VU Software Engineering 2

DEAD

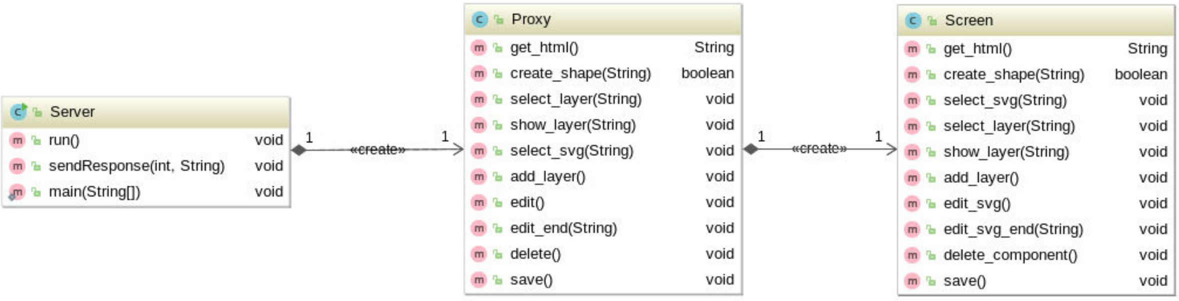
Personal data:

First name, Surname:	Klaus Bareis 01501513 Fabian Schmon 01568351 Margaryta Simkina 01446530
Date:	Januar 2019

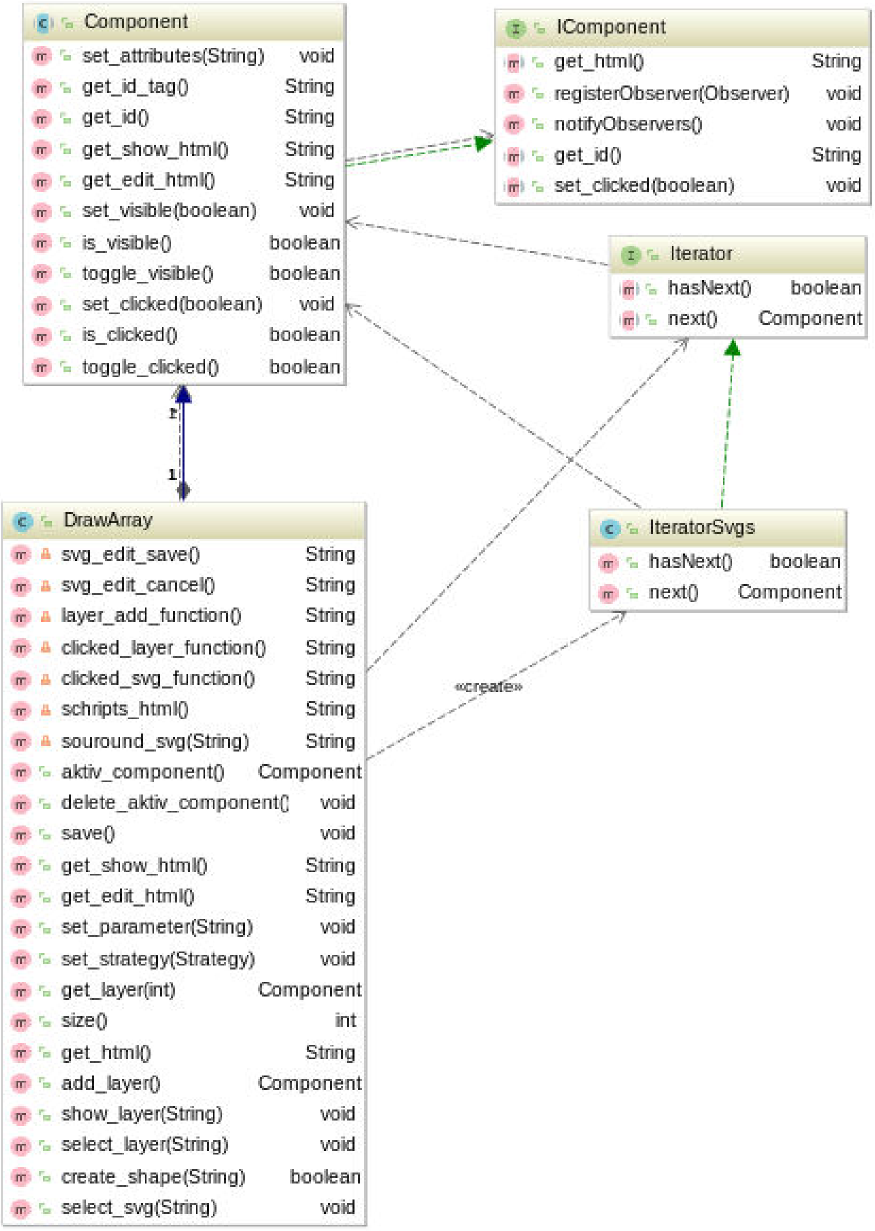
Design patterns

• Proxy Pattern

The proxy pattern was meant to control the client's access to the screen. In normal operation, the contents should be transferred as html, when saving as svg. Saving was moved to the server. For this reason, the pattern is only partially implemented.

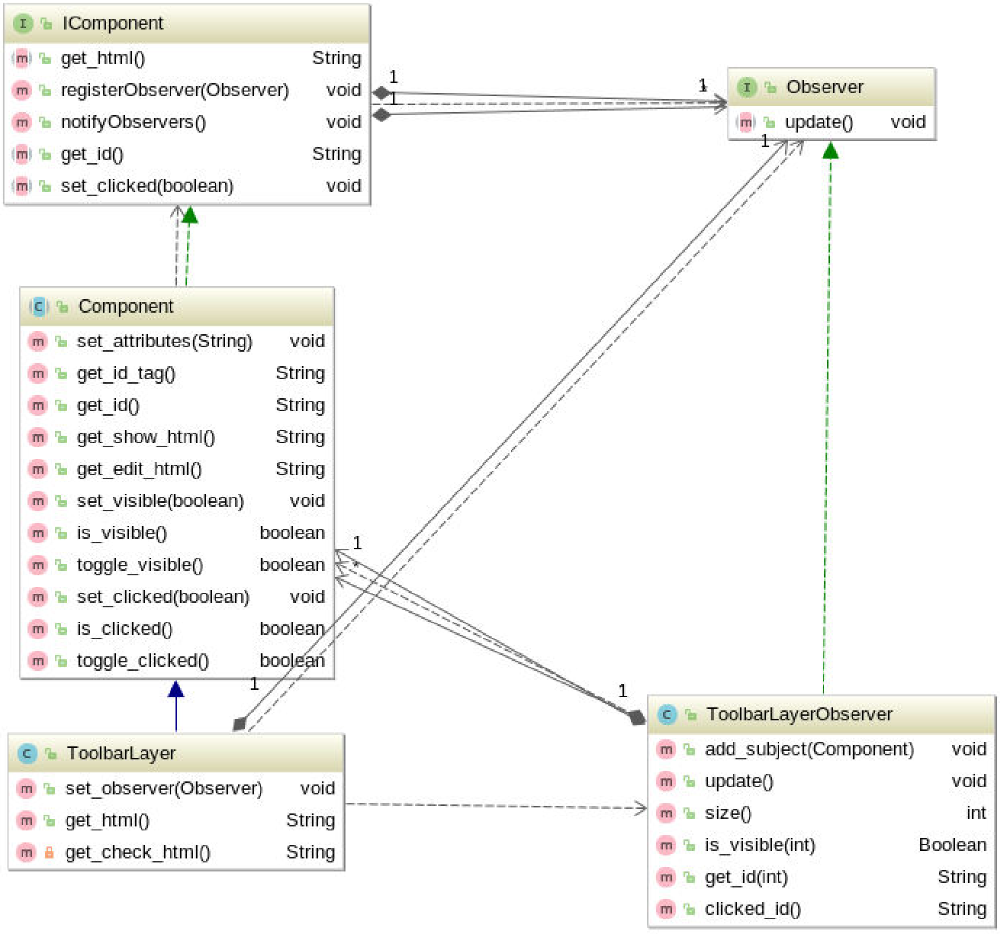


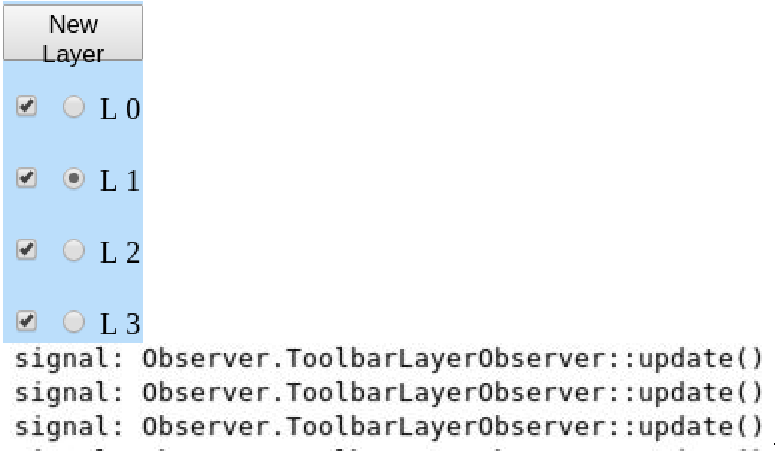
- **Iterator Pattern** We use Iterator to access objects, and navigate through them on a Screen. The information flows from screen.



• Observer Pattern

One use of the observer in our application is that of ToolbarLayerObserver. In the toolbar there is a button to create new layer in/on the screen. A click on this button informs the screen that he should create a new layer. Also, enable and disable, and switching between layers via this signal/slot concept is monitored.

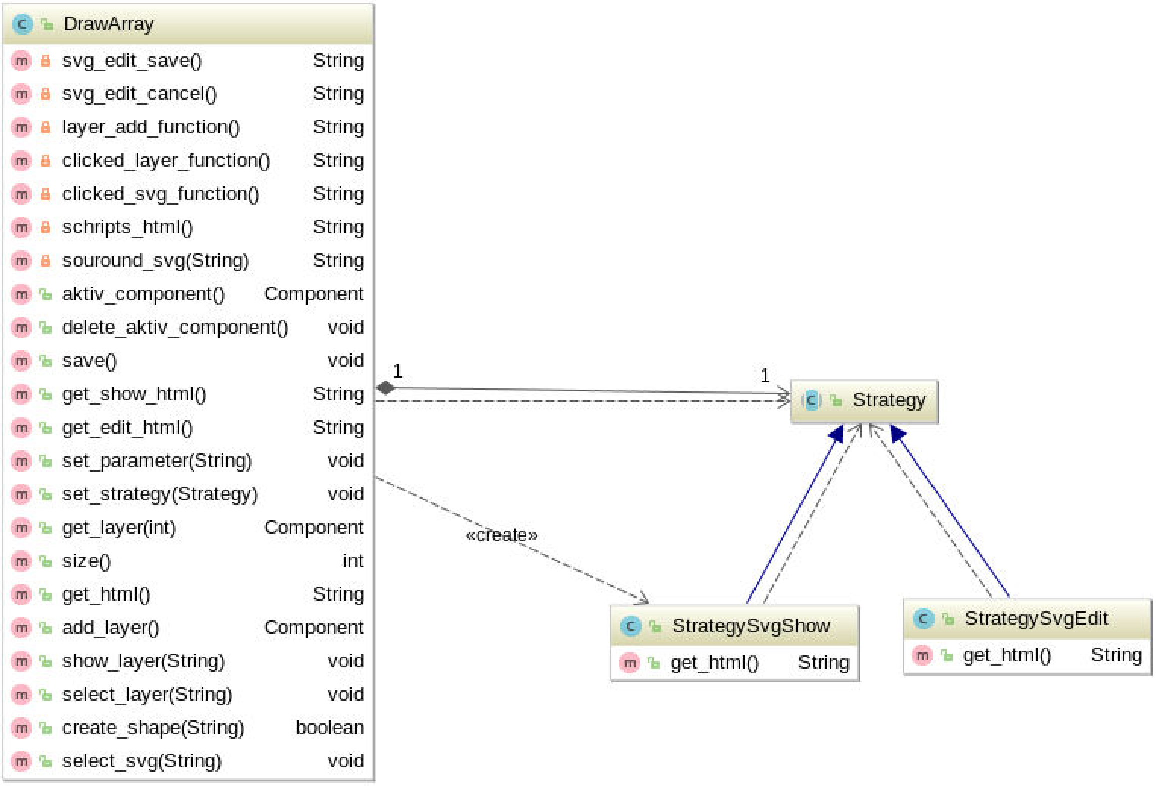




```
signal: Observer.ToolbarLayerObserver::update()
signal: Observer.ToolbarLayerObserver::update()
signal: Observer.ToolbarLayerObserver::update()
```

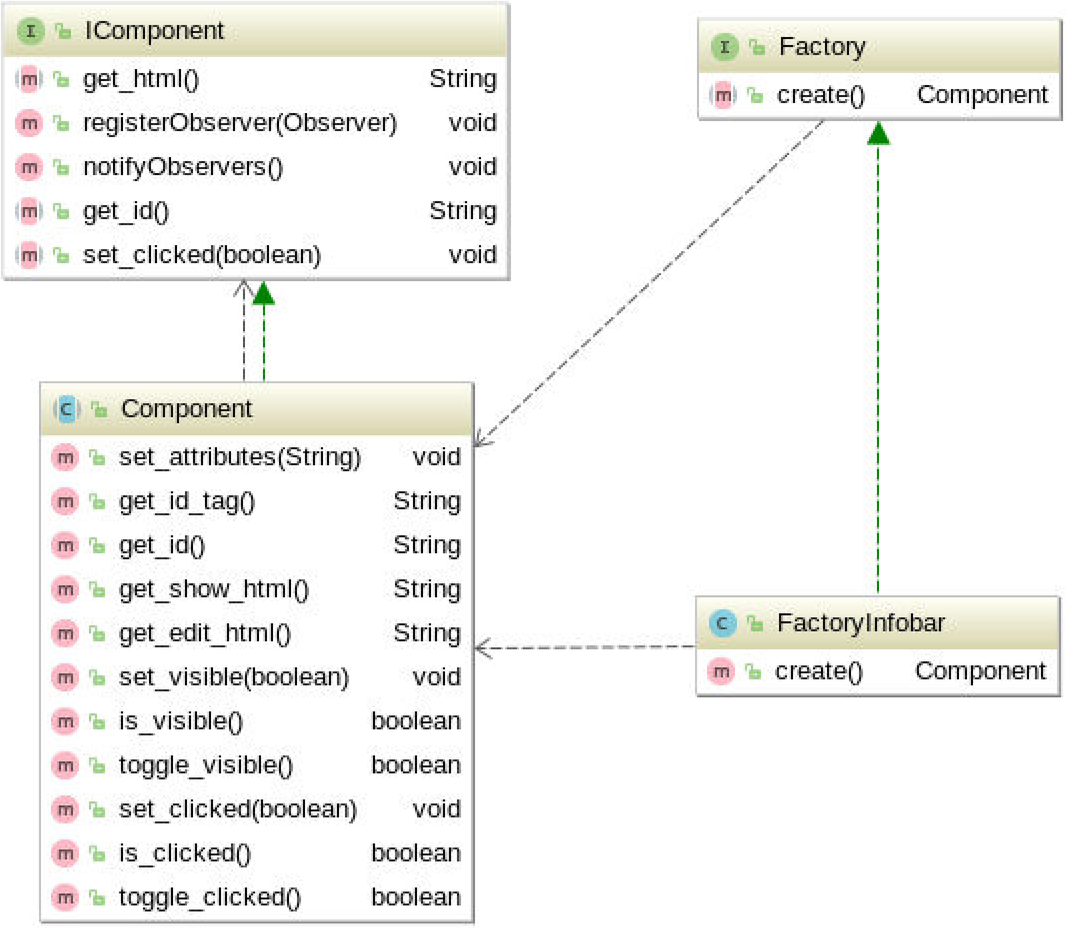
• Strategy Pattern

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.



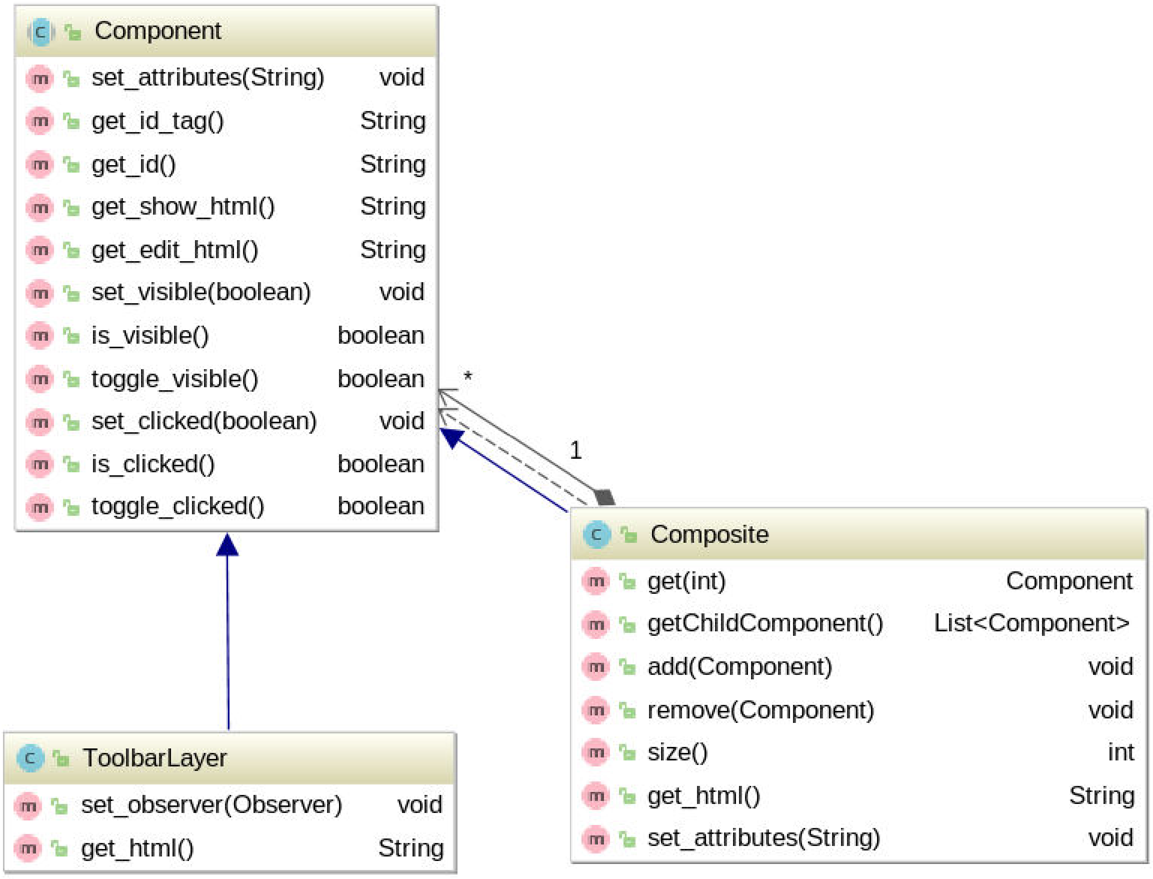
• Factory Method Pattern

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.

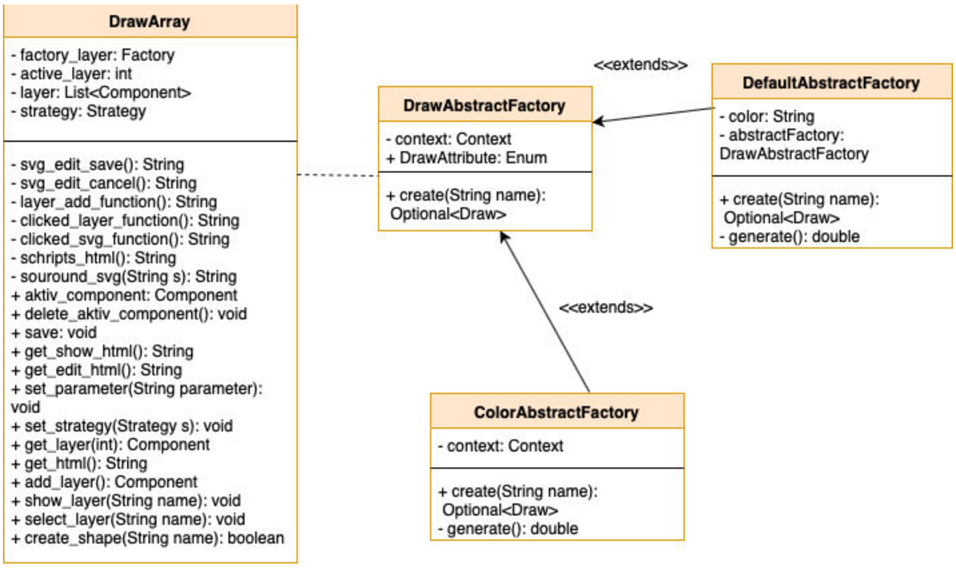


• Composite Pattern

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.

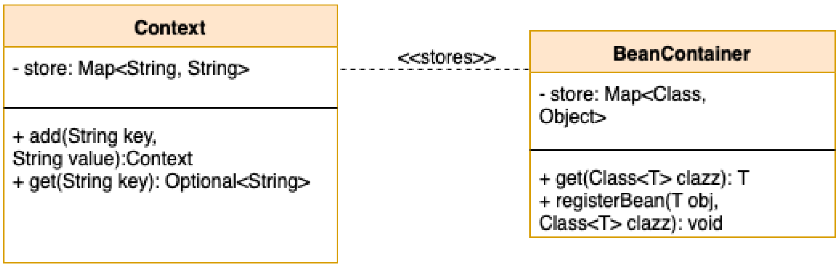


- **Abstract Factory Pattern**


DrawArray
- factory_layer: Factory
- active_layer: int
- layer: List<Component>
- strategy: Strategy
- svg_edit_save(): String
- svg_edit_cancel(): String
- layer_add_function(): String
- clicked_layer_function(): String
- clicked_svg_function(): String
- schripts_html(): String
- souround_svg(String s): String
+ aktiv_component: Component
+ delete_aktiv_component(): void
+ save: void
+ get_show_html(): String
+ get_edit_html(): String
+ set_parameter(String parameter): void
+ set_strategy(Strategy s): void
+ get_layer(int): Component
+ get_html(): String
+ add_layer(): Component
+ show_layer(String name): void
+ select_layer(String name): void
+ create_shape(String name): boolean
<<extends>>
DrawAbstractFactory
- context: Context
+ DrawAttribute: Enum
+ create(String name): Optional<Draw>
DefaultAbstractFactory
- color: String
- abstractFactory: DrawAbstractFactory
+ create(String name): Optional<Draw>
- generate(): double
<<extends>>
ColorAbstractFactory
- context: Context
+ create(String name): Optional<Draw>
- generate(): double

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**

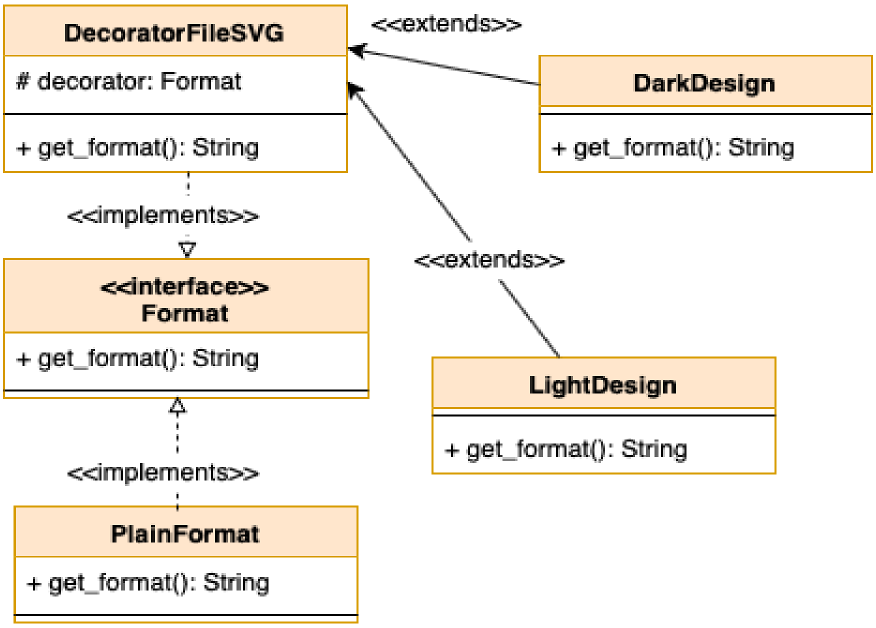

Context
- store: Map<String, String>
+ add(String key, String value):Context
+ get(String key): Optional<String>
<<stores>>
BeanContainer
- store: Map<Class, Object>
+ get(Class<T> clazz): T
+ registerBean(T obj, Class<T> clazz): void

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.

- **Decorator Patter**

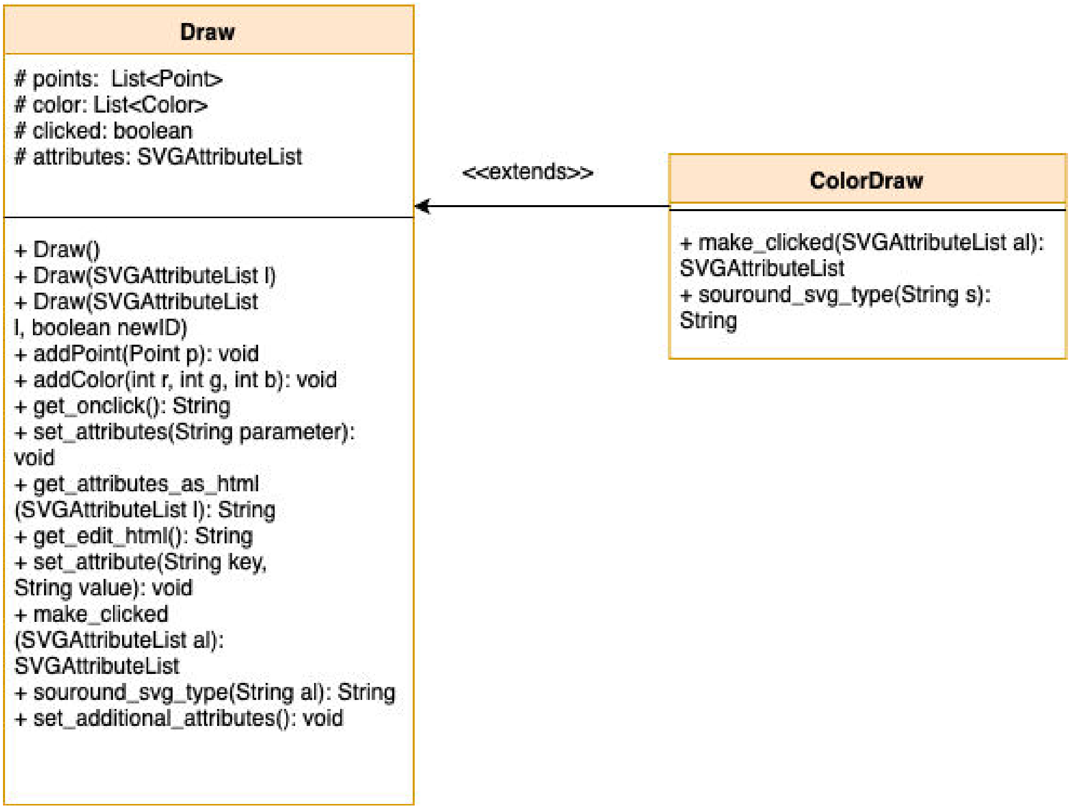
In our project, we decided to implement two different Decorator Patterns because it is considered to be extremely good practice if reading some news in this area, it is on a way to replace inheritance.

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.


Draw
# points: List<Point>
# color: List<Color>
# clicked: boolean
# attributes: SVGAttributeList
+ Draw()
+ Draw(SVGAttributeList l)
+ Draw(SVGAttributeList l, boolean newID)
+ addPoint(Point p): void
+ addColor(int r, int g, int b): void
+ get_onclick(): String
+ set_attributes(String parameter): void
+ get_attributes_as_html (SVGAttributeList l): String
+ get_edit_html(): String
+ set_attribute(String key, String value): void
+ make_clicked (SVGAttributeList al): SVGAttributeList
+ souround_svg_type(String al): String
+ set_additional_attributes(): void
<<extends>>
ColorDraw
+ make_clicked(SVGAttributeList al): SVGAttributeList
+ souround_svg_type(String s): String

Coding practices

1. *Readability & Clearness.*

   We always tried write code that simple to read and which will be understandable for developers, but still a bit of optimization was necessary at the end.

2. *Architecture first.*

   We approached first the architecture and a Diagram, and then made an attempt to build the whole project on our patterns and answering the planned structure, only a little bit of changes were then done.

3. *Simple.*

   We tried to keep it simple and self-explanatory, using names, that are speaking for theirs functionality.

4. *Comment.*

   In general, we avoided unnecessary comments and also tried to make it already clear for others. But still, for some functions, that were not that easy, we added necessary comments.

5. *Reviews.*

   What I found important, we were giving each other reviews and if necessary, corrected mistakes, bugs of each other, which helped a lot.

6. *No deep nesting.*

   We tried to use as little coupling as possible, but as much, as necessary.

7. *Structured and short.*

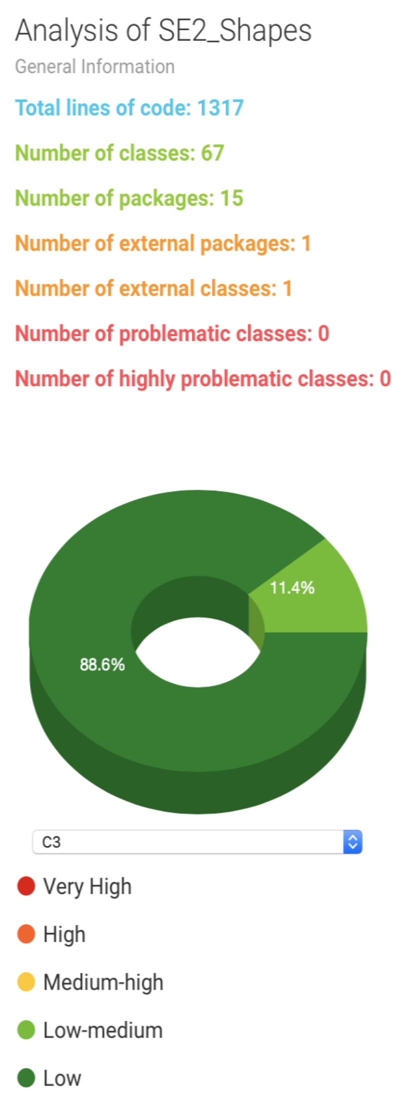
   We used limited line and class length, separated classes in packages and followed the structure.

Defensive programming

In a project there are some Exceptions or Try Catch blocks, but rather we could have extend it to a bigger coverage to avoid crushs. We tried to do basic checks for input variables though.

Code metrics

1. Code Metrics General Analysis



Analysis of SE2_Shapes

General Information

Total lines of code: 1317

Number of classes: 67

Number of packages: 15

Number of external packages: 1

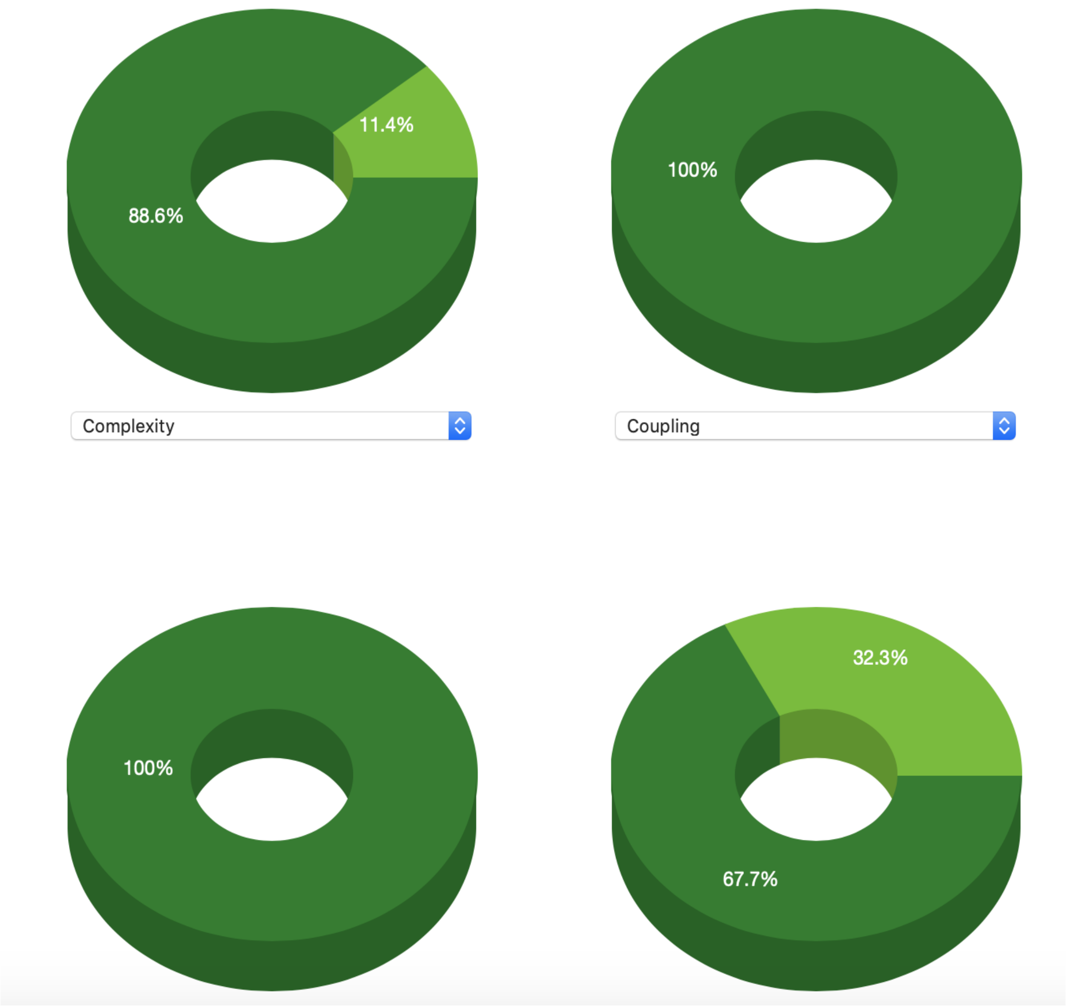
Number of external classes: 1

Number of problematic classes: 0

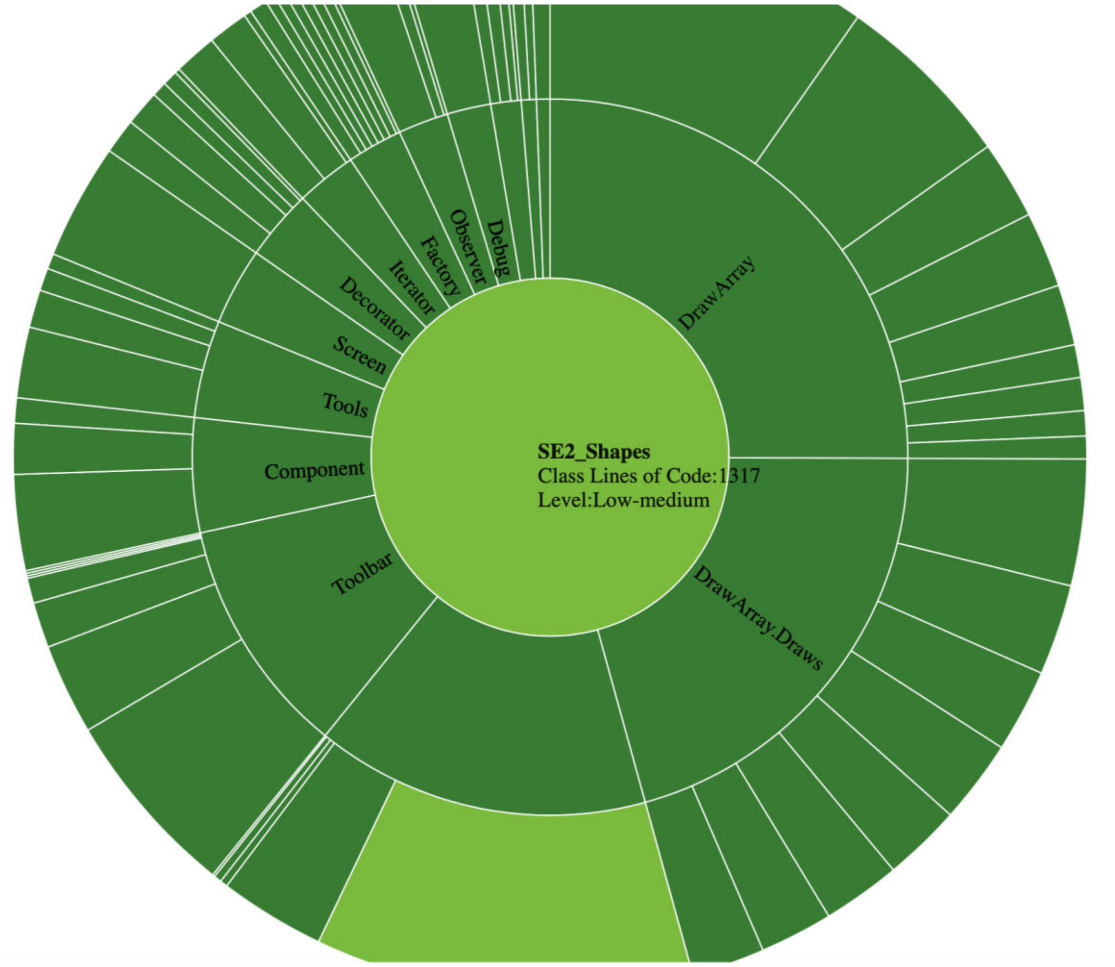
Number of highly problematic classes: 0

2. Distribution of Quality Attributes

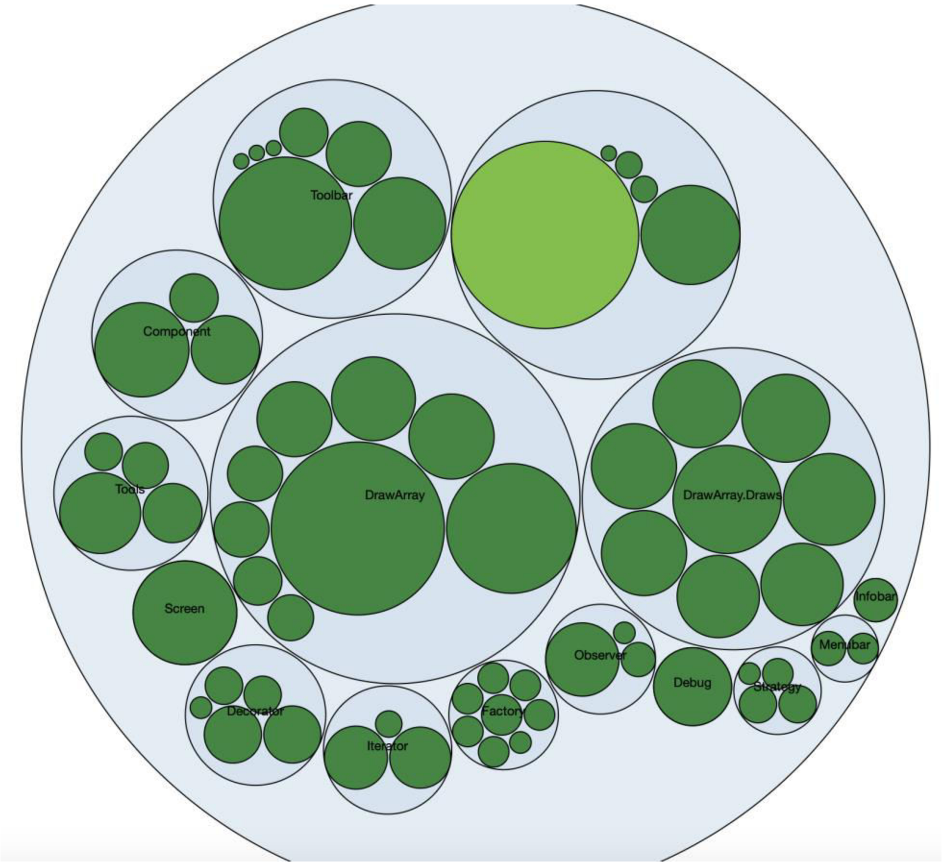
Complexity, Coupling, Cohesion, and Size



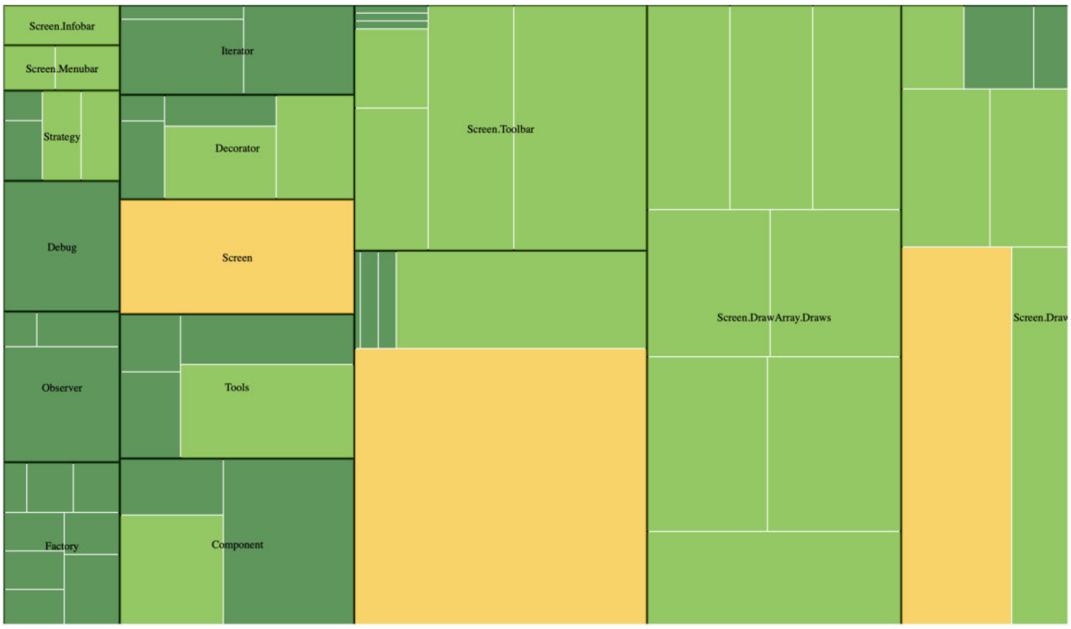
3. Metric Values in Sunburst Chart



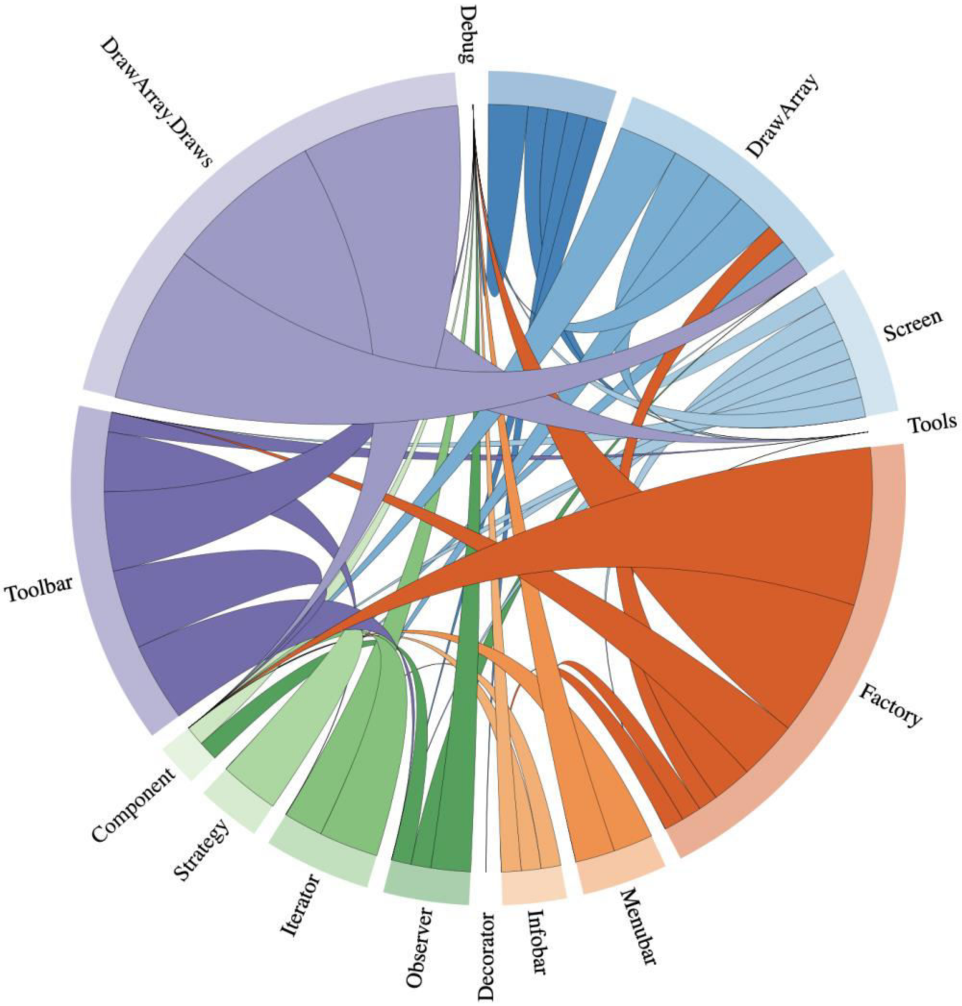
4. Metric Values by Packages



5. Metric Values in Treemap Chart



6. Package Dependencies



Team contribution

Design: Bareis
Basic Implementation: Bareis

Functional Requirements (FRs)

FR1

• lines	Bareis
• circles	Bareis
• ellipses	Bareis
• triangles	Bareis
• quadrangles	Bareis
• n-gons	Bareis
• stars	Bareis
• text	Bareis

FR2

• addition	Bareis. Simkina
• deletion	Schmon
• editing	Bareis
• movement	(Bareis over editing)

FR3
Bareis

FR4
Schmon

Quality Requirements (QRs)

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

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: We implemented few tests for logic (Composite, Component, Factory), as well as a basic test creation.

QR7:

• Basic files

Server	Bareis, Schmon, Simkina
Toolbar	Bareis, Schmon, Simkina
ToolbarLayer	Bareis, Schmon, Simkina
ToolbarOperation	Bareis, Schmon, Simkina
Menubar	Bareis, Schmon
Draws	Bareis, Schmon

• Observer Pattern	ToolbarLayerObserver	Bareis
• Strategy Pattern	StrategySvgEdit	Bareis
	StrategySvgShow	Bareis
• Iterator Pattern	IteratorLayer (not used)	Bareis
	IteratorSvgs	Bareis
• Composite Pattern	for Screen	Bareis
• Proxy Pattern	Proxy for HTML	Bareis, Schmon
• Abstract Factory Pattern	ColorAbstractFactory	Simkina
	ColorDraw	Simkina
	Color	Simkina
	DrawAbstractFactory	Simkina
	DefaultAbstractFactory	Simkina
• Factory Method Pattern	FactoryDrawArray	Bareis
	FactoryInfobar	Bareis
	FactoryLayer	Bareis
	FactoryMenubar	Bareis
	FactoryToolbar	Bareis
	FactoryToolbarLayer	Bareis
	FactoryToolbarOperation	Bareis
• Decorator Pattern	Format	Schmon
	DecoratorFileSVG	Schmon
	PlainFormat	Schmon
	LightDesign	Schmon
	DarkDesign	Schmon
• Decorator Pattern 2	ColorDraw	Simkina
	DrawArray	Simkina
• Abstract Factory Pattern	ColorAbstractFactory	Simkina
	DefaultAbstractFactor	Simkina
	DrawAbstractFactory	Simkina
	Proxy	
• Dependency inversion container	BeanContainer	Simkina
	Context	Simkina

QR8: We have implemented the whole project using Maven.
QR9: In a package Runnable we got our JAR file.

HowTo

Application can be launched by running a JAR in a terminal which is `java -jar Shapes.jar.`
Otherwise one can /implementation/src/main/Server.java "Run" -> "as Application" and then Application runs on port :8080. After that browser should be open on localhost.