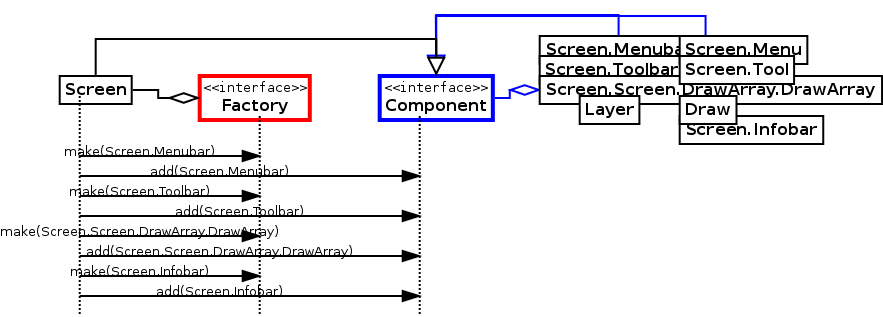
Semester Project  
A simple graphics editor

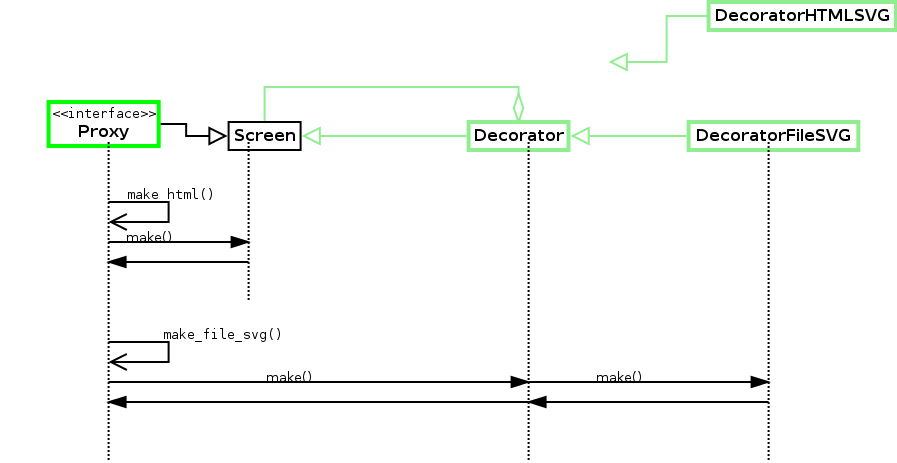
Design Patterns

**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.

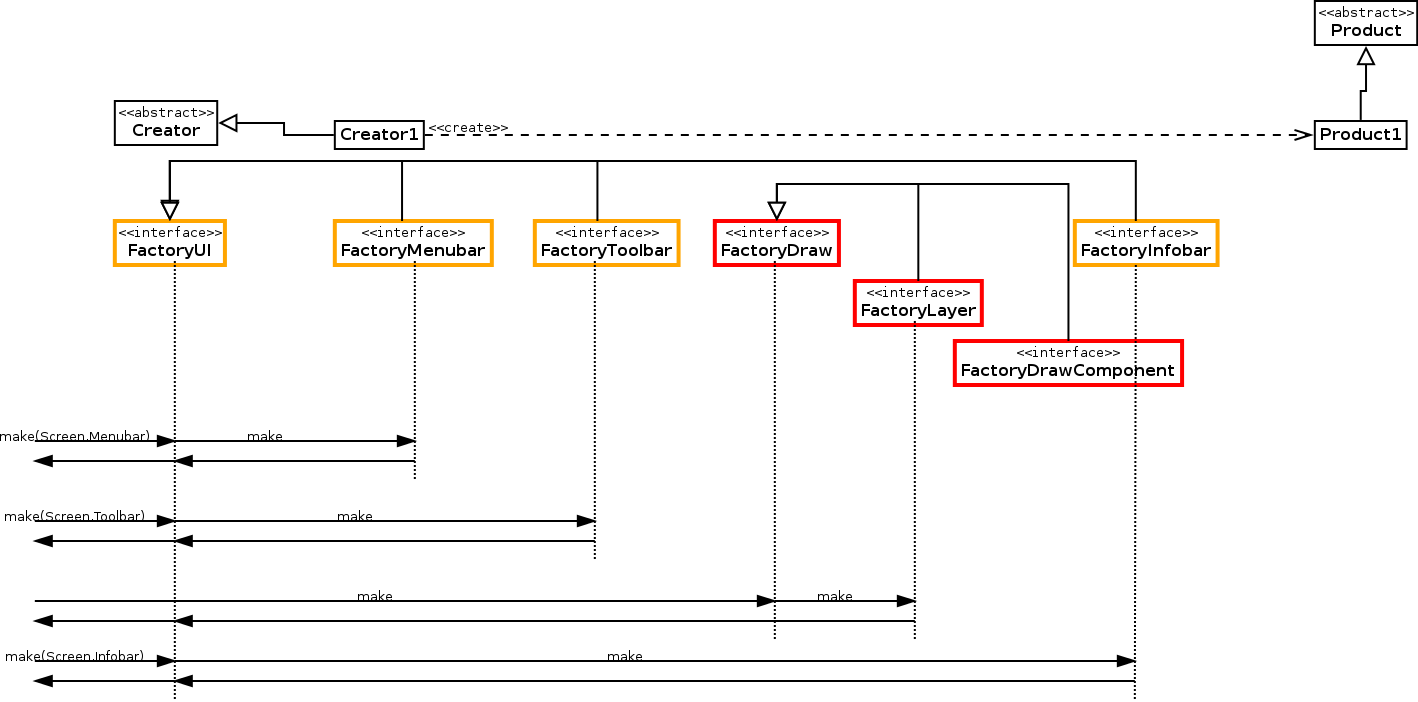
In our project, we decided to implement two different Decorator Patterns because it is considered to be extremely good practice if reading some news in this area, it is on a way to replace inheritance.

**First decorator Pattern** is a change of color of our scheme: on a Toolbar left, one can change it from light to dark and back without adding additional dependencies and implementing unnecessary methods.

**Second decorator Pattern** is position handler: we got a default one by simply clicking on a button, but also a Decorator one, when one gives parameters to create it in a. certain place. This wasn’t done with extension, but with external additional functionality in design pattern.

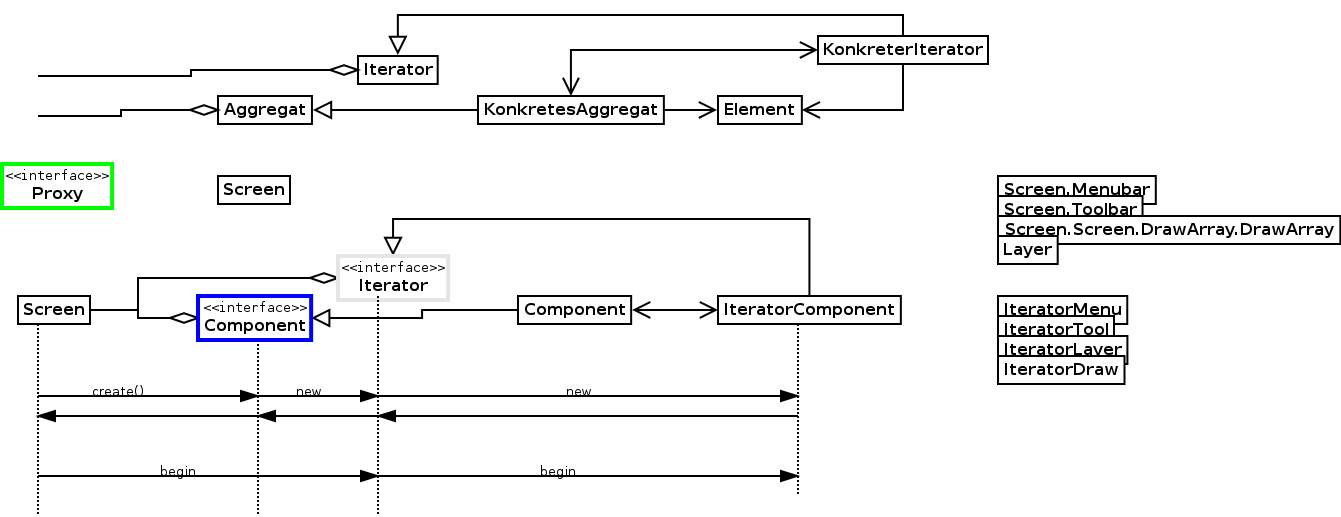
Further, when function will be called, it will still go through a default constructor, but code itself won’t be copied.

**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.

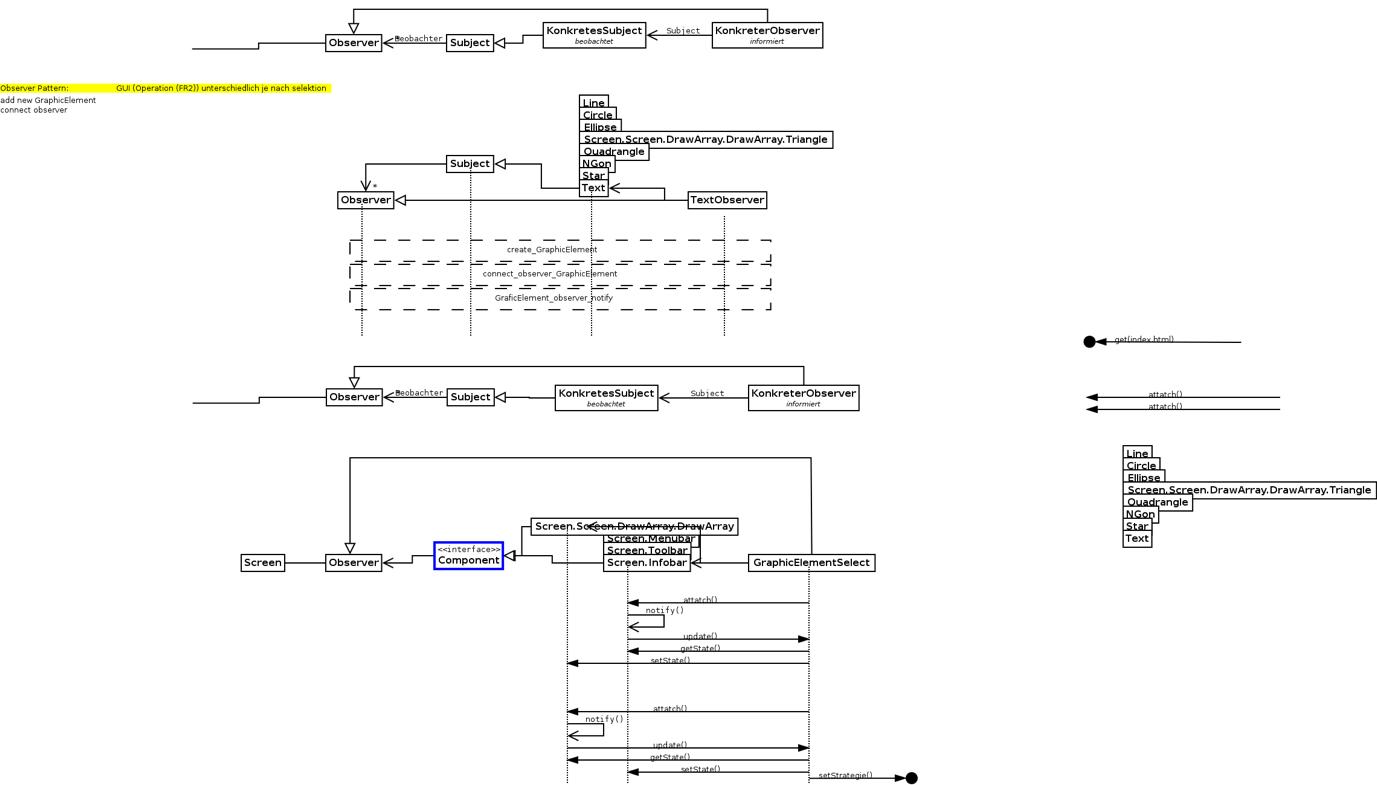
**Sequence Diagram: Iterator**



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.

4

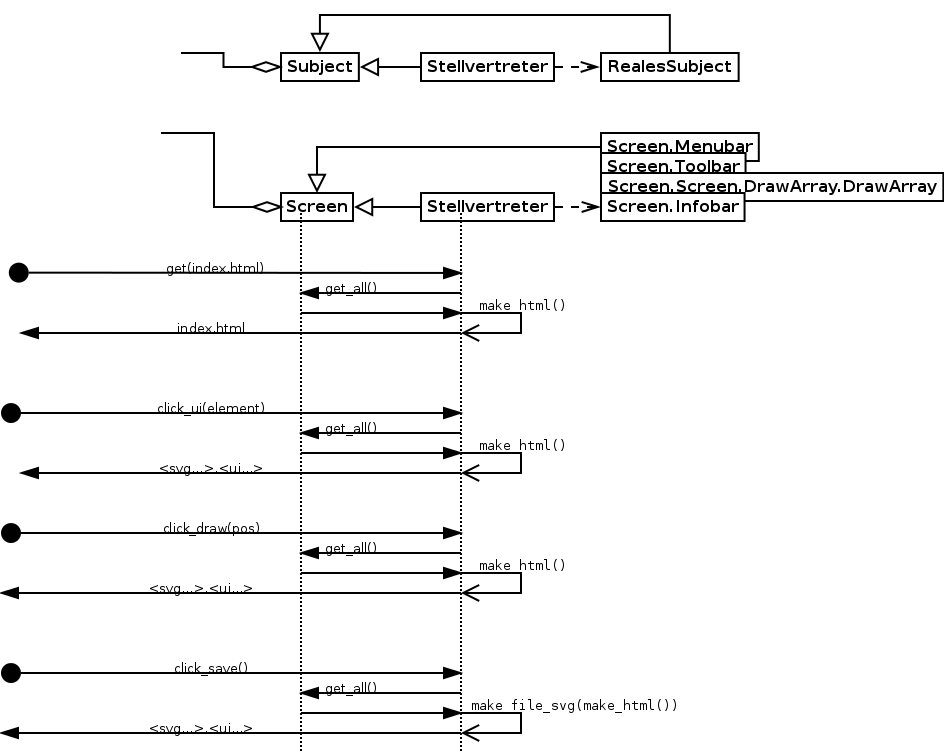
**Sequence Diagram: Observer**



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.

5

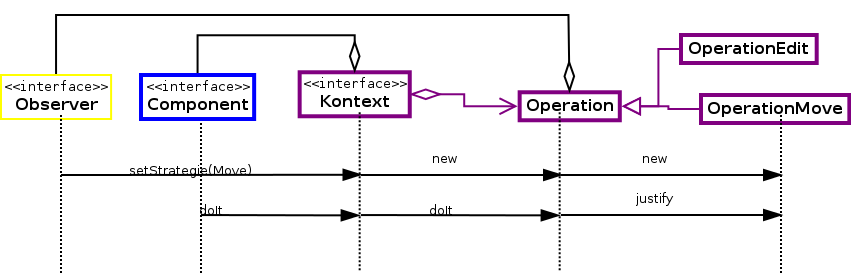
**Sequence Diagram: Proxy**



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.

6

**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.

**Abstract Factory Pattern**

Our AbstractoryFactory is initially responsible for creating multiple shapes with same attribute – in our case we chose color as an attribute which is to be given in a menu, and after that color is basically switched, we are creating shapes with the same color until we turn on the next color basically. In that perspective, we use *ColorAbstractFactory(params)* for specific “family of shapes” based on one factor together. Otherwise we use *DefaultAbstractFactory()*. In both ways, further on, we will create other factories inside of those factories, and Color one will still call ColorDraw with necessary parameters through Optional.

**Dependency inversion principle**

We decided as well to add dependency inversion Principle as a very good practice among current approaches: this enables way of decoupling software modules, getting rid of strong dependencies and being able to “reach out” from different level to another one. In our case we used BeanContainer which stores Context object inside. In that way, we are able to use our context object (which, for instance, we used a lot for creating/editing shapes), we would be able to use Context Object from any part of a project basically.

**Functional & Quality Requirements**

// For the creation of the project, we met together as a team and considered together how we would like to design the project

DID WE?

**FR1:** For FR1, we first composed and thought about what the graphic elements should look like, what parameters they should have and what methods we need. Fort hat we used brainstorming and couple mockups. We created a class Screen which includes two toolbars with many functionality, main window and couple more layers. It is possible to choose between dark and light scheme. We enable to create shapes in different location, size and color using up toolbar. Left toolbar is responsible for working on shapes and changing screen design.

**F2:** For FR2, we too have all assembled and considered how to implement these methods, what effects the methods have on the graphic elements, what needs to change. These methods changed more often during the first phase of the project and everyone worked on them. All four functions are implemented and clear to see on a main window. Move function is inside of edit: by changing parameters of X and Y coordinates, we change position of our shape.

* Create – Bereis and Simkina:

It is possible to create a shape by clicking on a button with a shape name by default or by using color / coordinates choice .

* Delete –Schmon

Deletion works ny clicking on an object that needs to be deleted and then clicking a button on a left toolbar.

* Edit & Move – Bereis

By choosing necessary parameters in a dropdown menu bar, we enable to both edit and move shapes.

FR3: We support 2 Layers – Bereis

// DOES IT EVEN WORK?

**FR4:** Saving Function is implemented in SVGFormat - Schmon

**QR1-QR2:** Each team member was responsible for creating readable and self explainatory code, otherwise it should have been properly commented

// LETS ADD COMMENTS AND JAVA DOCS?

**Q3:** We tried to avoid Code Grouping, limit line and class length/size, huge hierarchy and consistent indentation. Still, structure needed a *refactoring* which we implemented by coupling, creating new classes and dividing functionality into smaller pieces, but not dependent on each as in *inheritance*. We reduced switches size by some returns. Which did not work 100%, we still have big dependencies but that way we avoided too long and repetitive code. What we tried to organize well , was folder structure and proper classes and variables names.

**Q4:** We used defensive programming such as checking *input function parameters*, having a *proxy* which took care of further UI mistakes, as well, we used *optional* in order to avoid null returns.

**Q5:** We set up all necessary application type, concerns, technologies and so in a very beginning. Only things were added later by each team member were quality attributes. Our crosscutting concerns were Instrumentation and Logging (through a debug, we could also watch out threading).

**QR6:**

Observer Pattern: Bereis

Strategy Pattern: Bereits ?

Iterator Pattern: Bereis

Composite: Bereis

Factory: Bereis

Abstract Factory: Simkina

Decorator: we have two – one by Schmon and one by Simkina

Dependency inversion principle – Simkina

coding practices