API Specification

Silhouette

Group 12:
Mats Engelien
Lars Erik Faber
Håkon Marthinsen

Contents

1 Introduction

Link til GitHub: https://github.com/norwen3/Silhouette

1.1 Group Description

1.2 Delegation of Work

Description of Work

Each students writes about scenarios they have contributed with... Workload...

2 Background

TODO: Include some client-code here

There is no hiding that there are plenty of java web frameworks already existing out there, and with almost everything tried, it is hard to figure out a new exciting solution without reinventing the wheel. People generally use web frameworks for x core reasons:

- To not have to repeat code over and over, making the process a lot more efficient.
- To establish a structure in the applications so it sets a standard.
- To gain access to additional features that further simplifies the process.

If the framework does not provide these, it's questionable whether it is actually useful. However, if it does not stand out in any way, people will choose the already existing frameworks out there. That's why it's important to take insperation from these already established frameworks, but not a direct copy paste. Below are some key web frameworks that Silhouette has taken note of.

Spring focuses heavily on simplifying the components required to build a web application, and with Spring Boot, the user won't have to think about configuration and just start coding. Spring offers cloud support and is popular among companies that want to use containers. Spring uses "Beans" that are like regular java objects, but they are managed by a Spring IoC Container to provide inversion of control, that let's the user manage dependencies more effectively.

Java Server Faces or JSF is another important framework to look at. It is component based, which means that... It enforces clean MVC and offers Inversion of Control. Since the user is not writing plain low level HTML, but rather abstrac representations of UI-components, the abstractions are converted into the appropriate representation of HTML depending on the device and version of the internet.. Something like that.

Bla bla bla, what all these have in common is x, but Silhouette does y.. Silhouette takes insperation in that it provides dynamic conent. Silhouette also uses POJOs (Plain Old Java Objects) as components. Silhouette can also read from file and generate the page. The high level APIs in Silhouette greatly reduces the HTML code as it automagically generates code.

3 Method

"A section to describe the method under which the framework has been created (can be shared). This section should contain a description for each of the stages given in the "Framework Design" lecture (API Design Specification, API Design, Implementation and User Testing). You should describe how you have worked, not what you produced, in this section."

3.1 API Design Specification

Forklar hvordan vi kom fram til Tankesett av hvordan API-et skal gå fram, høy-nivå og lav-nivå prinsipper

From the get-go we knew our framework needed to generate HTML somehow, and decided that it would be best to

From the get-go we knew that our framework needed to support both the declarative and stylistic sides of webdesign, so we started forming two bas

3.2 API Design

Forklar hvordan vi kom fram til Builder pattern... Hvordan API-et faktisk ser ut

When designing the APIs in Silhouette, we started writing client-code to see what would work the best. We knew it was important to preserve the advantages of OOP, giving the user control by making objects and manipulating them in various ways. This meant that users have the ability to generate blocks of HTMl quickly with the tools we already know from OOP. We decided to write some client-code to see what would satisfy our requirements, and this would be one of the earliest iterations.

SHOW CLIENT-CODE HERE

The idea of x was important, and you can see that with the way it's set up and all. But another thing that we couldn't dismiss was the fact that we would get way too many classes if we had kept going the route of making one-to-one relations between classes and HTML tags, following the rule of 30 (Insert Source Here!), so we came up with a secondary solution. Say, for example, the user wants to make a video on their page. The video tag in HTML requires some smaller tags inside it that specifies file and filetypes and controls etc.. an we saw it as an opportunity to hide away these smaller tags within methods. This proved to be a good solution as it both cut down on the total amount of classes, and the user had less things to worry about but still achieve the same things. Eventually we settled on sone one-to-one classes, and some methods to generate simpler HTML code...

Design Decisions

Following best-practices for making APIs, for example, from

Later on once we got some feedback from other groups we came across another problem. Users expected to be able to chain multiple methods after eachother. But that was just not possible with how we originally structured our methods and classes, that's where builder pattern comes in. Builder pattern aided the framework in two areas: One, it served as a tool to hide the backend-implementation from the user, and two, it allowed for method-chaining which was exactly what we needed. This mean that previously, the user had to type:

SHOW CLIENT-CODE HERE

But now, with the builder-pattern in place, the user can do this:

SHOW CLIENT-CODE HERE

Personal Decisions

3.3 Implementation

Implementation was done in various stages as we had many ideas for how we wanted the resulting framework to behave. After a lot of discussion with the group, we began writing client-code to help us visualize the API to get better ideas on how we got incorperate the implementation. After several supervisions with the lectruer, we were advised to us a builder-pattern, which works wonders for the type of tasks we want to do, however it makes implementing a bit of a chore, as this means usually our

source-code will double. Getting to understand how this pattern works was not easy, since as we were writing our API-code we did not have a full understanding of the structure of method-calls/chaining, nor how to really instantiate these objects. This led to some early errors like giving each builder-class a specific name, and not implementing it correctly.

After getting to understand these things there are other issues that appear throughout. We have some global attributes that all objects in the API share, so it's logical to just write the method once and be done with it, however this does mean that casting between classes becomes necessary. This is possibly because we opted for internal static Builder classes. We felt that was the most logical and direct connection we could make between an objects and its builder.

Lastly, we needed to form a system to generate the actual. The big idea was to make some sort of compiler that puts strings together to make the necessary web files. It later became apparent that we needed one compiler for each API, as the logic and syntax are wastly different. As it stands, the compilers will either read html and css in the form of Strings, or plain java old objects (POJOs). Then once the user is done defining their pages and stylings, they will inform the respective compiler to perform a compile and generate the web files. If these files already exist, the compiler should just overwrite them to apply the new changes.

3.4 User Testing

User testing started in the middle of the development of Silhouette, right after all the classes that was need to builde a HTML document (the logical structure part) and the same for the CSS part. At this time the code was just object oriented programming and it flet easy to use, but it was alot of unnecessary writing and to a degree hard to follow.

After the first user testing with another group, we didn't get alot of crtical feedback on whats "good" and "bad" excepted for the things that we knew about, but we got feedback on the logical structure part and that is one of the things we needed.

After some time we knew what we wanted the code ot looke like and how to simplify it more, and that is when we started to use "Builder pattern" instead of the heavy object coding we had previously. This made it more readable for the eyes and work with.

Description of setup

We sent a jar file containing the framework to group and started with some simple instructions. Firstly, they needed to start a new project and add our framework/library to their project from the project structure menu. Once this was done, they were ready to start coding inside the main method.

At first, we let them figure out the process themselves, but once they met issues and got stuck, we gave explanations to what each code block represented and guided them through the process. The next section will cover the resulting code and feedback from our testing.

Frist user testing session

Right after they had tested our framework and completed the setup, we asked them to give their initial reaction and opinion. They said, "It was pretty good once we got the hang of the class names" which was a relief to hear. It seemed to us that they found the APIs to be intuitive, as they quickly grew familiar with the main classes. We then followed up with a few questions regarding their experience, and they told us they could easily expand further on their website with enough practice.

Second user testing session

4 Design Process/Results

4.1 Design Specification

Forklar hva High Level Design Principles...

Design Patterns

Builder Pattern We were very quickly guided towards using the Builder pattern when we first had decided on the Framework-type. The Builder pattern, unlike the Abstract Factory pattern, is a lot more like a ”Fluent interface”, meaning it relies heavily on method cascading. The builder pattern is useful as it attempts to seperate the construction of an object from the way it is represented, this way we can use the same process to create many different looking objects that are fundementally similar. We need to use the builder pattern when we want immutable objects. We want immutable classes because they have a wide range of benefits when we are developing an API. They are simple to use, they are synchronization safe, we only need one constructor, and they are great for maps(which we're using) to name a few.

The main problem with implementing the builder pattern is the sheer amount of extra code required. The lines will most likely more than double. However, those extra lines of code give us much more readable code and design flexibility. We exchange parameters in the constructor for much prettier method calls.

4.2 Project Structure (File Structure)

Type Reference Documentation

Link to type doc...

4.3 Client Code

Scenarios and Solutions

Scenario 1 Make two rulesets, one that is a regular ruleset and one that is a grid ruleset. Give each ruleset a unique selector. For the regular ruleset, add a blue background and change the text color to #32a852. For the grid, define three columns and two rows of varied size. Lastly, apply both of the rulesets to a Container of type ”header”.

Scenario 1 - Proposed Solution

```
RuleSet color = new RuleSet(".color");
color.addRule("background-color", "blue");
color.addRule("color", Color.Hex(#32a852));

Grid grid = new Grid("#grid");
grid.setColumns("1fr", "100px", "2em");
grid.setRows("50%", "120px");

Container myHeader = new Container("header");
myHeader.applySelector(".color");
myHeader.applySelector("#grid");
```

Scenario 2 Make a table whose size changes dynamically, add values to the header row and add values to the rest of the rows as they are generated. Apply a class to the table and set a header color for the table.

Scenario 2 - Proposed Solution

```
Table table = new Table();
int col = headerArray.length;
int row = array.length;
RuleSet color = new RuleSet(".tableClass");
color.addRule("background-color", Color.RGB(255,255,255));
ColGroup headerCol = new ColGroup();

table.setSize(row, col);
table.applySelector(".tableClass");

for(int i= 0; i<row; i++){
    for(int j= 0; j<col; j++){
        while(i<1){
            table.insertHead(i,j, headerArray[j]);
        }
        table.insertValue(i,j, array[j])
        if(i<1){
            headerCol.addCol(col, color.toString());
        }
    }
}
```

Scenario 3 Make a frontpage and two article pages and share a common stylesheet between them. Add a simple ruleset that targets the body element and give it a lightgreen background to see that the changes have applied to all the pages. Lastly, generate the code.

Scenario 3 - Proposed Solution

```
HTML myWebsite = new HTML("example website");

Page frontPage = new Page("frontpage", "My Front Page");
Page articlePage1 = new Page("article", "My First Article");
page articlePage2 = new Page("article", "My Second Article");

StyleSheet style = new StyleSheet("style.css");

RuleSet bg = new RuleSet("body");
bg.addRule("background-color", "lightgreen");

style.append(bg);

website.link("style.css");
website.addPage(frontPage);
website.addPage(articlePage1);
website.addPage(articlePage2);

website.initialize();
```

Scenario 4 Make a simple form consisting of three input fields for username, password and submit. Label each of the fields accordingly and give the fieldset a legend of ”Login Credentials”.

Scenario 4 - Proposed Solution

```
FieldSet myFields = new FieldSet();

Input username = new Input("text");
username.addLabel("Type Username");

Input password = new Input("password");
```

```
password.addLabel("Type Password");

Input submit = new Input("submit");

myFields.addFields(username, password, submit);
myFields.addLegend("Login Credentials");
```

Scenario 5 Make a Container of type header and give it a navigation bar with links to all pages previously created. Make the navigation bar into a CSS flexbox so that all anchors be laid horizontally and justify the content to be ”space-around”.

Scenario 5 - Proposed Solution

```
Container header = new Container("header");
Container nav = new Container("nav");

Anchor link1 = new AnchorBuilder("Front Page",
    "frontpage.html").build();
Anchor link2 = new AnchorBuilder("First Article",
    "article1.html").build();
Anchor link3 = new AnchorBuilder("Second Article",
    "article1.html").build();

nav.addElements(link1, link2, link3);

FlexBox flexbox = new FlexBox("header nav");
flexbox.setFlexDirection("row");
flexbox.setJustifyContent("space-around");
```

Scenario 6 Make a simple article page about yourself. Write your name in a level 2 heading at the top of the article, then add an unordered list containing containing key points about you. Lastly, add your email in bold text below the text.

Scenario 7 Display a local video on your website and make sure that it supports at least three different file formats.

Scenario 8 Make a questionnaire consisting of two questions. Make it so the first question has two possibilities and one correct answer, and so the second question has three possibilities with multiple correct answers. (Radio and Checkbox)

Scenario 9 Remove underline for all anchors inside nav Containers and change the text color to red on mouse hover for each anchor.

Scenario 10 Give the header a fixed gradient background from blue to red (right).

Scenario 11 Define a grid that serves as a page layout, and give it three columns and one row with equally distributed length. Make sure to wrap them under each other once the screen is narrower than 1000px, such that there are one column and three rows.

4.4 User Testing

The Code

Feedback

4.5 Revised API

5 Resulting Framework

Framvisning av det ferdie API-et

6 Discussion

7 References