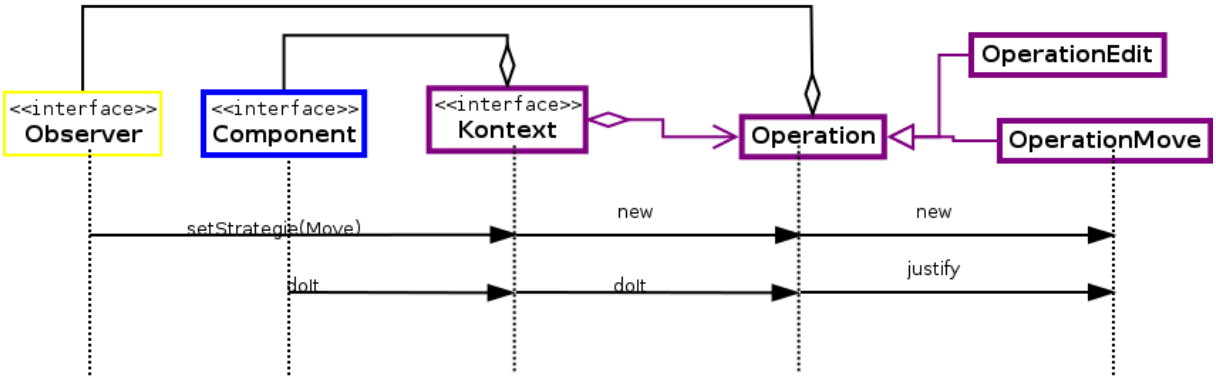
This diagram outlines the communication regarding the observer pattern. In abstraction, the “Component” interface is the subject which is linked to the “Observer”. “GraphicsElementSelect” is the concrete observer which observes a screen element, the concrete subject.

Sequence Diagram: Proxy



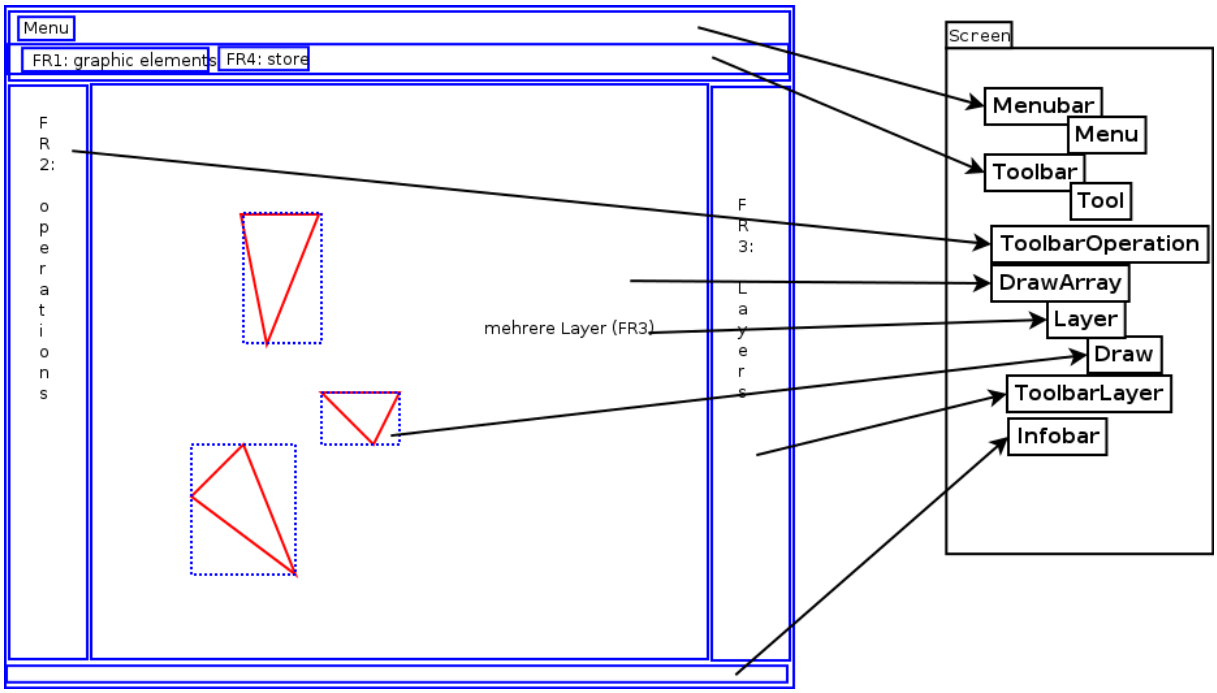
As also illustrated in the “Sequence Diagram: Client & Server” previously, the proxy acts as a central element in the client server communication. The “Proxy” receives commands, and passes the operations to the screen element. Finally, the proxy sends back the html data.

Sequence Diagram: Strategy



This sequence diagram illustrates the implementation of the strategy pattern. The “Context” consists of an “Component” and an “Operation”. Operation consists of “OperationEdit” and “OperationMove”. Depending on the chosen strategy set by the “Observer”, the incoming data will be handled in different ways.

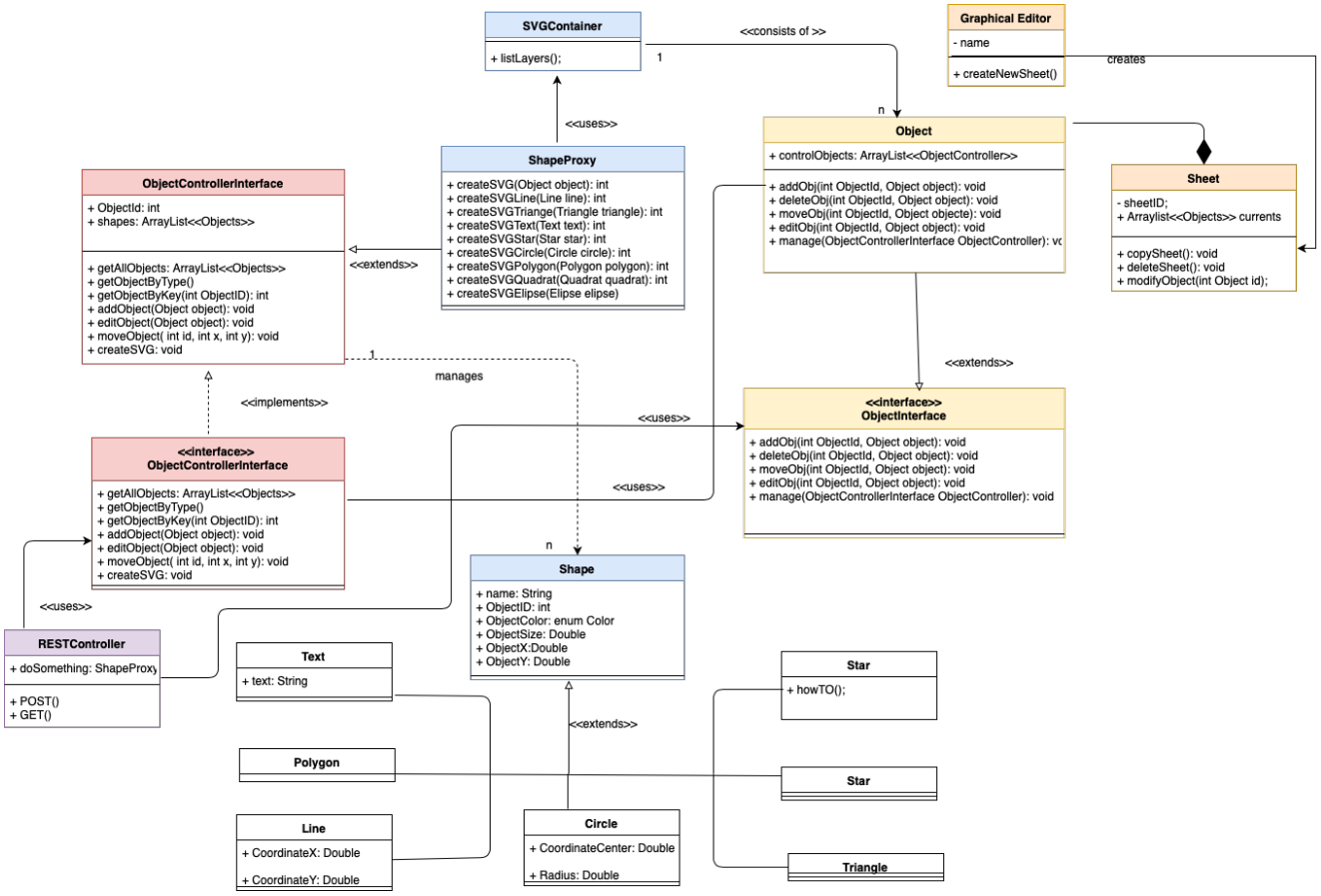
Diagram: UI – “Screen” Class



This graphic shows the conceptional appearance of the User UI that is displayed on the browser. Each graphical element represents a “Screen” element, as implied with the black arrows.

Design Draft 2

Class Diagram



Design Pattern

Observer Pattern

LayerInterface is an abstract class which is further implemented by the class Shape. ObjectControllerInterface is basically an Observer Interface because it is controlling an Observervable (which is in this case LayerInterface). LayerInterface is an object which notifies ObjectControllerInterface about the changes in its state.
For instance, when any action in a class Shape was taken, like, an add/edit/delete/move, a controller gets a notification/update on exact changes and therefore ObjectController updates its Status.

Strategy Pattern

A class Shape is an abstract superclass which has 8 different implementations (basically, 8 shapes). Each time a constructor of a child was called, a constructor of a superclass was

called, too. As well, ObjectControllerInterface is an abstract class which is implemented in an ObjectController.

Iterator Pattern

In ObjectController all the modifications on my Objects will be implemenated through a usage of a ListArray, where while add/delete/edit I access certain Shape (through a key), since they are all saved in ListArray and have ID which is a key. In a class MyShapes there is a method ListAll which goes through created shapes and lists them with their keys – ids.

Proxy Pattern

The Proxy provides a surrogate or place holder to provide access to an object. In our case we decided to prevent unnecessary usage of a memory, so that in a class ShapeProxy a method will be called which is responsible for a creation of graphics through SVG. The method called createShape first checks if Shape was already initialized or not. In the first case it continues to create a shape. Otherwise it first initializes the object and then creates a shape. This prevents initialization of an object twice.

Composite Pattern

Composite pattern will be mainly used for identification of a type of a Shape. For instance, the user can search only for triangles or so. For these purposes a method "classifyShape" will be used. In this case we are making a group of Objects answering certain parameters, so we make a composition of those. This will be done through a class MyObjects, where there is a whole array of lists saved. Object controller is responsible for the group task as well. MyObjects basically represents all the created objects, also provides all the functionality that ObjectController provides.

Abstract Factory Pattern

In our class ConrollerCreator there will be few methods for the qualities of our Object: identShape, identColor, identPlace, and then it's all divided in different classes where those extentions of a shape are to be implemented: AbstractShapeFactory, AbstractColorFactory, AbstractPlacement.

Factory Method Pattern

There is a factory for creation of objects – different shapes. There is an interface BodyCreator. In this class objects of a shape will be not only created, but also checked for Correctness through many assertions to minimize a number of errors during creation of an object. It is responsible for creation of objects without having to specify the exact class of the object that will be created.

Furthermore, software quality will be proven through a requirements list, which should make a procedure of creation most efficient and free from mistakes, minimize waiting times and make modifiability higher. In order to create a Controller, there is a class ControllerCreator, which also minimizes all possible mistakes and is responsible for correct and consistent creation. Besides that, there is a class ShapeProxy , where all the complexity/overhead of the target in the wrapper will be encapsulated.

Decorator Pattern

We do maintain the functionality of each interface. We came to the conclusion that it is not necessary to create “child” classes each time super class was created. And since our Shape has 8 implemenations, it would be irrational to create a polygon when a triangle was called, for example. As well, in an ObjectController we have done edit function, which is used for all our shapes – despite Shape has its children – we extend different edits for different figures, but we don’t have on ObjectController 8 different edits, so it is interchangeable. The Decorator attaches additional responsibilities to an object dynamically. So, whenever a function is called, it will only edit objects dynamically.

API Specification

In the current state of the implementation, there is only one API request from client to server to get the .html file, which is described below:

Title	Get HTML
URL	:8080
Method	GET
Success Response	Code: 200 Contend: { index.html }
Error Response	Code: 404 Contend: { error : “not found” }
Notes	Client has to make the request on port 8080

For an detailed documentation of all the classes and methods of the server, please find the very extensive Javadoc files in the folder “implementation”.

Code Metrics

The implementation for the SUPD assignment contains 39 Java-Class files (including two JUnit test cases). The total number of lines of code is 913, with an average value of app. 23 per class file. The highest lines of code per Java-Class is 125, while the lowest number per file is 1.