API Specification

Silhouette

Group 12:
Mats Engelien
Lars Erik Faber
Håkon Marthinsen

Contents

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

1 Introduction

Silhouette is a web framework that let's users create websites with Java. But why Java? Although websites can be made using raw HTML and CSS, this approach comes with a few caviats that can lead to issues during development. In-short our solution aims to avoid these issues alltogether, making the development process overall more enjoyable and creative, but let's go a little more into detail.

By design, HTML is a fixed markup language where the structure of tags determine not only the presentation but also the layout in the document object model (DOM). Content is grouped together in containers and follow a tree-like structure, which is nice for the end-user, but it can restrict the programmer. The problem here is that the programmer must follow the strict formatting guidelines of HTML, leaving no space for personalization. Another problem is that browsers will try to interpret the markup regardless of its validity, meaning that typos and other seemingly minor issues can easily go unnoticed.

An article titled "The Problem with HTML" by Dr. Maxime Chevalier-Boisvert argues how HTML, CSS and JavaScript is growing fast to become more than originally intended, blurring the lines between the three[1]. For instance, it is possible to do animations in both JavaScript and CSS. The problem with blurred distinctions is that web development becomes less organized. A second argument can be made that due to the fast growth of these languages, browsers fail to support the newest and greatest features. This forces developers to take precautions when designing web pages.

With the Silhouette framework, it keeps function areas more organized by clearly distinguishing between markup and styling through different APIs. It tries to promote best practices, such as not using in-line styling, by making these processes easier, and hiding away the bad practices. In addition, the methods will make sure to correct minor mistakes to prevent nasty surprises. Silhouette also implements solutions for tags that are less supported among browsers.

1.1 Group Description

The project group consist of Mats Engelien, Håkon Marthinsen and Lars Erik Faber. Throughout the development, everyone has contributed in both the design and programming aspects of the project, as well as writing in the final report.

Mats has had a big responsibility regarding the APIs and code implementation in the framework. He is a skilled programmer in multiple coding languages and is great at problem solving. In addition, he is good at discussing problems in the code together with the group.

Håkon is good at managing social connections and is also a creative brainstormer. Therefore, he has played an important role in communcating with testers in order to arrange user tests throghout development.

Lars Erik has done a good job creating of one of the APIs of the framework, and has laid the foundation in the final report. He is good at arranging meetings with the group and has written summaries for each guidance meeting.

1.2 Delegation of Work

Mats Engelien - Java (HTMl) and Documentation
Lars Erik Faber - Java (CSS) and Documentation
Håkon Marthinsen - Documentation and User testing

2 Background

There is no hiding that there are plenty of existing web frameworks out on the internet already, each designed to cover certain needs. With regards to this, Silhouette also does not try to reinvent the wheel, but it uses a lot of the same tools found in these well established frameworks, along with some quality of life improvements.

People generally use web frameworks to gain access to additional features and to reuse existing code, as opposed to redoing it from scratch. It is important that the web framework increases the velocity of web development compared to doing it in plain HTML and CSS. A good example is the *JSF framework* (Java Server Faces). It has a unique feature where users design UI-components through high level abstractions that are converted into the appropriate representation of HTML depending on the device and version of the internet. This greatly reduces repeated code and means that users can worry less about the technical aspects.

Another important side of Frameworks is to establish a structure in the application. The *Spring framework* creates such a structure by abstracting POJOs (Plain Old Java Objects) into special components called beans. These components are designed to be flexible and adjustable and can be wired together with other beans. This way, users can manipulate each component by interacting with the beans.

Vaadin is probably the framework closest to Silhouette. It enables users to make web applications completely in Java, which means it does not require any prior knowledge to HTML, CSS or JavaScript altogether. Everything made with Vaadin is a component that are presented in layouts on the page. To add functionality, the user adds event listeners to the desired components. And lastly, to stylize the web app, Vaadin uses a theme roller called *Vaalo* that is similar to CSS but comes with some additional features, like auto-contrasting text color depending on the background color. It does the majority of what Silhouette wants to achieve, which is why it is the closest to Silhouette.

An example showing how to make a text field in Vaadin

```
        // Simple textField for input
        TextField textField = new TextField("Search");
        textField.addThemeName("rounded");
```

How to create a simple header in Vaadin

```
        final HorizontalLayout top = new HorizontalLayout();
        top.setDefaultVerticalComponentAlignment(Alignment.CENTER);
        top.setClassName("header");

        final String resolvedImage = VaadinService.getCurrent()
                .resolveResource("images/header_image.png");

        final Image image = new Image(resolvedImage, "");
        top.add(image);
        top.add(title);
```

3 Method

3.1 API Design Specification

Throughout this process we have employed Scenario-driven design. This is an iterative process that changes each part of the resulting API along the way. This process works backwards from what the dream scenario would be, then loops along the way to change the resulting API to fit the client's needs.

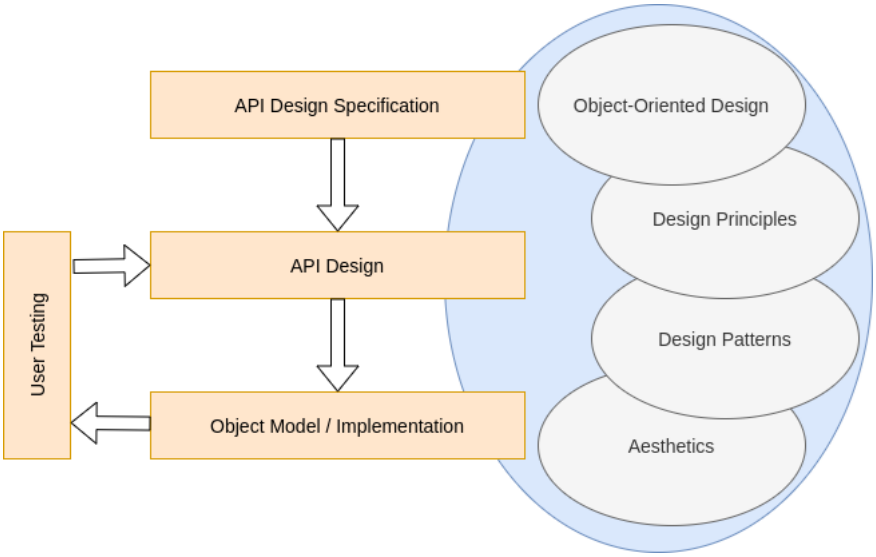

Figure 1: Api design process

The API design specification consists a collection of the client-code and scenarios that guide the design of the interface. A scenario is a problem that needs to be solved by the API and the client-code is dream/pseudo code that should solve the problem presented.

3.2 API Design

The design of the interface is the interface's structure. What areas of the interface can communicate, the file-architecture, which methods are needed and what each class should do.

3.3 Implementation

Implementation is the actual code that is written. The details of the implementation should not leak out into the API. Design patterns and principles, and user-testing inform how the physical implementation develops.

3.4 User Testing

User tests are performed to gain a better understanding of how the user interacts with the APIs.Generally we would give the testers the updated Jar file containing the framework, a type reference documentation, and a set of scenarios to solve. These scenarios varied a lot from test to test. Since we emphasize scenario driven design, these problems are formed as scenarios that encapsulate a certain interaction with the API.

During user testing, there is an ongoing dialogue with the testers so we can guide them should they get stuck, or to answer general questions. At the end of the session, we run a quick discussion around their experience of the framework. This is where they express their thoughts and give feedback.

The tests were done with groups 18, 40, and 5, as well as 5 other indivduals. In total, we ran 8 different user tests.

4 Design Process/Results

4.1 API Design Specification

Process

All relevant scenarios produced are attached as an appendix. To develop the APIs, we employed Scenario-driven design. As a group, we discussed what we wanted these APIs to do. The requirements that developed from this process were: 1: To object-orient writing HTML and CSS. 2: Have an almost one-to-one translation from HTML-tag and CSS-ruleset to java object. 3: Write to file and give that file a .html or .css extension. Based on these three main concepts for the API we created our initial scenarios for how this framework would be used. The design and implementations of the APIs are slightly different as they were designed and implemented at different periods of the development process.

Scenario 24 - Make a website with a Form consisting of a field for e-mail, phone number, a text field and a submit button.

This scenario hits on a few different aspects of the API. It requires a form class that has 3 different input-fields, and a button connected to it. We also need to put that form in a logical place in an html-file. This html-file also needs to contain a head with meta-data and necessary tags like a body, a header and a footer to be a webpage.

```
WebPage myPage = new WebPage(); // initialize webpage

ArrayList<InputField> listOfFields = new ArrayList();
InputField emailField = new InputField(type: "email", name:
    "email");
InputField phoneField = new InputField(type: "phone", name:
    "phone");
TextField textField = new TextField(type: "text", name: "text");

listOfFields.add(emailField);
listOfFields.add(phoneField);
listOfFields.add(textField);
Form myForm = new Form(listOfFields);

// generates html page in this order
myPage.generateHeader(); // generate simple header
myPage.append(myForm);
myPage.generateFooter(); // generate simple footer
```

Here we had usage of both high-level and low-level APIs. We could create simple one-to-one elements and we could automagically generate entire parts of a page.

Scenario 25 - Make an article page with a title, an image and 5 paragraphs of text Here the user needs to be able to create the body, main, and other semantic containers for their website, but these tags do the same things. In addition, we need to be able to populate the containers with other html-tags.

```
CreatePage thisPage = new CreatePage();
int numbersOfParagraphs;
String[numbersOfParagraphs] paragraphsText;
Title titleName = new Title();
Article articlePage = new Article();

thisPage.add(titleName);
thisPage.add(articalPage);
GetImg img = new GetImg(String imgName);
Paragraphs newParagraph = new Paragraphs()
articlePage.add(img);
articlePage.add(newParagraph(numbersOfParagraphs, paragraphsText));
```

These scenarios created many questions about how the design should be. We started doing one-to-

one translation between HTML-tags and java-objects, but now there were these containers that did not need to be their own classes.

4.2 API Design

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.

We designed the interface as a group, discussing how the classes would reflect the client-code. We separated the higher-level and lower-level aspects into their own packages, and decided that all html-elements would need to inherit from one superclass. The superclass then had all the base necessities of said element in a similar manner to the Factory method. However, there were many problems with this. Too many objects were being instantiated, no method-chaining, it acted clunky, and looked clunky, on top of being unreadable.

This is where we were introduced to the Builder-pattern.

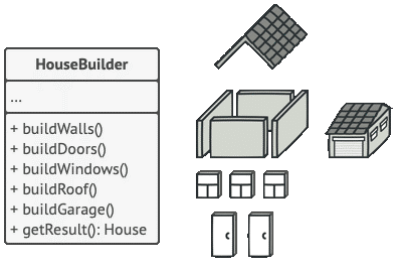

Figure 2: Builder Pattern Visualized

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.

However, with Silhouette, we took a different approach using the builder pattern to account for all the different options when writing HTML and CSS. The pattern creates a structure in the code that consists of building components and appending them to each other. With Spring, the user builds a component through the factory pattern that initalizes the beans by following given instructions, but we thought the builder pattern would be more flexible on the go:

Creating a paragraph and adding it to a div

```
Paragraph p = new Paragraph.Builder()
        .setText("Hello World")
        .build();
Container div = new Container.Builder()
        .addElements(p)
        .build();
```

The builder patterns organises the process of creating objects into steps, so one can make the same object look very different depending on how it was put together. Using a director on the builders allows for even more abstraction that we could use to develop a base for a high-level API. This design pattern allows for the making of immutable objects and method chaining. This made the code a lot more readable, though still somewhat repetitive.

Creating a paragraph and adding it to a div

```
                new HTML.Builder()
    .addElements(new Form.Builder(new FieldSet.Builder()
                        .setInputs(
                          new Input.Builder()
                                .setLabel("email")
                                .setId("email")
                                .setName("email").build(),
                                new Input.Builder()
                                .setLabel("phone")
                                .setType("tel")


                                   .setId("phone")
                                .setName("phone").build()
                                new Input.Builder()
                                .setLabel("name")
                                .setId("fullname")
                                .setName("name").build(),
                                new Input.Builder()
                                .setType("submit")
                                .setValue("Submit").build())
                        .build())
        .setAction("/dest.php")
        .setMethod("get")
        .build())
    .build()
    .initialize();
```

Now we needed to separate functionality. Which classes belonged, and what needed to be extracted. Following the Single Responsibility Principle, each class should have one and only one responsibility. In most cases we have tried to follow this as best we could. The one-to-one mapping that we initially built the API around was due to this principle, though deprecated or exceedingly similar HTML-elements were left out of the design and implementation, some due to time constraints, and some because they were better to combine, such as all page section-tags(¡div¿, ¡main¿, etc.). The HTML-class currently has too many responsibilities and breaks with this principle, due to the initialize()-method, which means the html-class currently both represents an html-tag and the an html file. The compiling and writing to file should happen in its own class like it does in the Framework's CSS API.

Abstraction to interfaces within the API allows for different implementations of the same method in multiple classes. Sticking with the interface segregation principle, we tried to ensure that these interfaces would only have a few specific tasks.

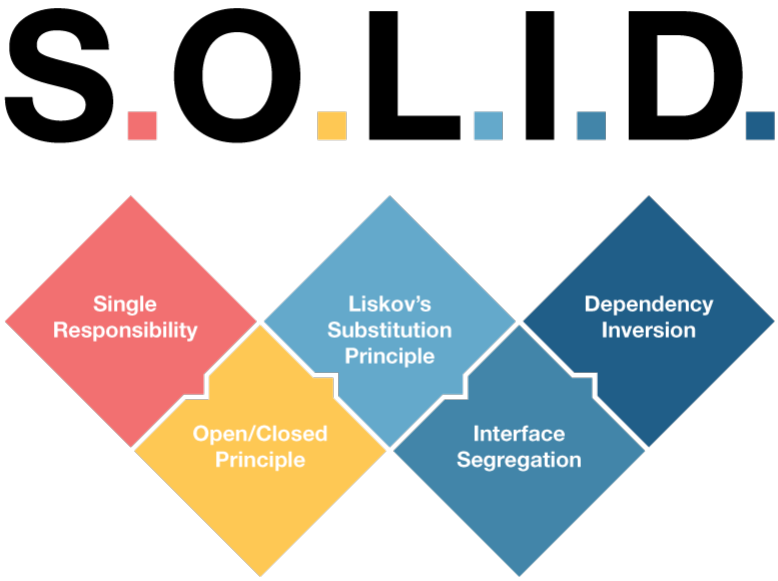

Figure 3: SOLID Principles of Software Design

Most non-abstract or static classes received their own internal static Builder class during the initial implementation, however some classes did not, like the Paragraph and Heading classes as they could be considered very small, and are used with high frequency. Though it unnecessary to implement the pattern here as well, due to user-test feedback, we decided to after all, since the change in pattern broke with the development flow. This led to the implementation of builder-classes for every class in the API that should be instantiated.

Personal Decisions

4.3 Implementation

Most non-abstract or static classes received their own internal static Builder class during the initial implementation, however some classes did not, like the Paragraph and Heading classes as they could be considered very small, and are used with high frequency. Though it unnecessary to implement the pattern here as well, due to user-test feedback, we decided to after all, since the change in pattern broke with the development flow. This led to the implementation of builder-classes for every class in the API that should be instantiated.

The top-level builder has been abstracted into its own interface. This interface is then implemented in each of the internal builder-classes. Even though a lot of very different looking objects are being created, both the higher- and lower-level components of the API should depend on the same abstraction, so there is some achieved dependency inversion, though implementation is not complete.

The biggest problem with the implementation of the Builder-pattern, and attempting to apply these principles, is that the builder pattern usually doubles the amount of code necessary, and using correct abstractions can be very difficult.

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 use 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.

Lastly, we needed to form a system to generate the actual file. 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.

4.4 User Testing

(THIS NEEDS TO BE CHANGED)

User testing started after a month of developing the first iteration of the API. At this point the low-level HTML API felt almost complete, however the high-level-, and CSS-APIs were bare minimum. The implementation was not ready and the APIs consisted of plain old java objects. It was therefore important know if the names and overall use made sense, and if there were any major flaws that we could improve upon.

After the testers solved the scenarios using our framework, we would have a quick discussion with them about their thoughts and opinions. As they expressed their views on the framework, we would take notes and often ask some follow-up questions. At the end of the session we would ask some general questions to see how their views compared with other testers. This would help us see how the changes affected the framework, as other testers had different opinions.

After the session with the testers, we would have a discussion with eachother about the feedback we got from the user testing. Since each group member had their own focus areas in the development, this would help a lot to get insight on the next big changes we were going to implement.

right after all the classes that was needed to build am HTML document 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.

Frist user testing session (Silhouette-HTML v.2)

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 (Silhouette-HTML v.2)

This user testing toke more time that what was needed, if it wasn't for an error from our side in the code structure. When we found out what the error was and the confusion was over, it was back to the user testing, but the person that was testing the program was stuck and didn't understand how to start and/or what to write (defining the html-object). After some time we gave the user a push in the right and the user started to type/write like it was natural for them or as the hade used our framework before.

Third user testing session (Silhouette-HTML v.2)

In this user testing the tester was tested in a different way, where the tester received minimal help and mostly through think and some fiddling on their own, because of the users higher understating to code structure and language.

The user managed the first part of the task easily (due to the naming of classes and method names) but began to strive on the second task. The user was supposed to create a new object called ”Container”, but started working on the method calls instead (because it seemed most natural to them) and had to ask for help and then came the next problem... what are Builders.

After some explanation it came naturally and the rest of the second task was then over and the user was on their owe, with a little hint here and there. The user could see the logic but felt that it lacked a flow in the structure and believed that getting in to ”Silhouette” for a new user is not easy without a manual or help from someone.

Fourth user testing session (Silhouette-HTML v.3)

Fourth testing was done in a group context, where one of them wrote and the others watched and tried to understand ”Silhouette” (I also had to describe how Builders works).

It started the same with this group as it was done with the other user testing section, but since one of the user was looking through the library and was reading the methods that had a similar name as in HTML, one tester got an idea what to write and they managed to complete this task. And they applied the same tactic to the next tasks, and they managed the whole user testing with a few hints.

Fifth user testing session (Silhouette-HTML v.3)

The fifth test was carried out by a student with a good knowledge of java and HTML, who wanted to learn the framework (“Silhouette”) from top to bottom. The user spent 1 hour and 30 min doing the task (without help) and gained insight into how ”Silhouette” HTML.BaseComponents.* (file's) worked. This user was more interested in the ”Silhouette” library and gave some feedback about that the structure is easy to understand, but there was a lack of comment (you can read comments for a jar file) on what the methods do/did, etc. and that the end result might affect a user “Silhouette's code” and meant that “it's too hard-coded” when a user is done with it.

Last user test

For the last user test we tested with a student who previously had the same framework course, and who was very familiar with builder pattern and API design. The session took about three hours, and we tested both the HTML API and the CSS API. We sent him the Jar file of the newest version of the framework, a type reference documentation, and some scenarios to solve with the framework.

Make a new HTML page. Set a custom title, and set the charset meta tag to ”UTF-8”

Scenario 17

For scenario 2, the user were instructed to make a body element and add it to the page. At first, it wasn't very obvious form him to use the Container object to build the body element, as it is passed

through the parameter. After some explanation, he understood why we had done it that way, but still felt like it was not as intuitive. He commented that it would be much better to just separate the most important layout elements into their own class, though this was already discussed within the group at the beginning of the project. Having a class for each element comes with two major issues. Firstly, if we would simply make a class for each element, there would be too many classes to keep track of. Secondly, how would we decide which element needed their own class? Because of this we decided to keep it as originally planned.

Scenario 18

Here he found it confusing that there were two ways of adding a heading to a container element. While the intended solution was to make a Heading object, he opted for the addHeading method inside ContainerElement.

Making a Heading object

```
Heading myHeading = new Heading.Builder()
        .setLevel("h1")
        .setText("My heading")
        .build();
```

Using the addHeading method inside ContainerElement

```
Container div = new Container.Builder()
        .addHeading(1, "My heading")
        .build();
```

Initially, this was done to give users more options when declaring the markup. However, this proved to be more confusing then helpful, so we decided to remove this additional method.

Scenario 19

Right off the bat, his impression was that we had over-used the builder pattern, to the point where it becomes unecessarily complicated. For example, he explained how it was unecessary to use builder pattern for the Paragraph class, as the paragraph should only take one argument in the constructor anyway. He suggested that it would be better to simply pass the text to the constructor.

With builder pattern

```
Paragraph p = new Paragraph.Builder()
        .setText("Hello, World!")
        .build();
```

His suggestion

```
Paragraph p = new Paragraph("Hello, World!");
```

His suggestion was not bad, but since we wanted to follow the same convention for every class, we felt it was better to leave it as be with the builder pattern. That way, there is not confusion whether it uses the builder or not. We also find the parameters more clearer with builder pattern, as it explicitly shows it in the method name.

Scenario 20

At first he was not fond of using enums when adding a new rule to the rule set, but quickly he grew the habit of using them. He commented that it felt unecessary at first, and that he would rather type in the propeties and values as a string. But after explaining why we use them (which is to make sure the user does not pass inn misspellings), he understood and agreed that it was a good solution.

Scenario 21

To finish off the session, we asked him some general questions about the framwork. Firslty, we asked him about what he thought about the name ”Silhouette”, and if he found it intruiging and informative. To this, he told us that the name is as ambiguous as most other names out there. He stressed, that it does not necessarily mean it is bad, but it did not give him any idea of what the framework did. At this point, we have become very fond of the name, but it is always nice to get feedback.

We also asked him if he found the framework to be difficult to use. To this he responded that builder pattern greatly increases the amount of code needed to generate a website, so in this case it would be faster to write it in standard HTML and CSS. We agree to this, wowever, we are still of the belief that the OOP aspects and the additional make up for it. Another issue for him was the quality of the type reference documentation, as it often failed to give him a good idea of what each class represented.

5 Resulting Framework

6 Discussion

The plan is to make a html-structure in Java by using object-oriented code structure.

In the first phase we were working together all the time. We collected as much information that we could find, about html and css. And after some working sessions we had a 1 to 1 structure (of the Java-code to html and css part). The css part was straightforward when it came to the structure part, but the problem was in the html. We found new and better ways all time to make the client code easier and more understandable.

In the second phase we have agreed that this is what we need for “Silhouette” to be able to create web pages. At this point we are working on the html part and hope that we can start on user testing aright away. After some user testing, we start on the third phase and now that we knew what we want the code structure to look like, we start to specializing us (and you can se that in the “Delegation of work”).

7 References

[1]: Chevalier-Boisvert, M. (2015). *The Problem with HTML.*
Retrieved from https://pointersgonewild.com/2015/04/16/the-problem-with-html/