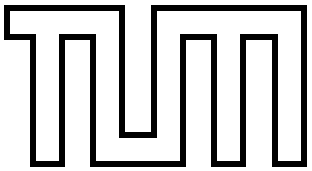


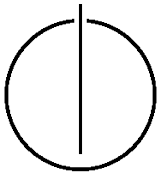
Department of Informatics

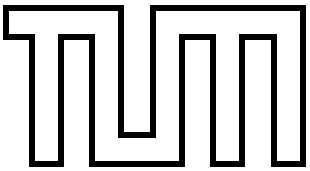
Technical University of Munich

Master's Thesis in Information Systems

Platform-Independent UI Models: Extraction from UI Prototypes and rendering as W3C Web Components

Marvin Aulenbacher





Department of Informatics

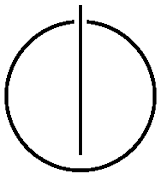
Technical University of Munich

Master's Thesis in Information Systems

Platform-Independent UI Models: Extraction from UI Prototypes and rendering as W3C Web Components

Plattformunabhängige UI-Modelle: Extraktion aus UI-Prototypen und Darstellung als W3C-Komponenten

Author:	Marvin Aulenbacher
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	M. Sc. Adrian Hernandez-Mendez
Date:	October 15, 2017



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 12. Oktober 2017 Marvin Aulenbacher

Acknowledgments

Firstly I would like thank Prof. Dr. Florian Matthes for giving me the opportunity to write the thesis at the Software Engineering for Business Information Systems (sebis) chair. I am deeply grateful to M. Sc. Adrian Hernandez-Mendez for his advice, valuable input and guidance throughout the development of this thesis.
In addition, I want to thank my parents Axel and Andrea for their endless support and motivation during my academic studies. Lastly, I want to thank my sister Meike for her support, expertise and proof-reading.

Abstract

Managing a process to design a user interface faces many issues. One is the communication between the designer and developer. This issues may even be greater in the domain of prototype driven development, since it may be iterative and refactoring starts the process from the beginning. These issues are significantly slowing down the development process and therefore the life and release cycles of the product.
The idea of this thesis is to provide a tool, that automates the process of transforming a drawing of the user interface by an UI expert to W3C Web Components. The data is exported from the UI expert's prototyping tool via the scalable vector graphics format and then transformed to a view model, from which W3C standardized web components can be generated.

Contents

1 Introduction

In the modern era of software development most notably agile software development, the goal is to shorten the release and development life cycles of software. Start-ups and small companies are continuously threatening established players with new concepts, technologies and innovative products. In recent years there has been an observable shift to heavy user oriented programming, specifically through web applications and mobile devices. This shift causes a high demand for front end or user interface designers. Many processes have been created to support the design phase in software development.
But specifically when it comes to heavy user oriented products, it is tough to go with the flow of short release and development life cycles. The reason is not difficult to identify. The process of designing a user interface is between the designer, the future user and the responsible front-end developer. As there are many people involved, it will be harder to speed up this process because of communication issues, cultural aspects etc. The designer has to develop a draft, which the front end developer implements and after that the user has to review the user interface to check if the requirements are met. If a requirement needs refactoring the whole process has to start from the beginning.

1.1 Problem Description

In the process of developing user interfaces with prototype driven development the designer and developer face certain communication issues, working on low- and high-fidelity prototypes. The goal of this passage is to identify several communication issues between those two employees and derive the problem description on this basis.
Communication issues between the developer and designer come up on different phases of the development of a prototype and are triggered by different causes. There are organizational structures, that do not contribute to a improve communication[Isa16], personal and cultural aspects[Joh12] as well as differences in skill sets and working experience. The problem they face is always similar as they do not share a common domain specific language. In this particular case the designer is familiar using prototyping tools whereas the developer is familiar using editors or IDEs. One idea to support this process is to assist the developer by creating an abstraction of the UI prototype which the developer can be expected to understand. One possibility of a platform independent abstraction of the view that matches the stated requirements would be the view model. This model could further be used to generate front-end source code and hence not only contribute to knowledge about the structure and composition of components of the abstracted view. Through this generation the time the developer needs to craft the high-fidelity prototype from the low-fidelity prototype, which was created by the designer, would significantly decrease. This decrease of needed time is even more valuable when the prototype development approach is as usually incremental and triggers refactoring actions after being presented to the end

user. Instead of starting from the scratch the developer could use the extracted and generated elements and would be able to speed up both the crafting and refactoring process.

1.2 Research Questions

The goal of this thesis is to provide a tool to bridge the gap between the developer and designer by generating a view model and W3C Components of the low-fidelity UI prototype for the developer. This could greatly impact the time and effort to finalize design and implementation details for the product.

1. Which design tools do designers use and why? (Two Tools)
2. How should the architecture of a supporting process for the generation of web components be structured?
3. How useful is the automation process based on two basic use cases for prototype driven development?

1.3 Research Method

The research for the extraction tool and the generation process is conducted on the design science research methodology described by Alan et al[HC10]. The framework is structured in three pillars *Environment*, *IS Research* and *Knowledge Base* to supply organizations and their research with information in a way that it can be utilized. The three pillars clearly decouple inputs and interactions and hence improve the insights of the research. The pillars and their respective inputs and interactions are shown in figure 1.1. Information systems are in general implemented within an organization to improve the effectiveness and efficiency of the organization as well as their research by aiding the managerial process with knowledge of the information technology and its use[HC10, p.2].
The environment pillar consists of people, organizations and technologies and formulate comprehensive business needs to the *IS Research*. This implies that research is always triggered by business needs.
The research process itself is separated into two specific processes, namely behavioral and design science. Behavioral science focuses on developing theories to predict or explain organizational and human phenomena surrounding the analysis and in the end justifying their results[HC10, p.10]. The design science paradigm has its roots in engineering and is fundamentally a problem-solving paradigm. It seeks to create innovations that define the ideas, technical capabilities and practices trough which the analysis, design and implementation can be effectively and efficiently accomplished[HC10, p.3]. In the end a tool or artifact can be crafted on this theoretical basics[HC10, pp.11].
The knowledge base provides fundamentals for the research such as frameworks, instantiations and methods that already have been used by researches, that were conducted earlier[HC10, p.7]. The new research therefore profits by prior research through the respective findings such as boundaries and experiences.
The acquired results by the conducted research loop back to the environment and to the knowledge base. In case of the environment the findings are applied to the appropriate

environment to satisfy business needs for instance. On the other hand the findings are also used to enrich the knowledge base and hence can be used for further research.

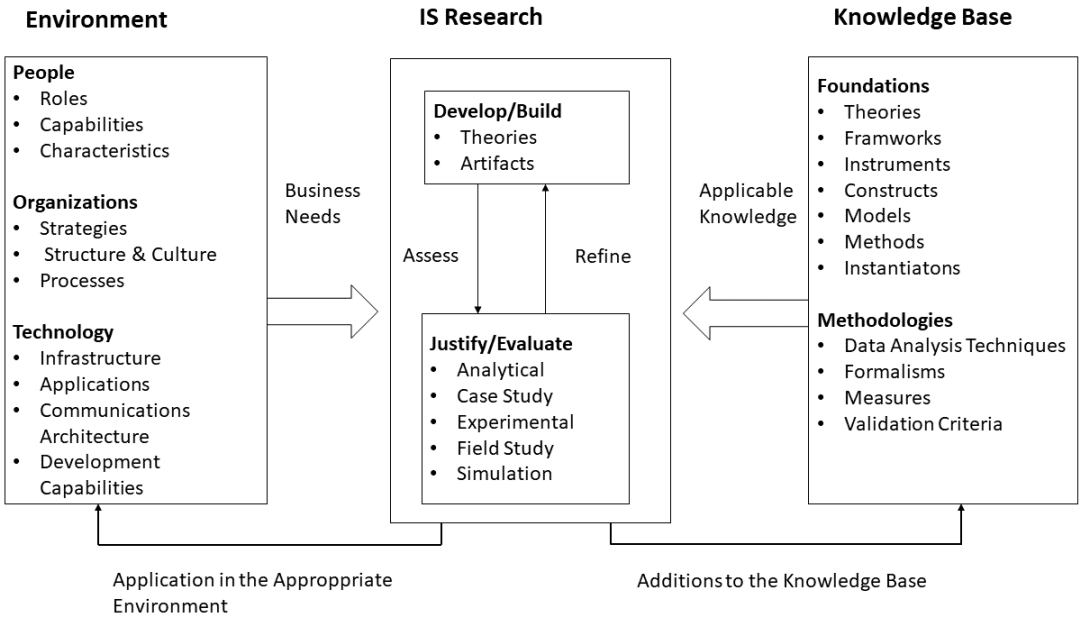


Figure 1.1: Research Framework Based on Design-Science Paradigm[HC10]

2 Domain Description

This chapter is contemplated as an entry point to the thesis and hence to provide understanding for the thesis' domain. It will describe each aspect of the domain in their respective section and will lead to a greater understanding for the abbreviated problem description and the idea that will be presented in the conceptual design chapter.

2.1 Web Application Development

In this section a short overview of Web Application Development will be given, as well as a deeper insight in the problems that small companies and start-ups have to face, when in the search of design know-how.

2.1.1 Overview

As mentioned before, the world of software development is changing rapidly and especially Web Application Development . There has been a rapid shift from traditional development approaches, which are defined by defining requirements, planning, building, testing and deployment to agile development approaches [MW17, p. 584]. Those agile approaches involve smaller and shorter releases, parallel programming, and iterations as core pf their process [MW17, p. 585]. The link between agile development processes and User-centered Design has been increasingly gaining interest [MW17, p. 584]. The described link on the other hand has lead to a strong demand for design know-how, as most companies are developing applications with emphasis on user-centered design.

2.1.2 Lack of Design Know-How

In recent years, big companies in the software development sector have acquired design companies to include know-how in their business landscape. There are many cases as for example [1] Google and Facebook acquired design companies, Accenture purchased Fjord, Adobe bought Behance, which is basically a creative online community [2], and Square bought 80/20.
The question that arises is what solutions can small and medium sized companies as well as start-ups develop, to keep on track with the big players.
Small companies, as they can not just acquire know-how have to develop processes to optimize collaboration with those design companies. One of this processes is part of the domain of this thesis and will be introduced in the next section is prototype driven development.

[1] https://www.wired.com/2013/05/accenture-fjord/
[2] https://www.behance.net/

2.2 Prototype Driven Development

2.2.1 Overview

Prototype driven development is a method to iteratively design a product. It is often used in the context of Web Application Development, because of the short release and life cycles. In this section, a typical process for prototype driven development is presented and explained to narrow the problem space of this thesis.

2.2.2 Typical Process

In Figure 2.1 the process of a prototype driven development is modeled. The process typically starts with a meeting between the application's designer and its customer. They formalize technical and visual requirements. In the second step the designer receives these requirements as an input and designs a low-fidelity prototype. Low-fidelity means, that the prototype is designed quickly and is an easy tangible representation of the concept [mob16b]. Low-fidelity prototypes can be paper-based or look-and-feel mock-ups. They allow ideas to be evaluated for numerous aspects, like usability with real users. This may lead to refining and testing the low-fidelity prototype again, before design decisions are finalized and lead to the high-fidelity prototype [Jac09, p.232]. Usually the designer uses prototyping tools like Figma or Sketch. The low-fidelity prototype is an image file, that represents the visual requirements. This image file on the other hand serves as input for the developer, who implements a high-fidelity prototype. High-fidelity prototypes are computer-based interactive representations of the product or application in its closest resemblance to the final design in terms of details and functionality [mob16a]. The main disadvantage of high-fidelity prototypes is that its iterations are more time consuming and thus prevent exploration of new ideas without jeopardizing the entire project [Jac09, p.683]. This prototype is ready for evaluation by the end user, who may have some improvements or change requests. In this case the process starts from the beginning.

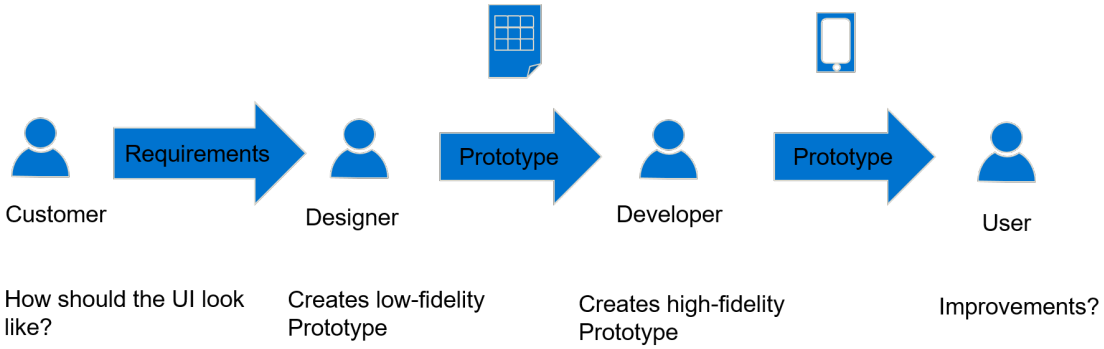


Figure 2.1: Prototype Driven Development

The key issue addressed by this thesis is the communication between the designer and developer. As they do not share a common domain specific language, they are not able to communicate in a proper way. For example, the developer is familiar with his IDE or

editor, as on the other hand the designer is used to prototype with his UI Prototyping tools. The idea of the thesis is to provide a tool, that helps the developer to create high-fidelity prototypes more quickly and be more efficient in case of re-factoring.

2.3 Prototyping Tools

This section is contemplated to give a quick overview of prototyping tools as well as the two tools, that were used for the implementation.

2.3.1 Overview

With the proliferation of mobile, there needed to be enhancements to common used paper prototyping. Paper prototypes are inconsistent, designed for a very specific use case and though not reusable. Even User-centered Design(UCD) explicitly recommends a specific life cycle stage, where the UI could be prototyped based on the future system's stakeholders like designers, developers, usability specialists, graphic experts, and end users [CKV07, p.1]. To fulfill designer's needs prototyping tools have been developed, that support the designer with vector graphics, predefined templates for environments and even real-time collaboration. Prototyping tools, as they are supporting a process, need to fulfill three requirements:

1. Design specifically for the context in which work will be viewed.
2. Communicate the end user's experience by showing how a screen changes based on input.
3. Add transitions and moments of joy to their design through motion graphics [Dyl17].

Of course, those three requirements do not make a complete list, but they still illustrate the most important ones from the perspective of a designer.

In the following two common used prototyping tools, namely Sketch and Figma, which have been used to analyze the structure of user interfaces will be presented in greater detail. This analysis finally leads to the implementation of the extraction tool that will be presented in the implementation chapter 4.

2.3.2 Figma

Figma is a powerful editing and prototyping tool. It has been build to create digital products and user interfaces. A neat feature is the component based architecture. This enables the possibility of reusing elements and not having to start from the scratch every time[3].

There are more features that make Figma stand out.
Firstly, the Vector Networks technology used, is a significant improvement on the traditional path model. A vector network improves on the path model by allowing lines and curves between any two points instead of requiring that they all join up to form a single

[3]https://www.figma.com/features

chain. This helps provide the best of both worlds. Splitting, deleting and recombining is much more natural with vector networks[Dyl17].
It also offers features to work collaboratively as a team. Furthermore it is shipped with version control, which is especially an advantage when working in rounds of feedback and multiple iterations, just as explained in the process of prototype driven development. The arguably biggest advantage compared to other tools, may be Figma's collaborative workspace. It offers real time commenting and assets, that are shared team wide[4].
In conclusion, Figma is a great prototyping tool which is pretty new to the design cosmos. It offers many new features and may get more attention with increasing time.

2.3.3 Sketch

Sketch is also a vector design tool, that is entirely focused on user interface design and only available on iOS. It is advertised as an alternative to Adobe's Photoshop, which used to be the tool for designing user interfaces or even static prototypes, although it was originally developed for image processing. Sketch offers styles, that are only relevant to UI design, as well as an iPhone previewing tool called Mirror and Artboards. A common core feature is the availability of UI templates for iOS platforms, Android Icon Design and even Material UI[5].
There are even more features:

- Autosave and versioning
- Vector editing and pixel perfection
- Edit elements on the fly
- Convert styles to CSS [Jea13]

A survey conducted by Substraction.com in 2015 asked about 4000 participants from almost 200 countries to report on their favorite tool for six major design tasks like brainstorming, wireframing, interface design, prototyping, project management and version control. Sketch was the favorite tool for wireframing and interface design, although falling short on prototyping[6]. Although the survey is not representative on how it was conducted, it clearly shows the popularity of Sketch among the participants.

2.4 W3C Components

This section is contemplated to serve as an introduction to the W3C Web Components specification, which is drafted by the W3C consortium. The goal of the consortium is to include those standards in every modern web browser.

[4]https://blog.framer.com/a-simple-way-to-import-and-iterate-with-figma-8cf0cd4d21b3
[5]https://designcode.io/sketch
[6]http://tools.subtraction.com/

2.4.1 Overview

The W3C Web Component standard consists of several separate technologies, that will be introduced in the design subsection. Web Components are reusable user interface widgets, which are part of the browser or are planned to be included in the standard. Hence most of the Web Components do not need any external or transformation libraries to be executed by modern web browser[7]. Even if certain features may not be supported by the browser yet or are still in development, there are so called Polyfills which make sure the browser can use the technology by using already available libraries[8]. As an example Custom Elements, which are part of the W3C specification, are already available in Chrome and in development for Firefox[9].The Web Components can be created by using open Web technology and can be seen as an improvement to the HTML, CSS and JavaScript combination. There are many famous front-end frameworks, that are based on the Web Components standard, like Google's Polymer 2.0[10] or Vue.js [11].

2.4.2 Design

The W3C Web Components consist of the following four technologies, which can be used separately:

- HTML Imports
- Custom Elements
- Shadow DOM
- HTML Templates

Custom Elements are arguably the most exciting part of the specification. A developer can create new HTML tags, extend existing HTML tags or components, which other developers have created. Those possibilities enable a web standards-based way to create components that are highly reusable and consist of nothing more than vanilla JS, HTML, and CSS. The result is less and more modular code. [Eri17a].
Custom elements also have so called life cycle callbacks, which are invoked after a specific event has happened. They are implemented to help the developer hook in between the component's life cycle.

[7] https://developer.mozilla.org/en-US/docs/Web/Web_Components
[8] https://www.polymer-project.org/2.0/docs/polyfills
[9] http://caniuse.com/#search=custom%20elements
[10] https://www.polymer-project.org/
[11] https://vuejs.org/v2/guide/components.html

The three life cycle events are:

1. readyCallback
2. insertedCallback
3. removedCallback

As the function names already indicate, readyCallback is called after a custom element has been created, insertedCallback is called after the custom element has been inserted into a document and removedCallback is called after the custom element has been removed from the document[12].
The Shadow DOM on the other hand is responsible for encapsulation and is therefore the core of the Web Components. Shadow DOM is able to fix CSS and DOM issues by introducing scoped styles. Those scoped styles simplify the use of CSS, because one does not have to worry about naming conflicts and is able to use simple CSS selector names. Shadow DOM also hides implementation details and can wrap Custom Elements[Eri17b].
HTML Templates as the name already states, is thought to be the starting point of a particular application so that the format does not have to be recreated each time it is used[13]. Using HTML Templates offers certain advantages like that its content is effectively inert until activated, because HTML Templates are by default deactivated. This also means that there are side effects like scripts running or selecting elements with

```
document.getElementById()
```

until the point of activation[Eri13b].
The last part of the specification are HTML Imports. Every type of resources used on the web have their respective syntax to be loaded. Comparing it to JavaScript's script loader

```
<script src>
```

or CSS'

```
<link rel="stylesheet">
```

stylesheet loader[Eri13a], there is no well defined way of importing HTML. HTML Imports close this gap for HTML content. It is a way of including HTML documents into other HTML documents.

2.5 Related Approaches

The goal of this subsection is to reference related scientific approaches and to elaborate the theoretical foundation of the implemented tool on the basis of the research method described in the introduction.
A lot of literature in this specific domain is about reverse engineering legacy software or software architectures. It means to transform a system or system architecture back to some kind of model that displays all relevant parts and relationships. In the scope of this thesis

[12] https://www.w3.org/TR/2013/WD-components-intro-20130606/#lifecycle-callbacks
[13] http://www.thefreedictionary.com/template

the focus can be narrowed down to only the graphical user interface part of systems. One example for this kind of approach is the automatic extraction of graphical user interface models for industrial testing by Aho et al. The team developed so called Murphy Tools that in combination with their innovative platform independent technique is able to extract GUI models based on a dynamical analysis of the GUI[Aho+14, p.1]. This model is than later used to support industrial testing of the software that the model was extracted from. Aho et al divide their process into three steps:

1. Application specific model extraction script.
2. Murphy tool explores GUI application and dynamically extracts model.
3. Visually inspect the model and validate the correctness of observed behavior.[Aho+14, pp.3]

Some experience the team acquired are very interesting foundations for the further work of this thesis. For instance they recommend to partition the extracted model into a set of smaller and simpler models to reduce complexity and increase the usability of the models in the daily work of developers[Aho+14, p.5].
Another interesting approach and theoretical foundation of this thesis is the basic architecture by Chikofsky and Cross for a platform independent reverse engineering process. The architecture splits the transformation in three distinctive steps. It starts with the semantic analyzer continues with a enriched information base and ends with the view composer[CC90]. The result of the information extraction process is a platform independent model.
The third related approach to mention is Model-Driven Composition of Information Systems from Shared Components and Connectors by Leone et al[LSM13]. The teams proposes a process that on the basis of an UML model, generates Java source code. The UML model has to be created by the developer with the help of the *CompIS* platform which is introduced by Leone et al. The platform is basically a UML editor that supports the developer in creating the composition of the information system. Figure 2.2 displays the general iterative model driven development process. The UML editor reuses findings from the repository and passes them to the UML model, which is defined by the developer, to the code generator. This code generator generates Java source code and passes it to the *CompIS Object Database*, which is responsible for building the basis for the deployed code[LSM13][p.211]. The arrow from the *CompIS Object Database* to the UML editor expresses the iterative nature of this approach. The most interesting part of the findings presented by Leone et al, is the fact that they use a model driven development process to generate source code. In this case the code generation is used for the front-end and back-end.

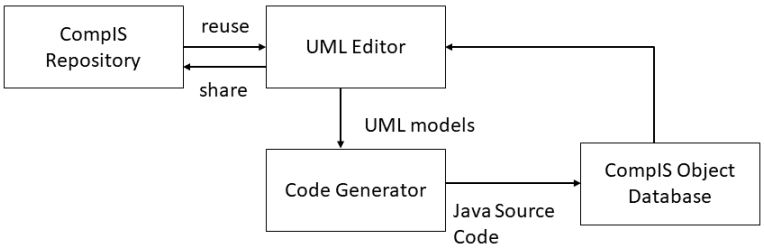


Figure 2.2: Simplified Model Driven Development Process[LSM13, p.211]

3 Platform Independent UI Model

This chapter is contemplated to give an overview of the early design goals of this thesis. It will feature a description of the view model, the adaption of the view model and the targeted process for the implementation part of the thesis.

3.1 View Model

The view model is an abstraction of the view. The view itself is part of the user interface including layout, structure and all contained elements. The idea of such a model is to be platform independent, so it can be used in any arbitrary environment.

3.1.1 UI View Model

Figure 3.1 displays one way to model an abstraction of the view. The UI consists of components, which themselves encapsulate either simple elements or composite elements. In Web Application Development simple elements could for instance be represented as HTML's p or span elements

<p>

which are neither nested nor representing a complex part of an UI.

Composite elements on the other hand are are more complex like for example a div wrapping another div

```
<div id="wrapper">
    <div id="wrappedElement"></div>
</div>
```

to represent the UI as nested components with implicit hierarchy. The model is an adaption of the well-known composite pattern, which itself is a common used design pattern in software engineering.

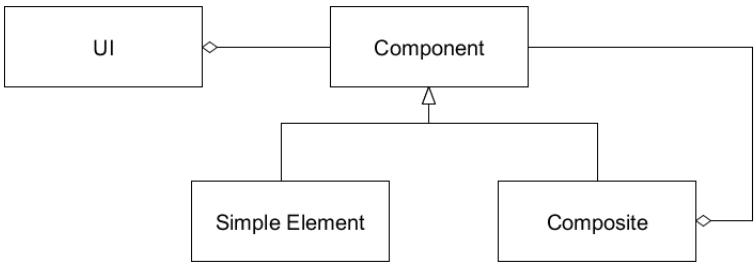


Figure 3.1: UI View Model

3.2 Separations of Concerns in UIML

A well-known approach to achieve true platform independency is the User Interface Markup Language (UIML) by the Organization for the Advancement of Structured Information Standards (OASIS). The design objective of this approach is to provide a vendor-neutral, canonical representation of any UI suitable for mapping to existing languages[OAS08]. All elements that are used in UIML are defined by the Extensible Markup Language (XML). An example for a general structure of an UIML document is visualized in figure 3.2. Each interface element, namely structure, behavior, content and style, is called a part[OAS08]. The presentation of the interface parts, is different for every platform or device. To provide logic to those different interfaces, they are connected to data sources, which can be applications, database management systems etc.

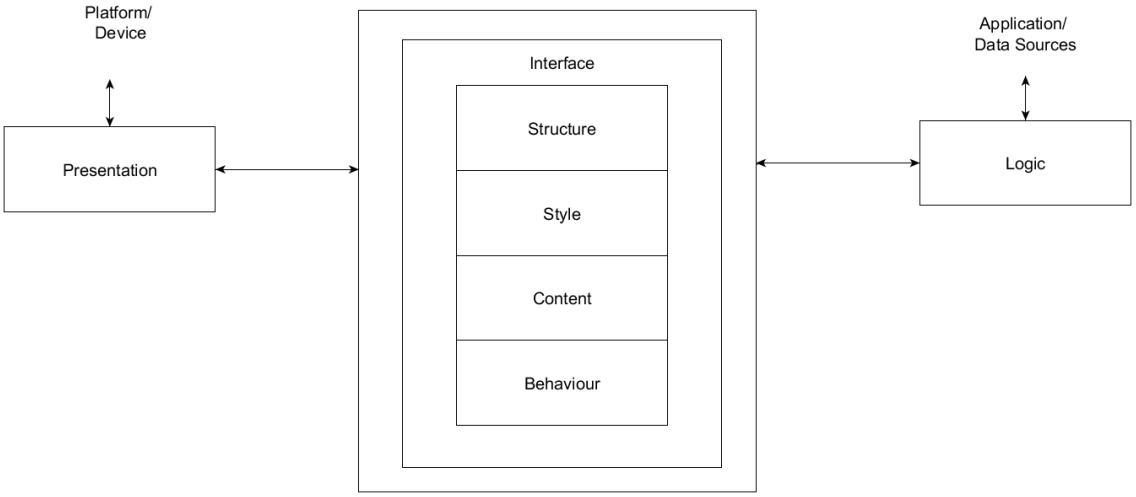


Figure 3.2: Separation of Concerns in UIML
[OAS08]

The UIML approach is a generic one and many of the elements are unnecessary for

the scope of this thesis. Figure 3.3 shows the simplified model as JSON schema[1]. As the target standard, which is W3C Web Components are as stated in section 2.4 component based, the model is more tailored to this approach and the generic term interface has been replaced by the term component. In this adaption components have their children attached as an array of components instead of the UIML way of redefining those elements each time. Furthermore the representation has been changed from XML to the JSON format. The main reason to chose JSON was because of the implementation which is written in JavaScript and JSON is the go-to interchangeable data format for the script language.
A component is modeled by its style and structure. Those parts are very important to render components. The first idea was to remove the content element completely, because the input SVG file does not supply any textual data, yet everything inside of this standard is vector based. As this may be true there are some cases, where an element of the SVG can supply textual information, e.g. when a *group* element is wrapping a component. This issue will be explained in detail in the design and implementation chapter 4. All assumptions for SVG are only valid, when the SVG is exported from one of the presented prototyping tools in section 2.3. The behavior element was removed completely, because it would go beyond this thesis' scope.

3.2.1 Structure

The interface portion of an UIML document defines a virtual tree of parts with their associated content, behavior and style[OAS08]. Figure 3.4 shows a simplified graphical representation of such a virtual tree. The standard also specifies the part element, which represents an instance of a class instance and can be nested inside each other[OAS08]. In the proposed adaption of the UIML standard, the structure parts, as stated above, have been replaced by a simple array of the component's children as well as a specific class name. The references of the child components will later make it easier to map the view model to a DOM structure. Class names on the other side introduce typing for the components.

3.2.2 Style

The style element in the UIML standard contains a list of properties and values that are used to render the interface[OAS08]. As the scope of these components will be tailored to their own shadow tree, every component can have its own style properties and does not need global unique identifiers. Therefore, an object for the component's properties, which in JavaScript is implicitly a key-value pair, is sufficient. This tree scoping ability is one of the advantages of the Shadow DOM, as stated in section 2.4.

3.2.3 Behavior

The UIML standard specifies a behavior element used to describe the interaction of an end-user with the user interface[OAS08]. This element was removed from the proposed view model, because it would go beyond the scope of this thesis. In general, SVG elements can be enhanced with script tags, similar to HTML[2].

[1] http://json-schema.org
[2] https://www.w3.org/TR/SVG/script.html

```
{
  "componentName": "",
  "type": "object",
  "properties": {
    "content": {
      "type": "object",
      "properties": {
        "text": {
          "type": "string"
        }
      }
    },
    "structure": {
      "type": "object",
      "required": [
        "className"
      ],
      "properties": {
        "className": {
          "type": "string"
        }
      }
    },
    "style": {
      "type": "object",
      "properties": {
        "property": {
          "type": "object"
        }
      }
    },
    "children": {
      "type": "array"
    }
  }
}
```

Figure 3.3: View Model Adaption of UIML in JSON Schema

3.2.4 Content

Another important part of the UIML standard specifies the content element. A part in an UI can be associated with various types such as words, characters, sounds, or images. Since UIML supports separation of content in an UI, it is useful to display different content under different circumstances like such as rendering different languages in the front-end[OAS08]. In modern front-end frameworks like for instance React.js, this issue is solved by a component's state. As stated before, the content element is used to display textual data such as labels and is therefore incorporated in the view model adaption.

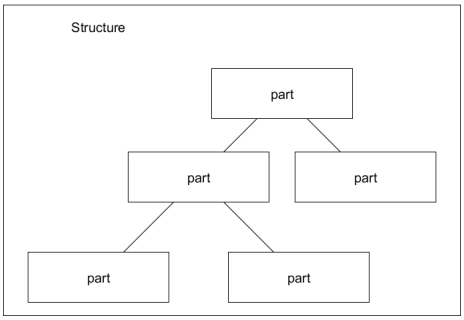


Figure 3.4: Example of the Virtual UIML Tree
[OAS08]

3.3 Proposed Process

This subsection is intended to provide an overview of the targeted process. Every part of the extraction and model generation process will be explained in its respective subsection. The respective parts are based on the findings of the research method and explains why the architecture was designed in this way.

3.3.1 SVG Parser

The first step is to convert the received SVG into the JSON format. JSON was chosen, because the target language is JavaScript and JSON is the programming language's standard interchangeable data format. Hence SVG is well-formed XML, many parsers are available for this task. The decision to use the xml2json[3] package was selected, because firstly it is well documented and lightweight and secondly it nearly worked out of the box without much configuration effort. The package itself uses node-expat to convert the XML file into the target JSON. Figure 3.5 presents an abstract view of the parsing process, with its corresponding input and output data. The output of this process is called *raw JSON*, because no modifications were made yet, but a transformation to JSON.
This part of the process does not directly correlate with any related approaches that were mentioned before. Nevertheless, the extraction process has to be defined on the basis of a data format and as the target programming language was defined to be JavaScript, the decision to use JSON can be seen as an advantage.

[3]https://github.com/buglabs/node-xml2json

Figure 3.5: SVG Parser

3.3.2 Extraction Process

The extraction process is the core part of the provided tool and an overview is given in figure 3.6. It is basically a transformation function that extracts information from the *raw JSON* and transforms it into a platform independent view model.
The extraction process correlates with the basic architecture of a platform independent reverse engineering process introduced by Chikofsky and Cross[CC90], as explained in 2.5. The first phase of the extraction process is adapted from the semantic analyzer, as it is a kind of preprocessor by preparing the data. It identifies the relevant parts of the object such as headers, wrappers and components and extracts this information. The view model greatly relies on the findings of this step, since the second phase of the extraction process, which is the application of the specific rules, can not be done without preprocessing.
The second phase of the extraction process is responsible for extracting the useful information from the processed object to form the information base of the following view model generator. This generator takes the information base as an input and composes the single components into a hierarchical tree to form the view model. This step is can be seen as an adaption from the view composer by Chikofsky and Cross.

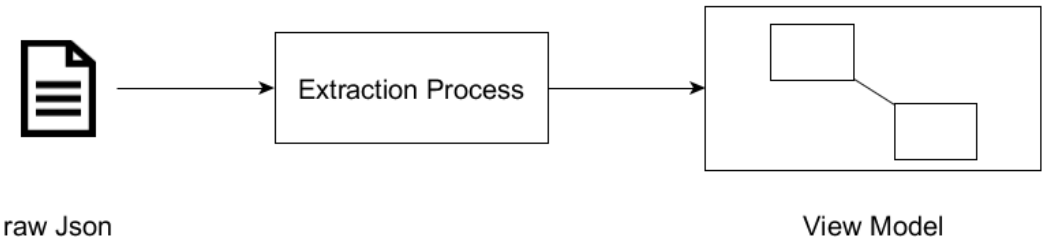


Figure 3.6: Overview Extraction Process

First Phase of Extraction Process

The extraction process applies rules to the input object, whereas every rule is responsible for one particular part of the object. A rule is a pure function, that receives the object as an input and returns the part of the object it is responsible for. If the rule can not be applied, e.g. when the component the rule is responsible for is not part of the input object, the rule just returns the object that was passed. As exemplified in figure 3.7 generic rules are chained. Generic rules are the ones, that generally need to be applied first, because they are important to remove unused parts of the objects or for the component hierarchy.
As the rule names suggest, *removeSVG* removes the SVG part of the input object, as it only holds vector information, that can not be assessed in the following process. *removeCanvas* removes the canvas object from the input, since every SVG exported via Figma is wrapped in a group element called *canvas*[4]. Furthermore *extractWrapper* and *rootComponentExtractor* are very important for the hierarchy of the view model and are thus chained before every other extraction rule. Those rules will be explained in detail in the implementation chapter 4 as well as an overview of all available rules.
The result of figure 3.7 is an intermediary JSON called *preprocessed JSON* from which encapsulating objects have been removed and an useful hierarchy could be extracted, which is then passed to the second phase of the extraction process.

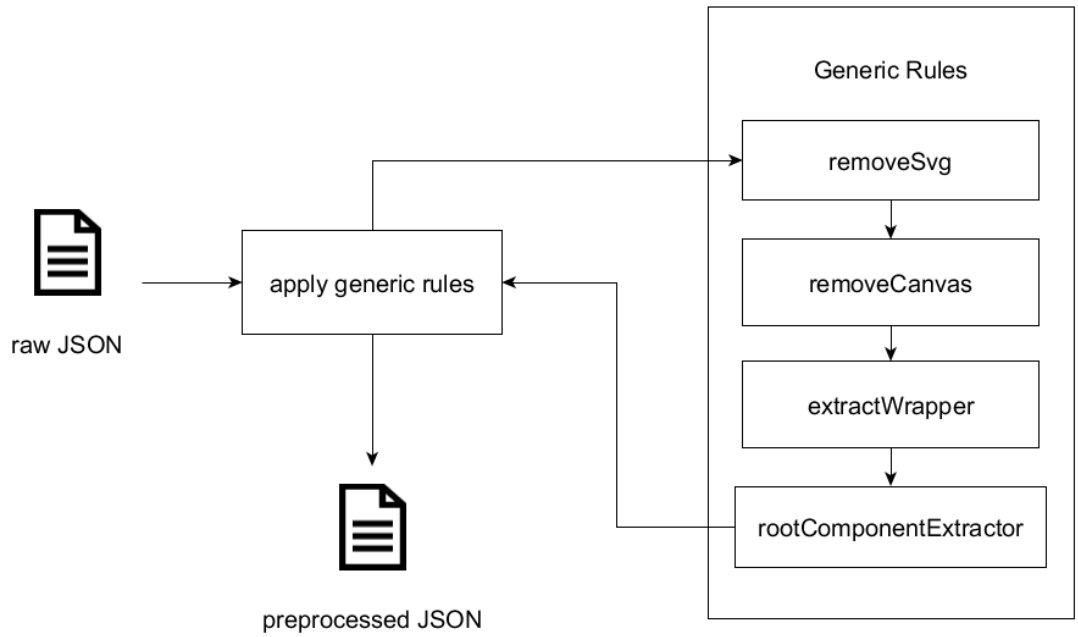


Figure 3.7: First Phase of Extraction Process

[4]https://help.figma.com/editor/canvas

Second Phase of Extraction Process

The second phase of the extraction process also features rules that are, as explained before, pure functions and are responsible for one part of the input object. The architecture of the second phase may look familiar but is far more extensible. In contrast to the first phase, the rules do not have to be applied in a particular order. Every rule takes the whole object, applies its logic and returns the part of the object that it is responsible for. Those accumulated values from each rule get merged at the end of the process to produce the target view model. Figure 3.8 shows four example rules that are applied to the *preprocessed JSON*.
This architecture allows the extraction process to be greatly extendable. As a result a new rule only has to be added to the array of applied rules or an existing rule can be modified, because neither of those possibilities does modify the other rules or disturb the process. The extensibility mentioned above though is only available for rules that are not generic, since those are important to remove unneeded parts of the object.
One of the goals of this thesis is to supply an architectural design and a set of rules that may be extended later. The explained architectural design complies with this goal and may be reused or extended easily.

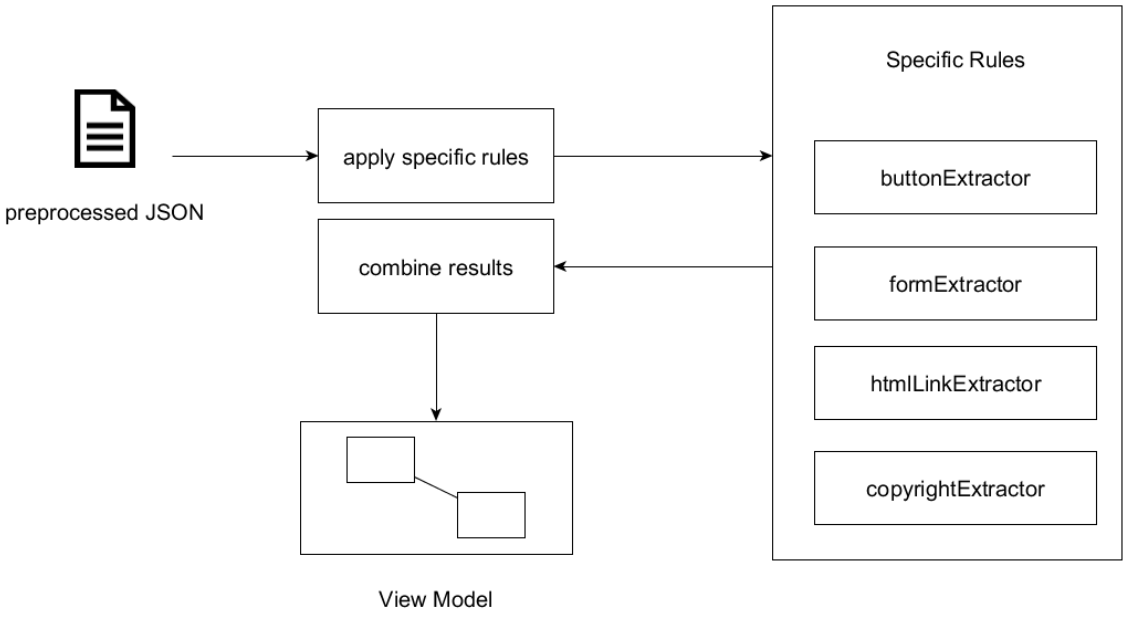


Figure 3.8: Second Phase of Extraction Process with Example Rules

3.3.3 DOM Abstraction

Popular front-end frameworks like React.js use its own DOM implementation (react-dom)[5]. The advantage of this approach is that the mapping of React components to the react-dom

[5]https://github.com/facebook/react/tree/master/packages/react-dom

implementation is much easier, since it is created on this behalf. At rendering time, React maps the components from its DOM abstraction to the browser's DOM.
This approach is also used in this thesis as recommended by the adviser. The components are mapped from the view model to a DOM abstraction. The framework used for this mapping is a lightweight DOM abstraction called *virtual-dom*[6].
The first step is to create a root node. Afterwards all children are appended as virtual nodes or DOM elements to the root node. The framework features the most important implementations of the DOM such as root nodes, appending children and rendering those as a tree. Figure 3.9 demonstrates the general structure of a DOM adapted from W3C schools[7].
The root element is directly attached to the document header and consists of two parts. The first element is the header, which is used to define general information about the document like exemplary modeled by the title element. The second element is the body where all child nodes are attached to. Those elements are exemplary modeled by a link element (a) and a headline (h1). Moreover, those elements could have children attached to form a deeply nested structure. All of those described structures are forming the DOM tree, which is then rendered by the browser.

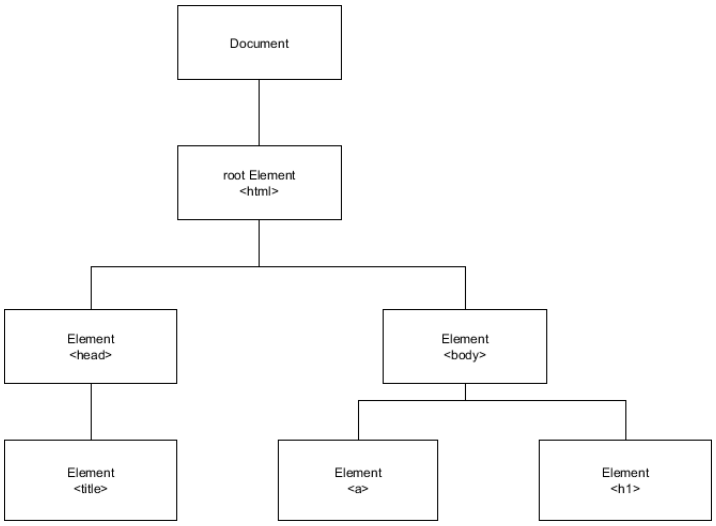


Figure 3.9: General Structure DOM

3.3.4 Component Generation

The component generation is the final step of this thesis' tool implementation. It reveals the generation and rendering of the extracted components as W3C Web Components. The W3C Web Components specification is designed with great foresight and is thought to be the web standard of the future. This is one of the reasons why the decision was made to render W3C Web Components and not as components of a component based framework.
The proposed process has some similarities with the approach of Leone et al introduced in

[6]https://github.com/Matt-Esch/virtual-dom
[7]https://www.w3schools.com/js/pic_htmltree.gif

section 2.5. The major differences between the approach in this thesis and the presented one, are due to the differences in domains and scope. For example the developer does not need to define the model by himself as it is composed by the extraction tool. Moreover, since the target language is not Java but JavaScript the output language of the generator is changed as well. In addition to the specific domain of UI prototypes the generation process produces only UI specific source code, as even the behavior parts are skipped. One more thing that was adapted was the kind of code generation. Leone et al used *Acceleo*[8] which is a open source code generator implementing the Model to Text Language (MTL) standard. The code generator uses a template based approach[LSM13, p.218] which was adapted by this thesis' component generation.
Figure 3.10 shows an overview of the component generation process. It receives the DOM abstraction as input and produces custom elements. The first step is to generate a HTML template for each of the extracted components. This leads to great reusability and extensibility as HTML templates can easily be imported in every HTML document. Every HTML template consists of a HTML part and a style part, which is in this case CSS. Both properties get wrapped inside a template tag. The result of this step is a HTML template that contains CSS and HTML information, but is only one document. This is why the figure is modeled with the HTML template on the top with CSS and HTML information attached.
The next step is to render this HTML template as a custom element. Every custom element gets implicitly wrapped in a shadow root which leads to two main advantages:

- Tree scoping.
  Prevents namespace issues and ensures that every custom element uses its own style

- Hides implementation details.
  The script part is hidden from the DOM.

The result is a custom element for every extracted component wrapped in its own shadow root. Those components can then be bundled in a shadow tree to form the result of the extracted view model.

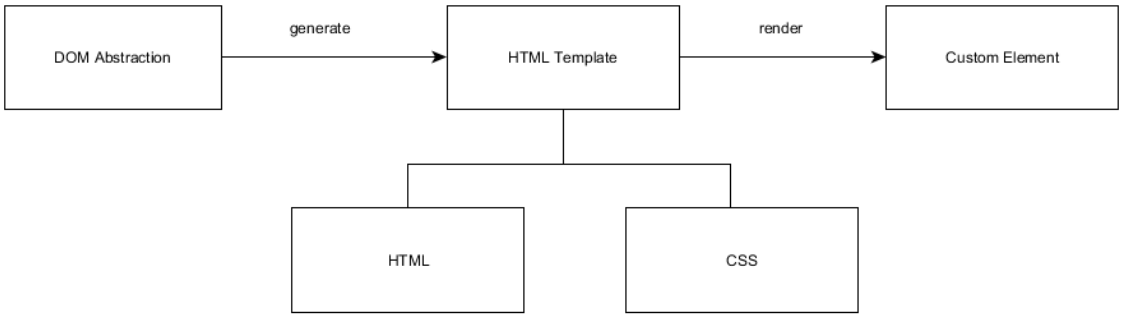


Figure 3.10: Overview Component Generation

[8]https://projects.eclipse.org/projects/modeling.m2t.acceleo

3.3.5 Component Model

The component model is an adaption of the IT4IT reference architecture. The reference architecture is defined by "The Open Group" and provides prescriptive, holistic guidance for the implementation of IT management capabilities for today's digital enterprises[9]. This architecture is very well suited to model the processes and the crafted artifacts.
Figure 3.11 visualizes the adaption of the IT4IT reference architecture. The rectangles represent the processes or responsible stakeholders within that process. The bubbles inside the processes represent the crafted artifact or the tool the artifact of the process was crafted with. The labels on the bottom represent the status of the current stage of the process.
Although the adaption has no defined starting point, because of its circular structure, it is self-evident that the beginning has to be the design guidelines. This process is contemplated to produce design guidelines, which helps the extraction tool to rely on some general rules. Later in chapter 4 this issue will be described in greater detail.
The next logical step is the designer who uses prototyping tools to craft the UI prototype. This process depends on the design guidelines. Therefore it is scheduled behind the design guidelines. The UI prototype is the input for the parse process.
This process is responsible for parsing the well-formed SVG to the *raw JSON* and prepares the object for the extraction process.
The *raw JSON* artifact is used as input for the extraction process. It produces a *result JSON*. This JSON object is an intermediary artifact between the *preprocessed JSON* and the view model. It is defined different in this model, since it is contemplated to provide a better overview of the architecture. The extraction process is explained in great detail in section 3.3.2. The *result JSON* artifact is passed to the view model generator.
The view model generator process is responsible for crafting the view model artifact. It includes the extracted components as well as their attached style and content information. The view model is platform independent and an abstraction of the source SVG.
The next step is to map the view model to a DOM abstraction. This is thought to simplify the component generation, as the components are firstly mapped to a simplified DOM and following that to W3C components. The mapper produces an important intermediary DOM abstraction, which is then passed to the component generator.
The component generator produces W3C components from the DOM abstraction. Therefore, the crafted artifact are the W3C components. Since the IT4IT reference architecture models the whole IT management capabilities the last process is representative for possible stakeholders.
The modeled entity for those stakeholders is a test user. This test user verifies the prototype according to certain criteria and may trigger the process from the beginning. In this case the whole process unleashes its whole power, since it is nearly fully automated. The only bottleneck is the designer, who needs to modify the UI prototype according to the tester's feedback.

3.3.6 Data Flow

Figure 3.12 shows an overview of the data flow of the whole process. It starts with the *SVG*, which is exported via the Figma prototyping tool. This data serves as an input for

[9] http://www.opengroup.org/IT4IT/referencearchitecture

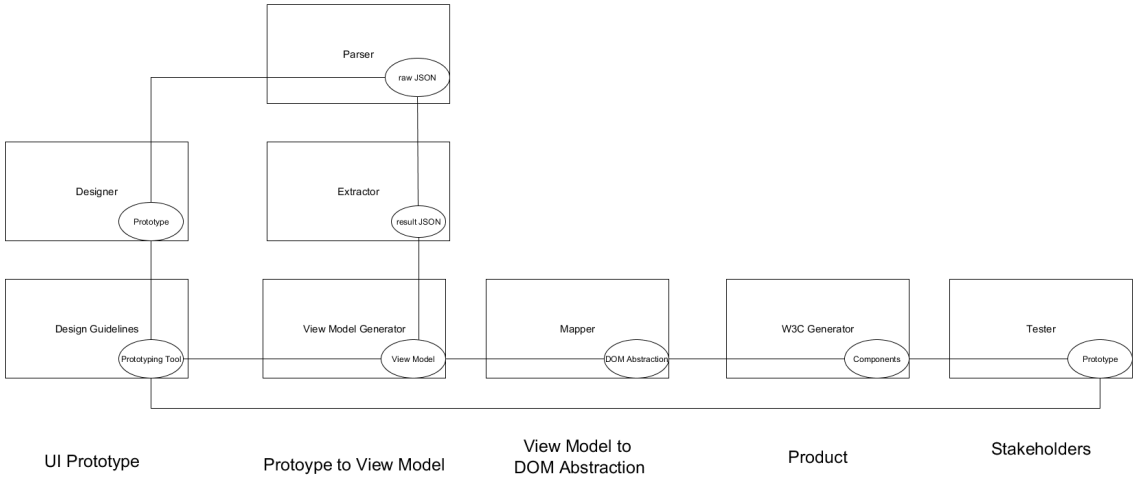


Figure 3.11: IT4IT Reference Architecture Adaption

the SVG Parser, which parses the well-formed XML to JSON. The output of this process is the *raw JSON*, which is itself the input for the extraction process. The first phase of the extraction process produces a *preprocessed JSON*, which is handed to the second phase of the extraction process to be transformed to the platform independent view model. In the last step the view model is mapped to W3C Web Components by the component generation process.

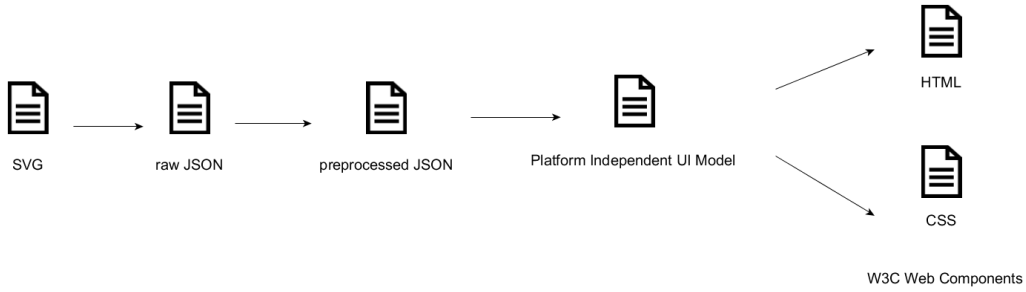


Figure 3.12: Overview Data Flow

4 Design and Implementation

This chapter explains in detail how the extraction tool was implemented. It starts off with an overview of the used frameworks and environments as well as the chosen development approach. It then continues with the explanation of the JavaScript module structure, explains rules in detail and presents examples from the code base.

4.1 Development Approach

In the following the chosen development approach is explained in general and to which extend this approach is used. Moreover, the used ECMA-Script stages, plug-ins and most important frameworks that were used to support the implementation are illustrated.

4.1.1 ECMA-Script Environment

New ECMA-Script features are designed by the Technical Committee 39(TC39). Its members are companies and all vendors of modern browsers[1]. New feature requests or proposals have to get through the so called TC39 process, which features four stages. Stage-0 also called strawman phase is a way of submitting ideas for evolving JavaScript, so it is pretty informal yet. The next stage, stage-1 includes a formal proposal for the feature. If this proposal is accepted by the TC39 the feature ascends to stage-2, called the draft. As the stage name indicates, a first version will be added to the specification. The feature now has to include a formal description of syntax and semantics. Stage-3 is the candidate phase, where the proposal is finished to a great extend and is now waiting for feedback from implementation and users. The last stage, stage-4 embodies that the feature has been accepted and will be included in the standard. It has to complete some standard tests as well as to show some significant practical experience with its implementation.

Import and Export Syntax

The feature, greatly used in the implementation part of this thesis and defined in the ECMA-262 Specification, is the import statement[2]. It lets the developer define default and named exports as well as importing those in your modules. Named members are imported via the following syntax:

```
import { member } from "module-name"
```

On the other hand default exports can be imported to a module via

```
import defaultMember from "module-name"
```

[1] http://2ality.com/2015/11/tc39-process.html
[2] https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/import

The default export should be chosen wisely, since there is only one default export allowed per module. Those are the most used features but there are many more as aliasing members or importing the namespace of a module with:

```
import * as name from "module-name"
```

4.1.2 Compilers and Frameworks

This subsection is meant to illustrate the most important frameworks that are used as well as the JavaScript compiler Babel.js.

Babel

Babel.js [3] is by far the most popular code compiler for next generation JavaScript. It allows the developer to write future syntax, for instance syntactical sugar that is not yet supported by the modern browsers. Babel transforms source code into a syntax that browser can execute nowadays. The developer is able to determine the ECMA-Script stage in the *babel.rc*-file or can use single babel plug-ins, that implement certain transformation rules. One of the most used features of future JavaScript in this thesis, was the object rest spread operator. This operator has many advantages when working with objects, like applying objects to objects or updating current object's properties. Those abilities also come in very handy, when working with immutable data structures like for instance React.js properties. Since the operator always returns a new object, in case of assigning new properties to an object, instead of mutating the current object.

Prettier.js

Prettier.js [4] is an opinionated code formatter, which supports JavaScript, JSON, JSX but also syntax like GraphQL and Flow. It evolved from ESLint[5], which calls itself a pluggable linting utility for JavaScript and JSX. The big advantage comparing Prettier and ESlint is the very limited configuration options in Prettier. This defines a clear code style for projects and helps the developers in teams to read each others code more quickly. Furthermore it assures code formatting over all phases of the development process. In this thesis the framework is used to format all of the written code.

4.1.3 Lodash

Lodash [6] is a modern JavaScript library that brings aspects of functional programming to JavaScript. It is great for mutating JavaScript objects and arrays and provides many helper functions. In the case of this thesis, the lodash library was mostly used to iterate over the *preprocessed JSON* and to identify components in that object. The library is fast and easy to use and is of great use when working with data structures.

[3] https://babeljs.io/
[4] https://github.com/prettier/prettier
[5] https://eslint.org/
[6] https://lodash.com/

4.1.4 Test Driven Development

Test driven development (TDD) is one of the most fundamental processes in agile software development. The approach produces loosely coupled and highly cohesive source code [AMS06, p.154]. When developing with test driven development, the way of thinking about the implementation is different. A developer thinks about the desired functionality of his implementation, then writes the first test, which then constructs the implementation step by step. The result is cohesive code and the developer is protected from refactoring his code over and over again. Obviously it leads to a higher code coverage[AMS06, p.154], which prevents unexpected errors or bugs. Nevertheless, writing code with this approach expends a lot of time and often feels like creating lots of boilerplate code especially when unit testing.
For the scope of this thesis a test driven development approach is well suited because it is easy to unit test every single rule. This evolves from the fact that rules are pure functions, which means that they create the same output every time an equal input has been passed. After testing all the rules independently the next step is to test the combination of them. This is not as time expending as for instance system or integration tests, which often feature tests of user interfaces, data binding and user behavior.

4.1.5 Test Framework

The most important criteria that were taken into account were the following:

- Unit testing framework
- Easy configuration
- Fast and lightweight
- Coverage report

There were two frameworks taken in account after an initial analysis namely, Facebook's Jest[7] as well as the combination of Mocha[8] and Chai[9]. The bundled power of Mocha and Chai is very popular among the JavaScript world and though have evolved so called *flavors*. Those flavors are developed for a certain reason for instance expect/should or assert syntax. Moreover, they are often just syntactical sugar for writing tests more readable. Mocha is the corresponding test runner.
Jest however, was built on top of Jasmine[10], which itself is a unit testing framework. Over time Jest has moved away more and more from Jasmine mainly due to the goal of developing a matcher tailored for Jest's requirements[11] . Nevertheless, Jest inherited Jasmine's describe syntax, which is the way tests were written in this thesis. In the meantime Jest developed its own test syntax, which is very similar to the stated test describing logic.

[7] https://facebook.github.io/jest/
[8] https://mochajs.org/
[9] http://chaijs.com/
[10] https://jasmine.github.io/
[11] https://www.heise.de/developer/meldung/JavaScript-Unit-Test-Framework-Jest-15-verabschiedet-sich-von-Jasmine-3311094.html

After a quick evaluation the decision was taken to use Facebook's Jest, because it satisfies very well the mentioned criteria. The framework does not need any configuration in the scope of this thesis as well as it is contrary to Mocha and Chai test runner and test describer, bundled in one application. When executing Jest, the tool looks for any files in the project that match a test pattern and executes them. The structure of project will be shortly explained in the following subsection 4.1.6. On top of all that, Jest has a built in coverage report which is supplied by the reporter tool Istanbul.js[12].

4.1.6 Test Structure

An exemplary overview of the test structure inside the project is presented in figure 4.1. The root project folder itself contains two folders. Firstly, the *src* folder, where every JavaScript file is located and inside of this folder other folders can be found for structuring the files and logic like the *rules*- or *componentRules* folder. Secondly, there is the *test* folder, which has exactly the same structure and files as the *src* folder. It obviously contains all the tests for the corresponding JavaScript files. Jest finds these tests easily when adding the string test between the JavaScript file to test and the file ending, for instance *formExtractor.js* is the JavaScript file to test and *formExtractor.test.js* is the corresponding test file. This folder structure is favorable, because every test file is easy to find even if the developer has not written the test by himself.

```
extractor
├── src
│   ├── rules
│   │   ├── rootComponentExtractor.js
│   │   └── etc
│   ├── componentRules
│   │   ├── buttonExtractor.js
│   │   ├── formExtractor.js
│   │   └── etc
│   └── index.js
└── test
    ├── rules
    │   ├── rootComponentExtractor.test.js
    │   └── etc
    ├── componentRules
    │   ├── buttonExtractor.test.js
    │   ├── formExtractor.test.js
    │   └── etc
    ├── index.test.js
    └── testObject.js
```

Figure 4.1: Overview Test Structure

[12] https://istanbul.js.org/docs/tutorials/mocha/

4.2 Extraction Process

4.2.1 First Phase of Extraction Process

Since section 3.3 is already covering the concept of the first phase of the extraction process, the goal of this section is to analyze the structure of the exported SVG and how to transform it the *preprocessed JSON*.
As the SVG is well-formed XML and the parser transforms it to JSON the *rawJSON* is often a deeply nested object. Therefore, the JSON does not lose any structural information, which is really important to create the view model later.
Some structural elements in the *rawJSON* are generally very similar, because they are exported via the Figma prototyping tool. Figure 4.2 displays a simplified structural model of a typical SVG file.

The first part of the object is obviously the SVG header. It often contains some information about the size of the SVG like width and height, some XML related properties such as *name-space* and *xlink*. Moreover it includes a version number and a view box, which specifies a rectangle in user space which should be mapped to the bounds of the viewport[13]. Lastly the SVG header includes a description with the property indicating, that the SVG was created by Figma.
The following element is the canvas object, indicated by the identifier *canvas* and followed by a group object. This group object wraps the rest of the SVG and has also an identifier that often indicates the name of the current view.
The next element is the first that may contain information that can be extracted and because of that is called *rootComponent*. Components are always indicated by group object arrays. The objects inside of that array always have an identifier and some style properties and are signaling the start of the particular components. This is where the extraction process really starts and the first phase of the process is finished.

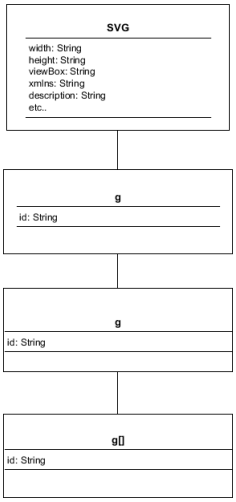


Figure 4.2: Hierarchic Structure of a SVG File

[13]https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/viewBox

4.2.2 Second Phase of Extraction Process

Hereafter the general rule structure will be presented by illustrating an example as well as an overview of all available rules.
Rules are pure functions that take a part of the *preprocessed JSON* as input and return either the component or just an empty object. Those rules are invoked by the *viewModelExtractor* module, which schedules the parsing of the SVG object, the preparation of the object, the application of all available rules and in the end combines the result to the view model. Every part of the raw input object that is meant to be an object, is applied to the rules.

Specific Rules Overview

Figure 4.3 reveals an overview of all available component rules. Each rule's name indicates the type of component they are responsible for.
As stated above, those rules are applied to the *preprocessed JSON* by the *viewModelExtractor* module. The results of every single rule will in the end get combined to form the view model. If a rule fails, it will just return an empty object, which will get filtered out and not be part of the result view model. Every main function of each rule, which is responsible for identifying the component, extracting the important parts like content and style and returning the result object, is exported to a bundling module. In this module those main functions get composed to an array of functions and exported via the default export syntax. This default export is imported by the *viewModelGenerator* module which itself iterates over this array and tries to apply the rules to the *preprocessed JSON*. Those two loops form the main part of the view model generation process.

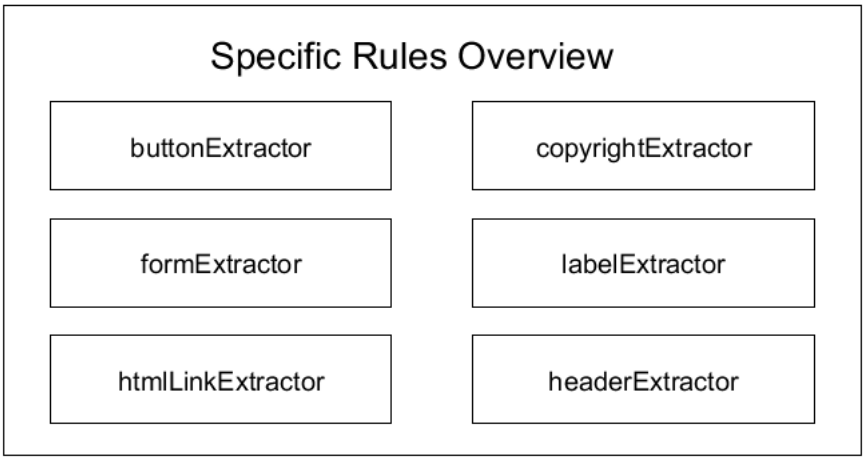


Figure 4.3: Overview of Specific Rules

Specific Rule Example

To clarify the general functionality of a rule, figure 4.4 shows the flow chart of an exemplary rule, namely *formComponentExtractor*. The particular steps are a summary of the most

important function calls, assignments and validators.
The *extractFormComponent*-function is invoked by the *viewModelExtractor* module and is the main function of the extractor. It applies the input object to various validators, calls other rules and at the end of the function returns a result.
The first step after the invocation is to initialize a result object as well as an identifier for the wrapping component. Secondly, the first validator is applied to check if the received object has a form component structure. When this conditional statement evaluates to false, the current result is returned, which is an empty object at this moment. This empty object gets recognized by the *viewModelExtractor* and therefore the output of the rule is ignored.
If the conditional statement evaluates to true the *getFormAndLabelParts*-function, which is also part of the *formExtractor*-module, will get called. This function returns the label- and form-part and assigns them to their respective variables.
Following this, another conditional statement checks if a form-part could successfully be extracted. If it evaluates to false, the result is returned, which is still is an empty object. The validator only checks the form-part, as a form component could be extracted without a valid label-part but not without a form-part.
If a form-part is present another assignment follows, which itself calls two extractors that return the form and label for the *formComponentWrapper*. This wrapper is contemplated to be the final result, if every validation is successful.
Following this, the next step evaluates if the label-part has a useable identifier, that later could be used as global identifier of the wrapping component. This identifier is obviously not available, if no label-part is present. Therefore, this conditional statement has to be called before combining the particular parts to the result and returning it. The returned component conforms to the rules of the adapted view model and is integrated into the component hierarchy of the view model.

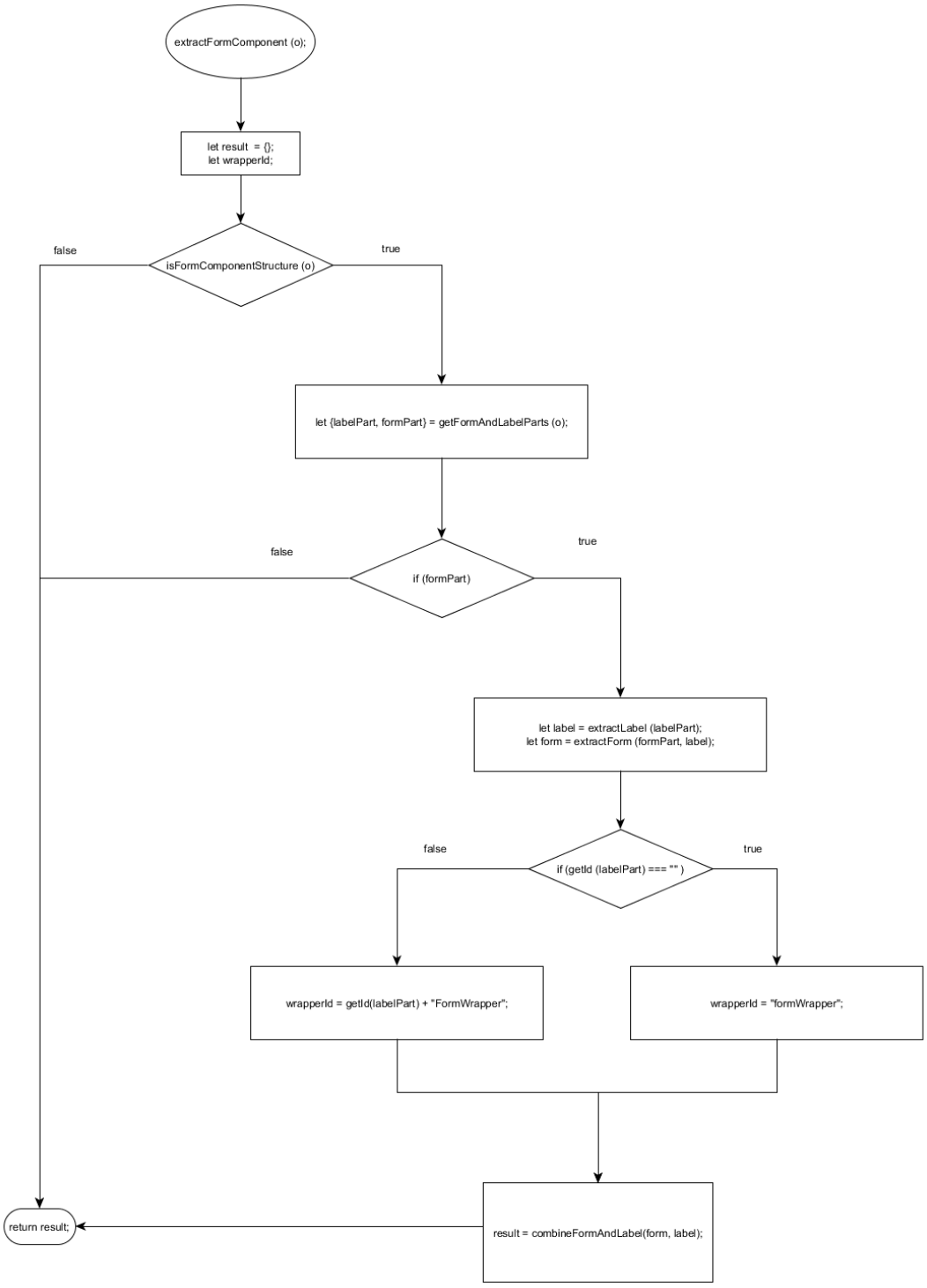


Figure 4.4: Flow Chart Form Component Extractor

4.3 DOM Abstraction

The DOM abstraction is contemplated to simplify the mapping from the view model to the target DOM.
Figure 4.5 displays on the left side an empty view model and on the right side the root node with one virtual node attached as child. The component name is directly mapped to the tag name. As the first extracted component always is the *rootComponent* its values will be mapped to the node. Node name is only present, if it is a DOM element. On contrary the node field is empty, when the element is of type virtual node.
The class name is also directly mapped when the current element is a DOM element. If the element is a virtual node, its class name is mapped to the properties object. The properties of the array are also responsible for holding the element's style information.
The content part of the component is rendered as virtual text and attached to its parent's children array. This is a way of enabling an easy mapping from the view model to the DOM abstraction.
All of the other displayed DOM elements or virtual nodes are not used in the scope of this thesis because they are either not having a valid mapping partner on the side of the view model or are handled by the virtual DOM framework. This mapping to a virtual DOM also enables child parent relationships in both directions, since the child nodes have their parent node attached at the bottom of the virtual node. Contrarily the view model only visualized parent child relationships but not vice versa.

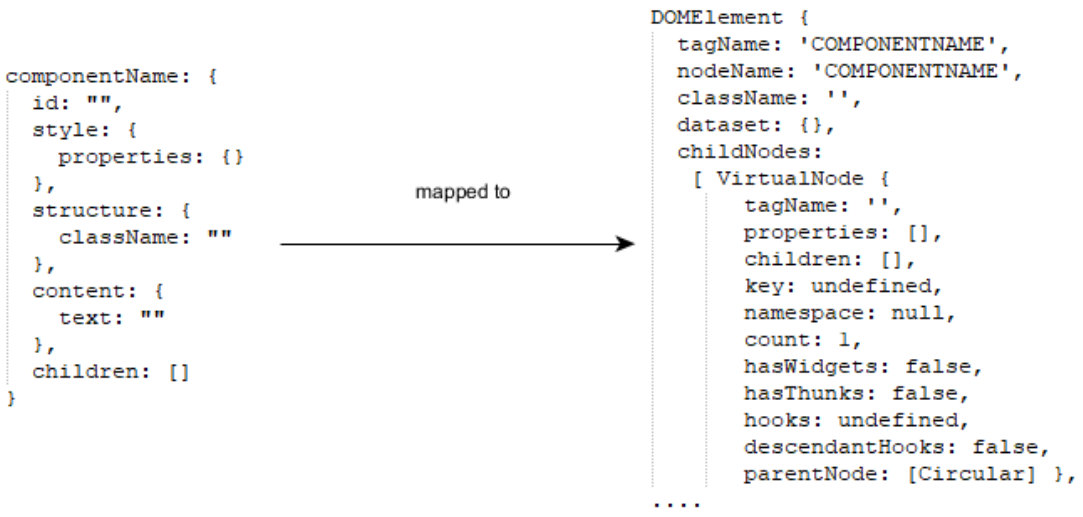


Figure 4.5: Mapping Empty View Model to Virtual DOM

4.3.1 Exemplary View Model to DOM Abstraction Mapping

Figure 4.6 shows an exemplary mapping of a *rootComponent* and its first child a copyright component. The mapping of the *rootComponent* is pretty obvious and is essentially easy to match. The node is created with the identifier of the component and its component name is used as tag name, which happens to be equal in this specific case.
The *rootComponent's* first child is in the visualized view model a copyright component. This leads to rendering the component as a virtual node of the *rootComponent*. Its tag name is mapped from the component's identifier. Moreover the style and structure parts are mapped to the properties array inside the virtual node. It must be kept in mind that the fill property has been mapped to color, since it is the valid property in the CSS world. Lastly the content element, which is a text, is mapped as child to the component. The type of this element is *VText*, as it is responsible for the plain text part of a DOM element. The last element that is filled by the mapping process is the *parentNode*, which happens to be the *rootComponent* DOM element.

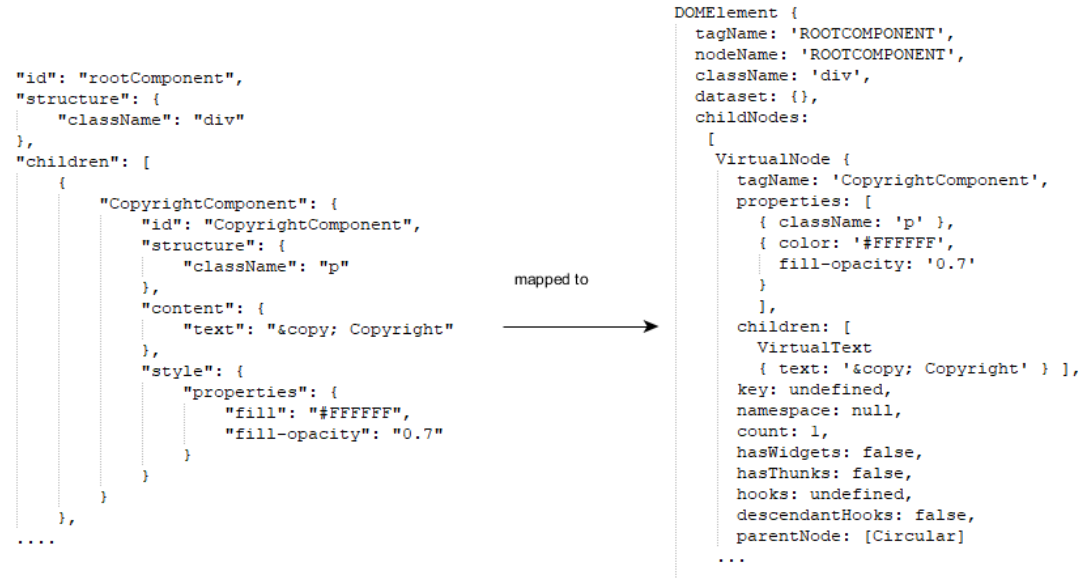


Figure 4.6: Exemplary Mapping to Virtual DOM

4.4 Component Generation and Rendering

This subsection describes the component generation and how they are rendered with the help of the Polymer-2 starter kit[14]. This step finishes the implementation part.

Web Components Naming Conventions

The web components specification features clear rules on how to name components. The specification implicates that every element has to contain a dash to force the developer to add a namespace [15], for instance *johns-tabs*. Anything without a dash will throw an error. The text before the dash should be used as namespace, whereas the text after the dash is used to describe the element or the type of element.
For the implementation part the component names were chosen according to the rules prefixing every component with "my" and adding the identifier of the component after the dash, for instance *my-button1*.
Moreover, there are reserved names that need to be avoided:

- annotation-xml
- color-profile
- font-face
- font-face-src
- font-face-uri
- font-face-format
- font-face-name
- missing-glyph

4.4.1 Component Generation

The component generation produces a separate HTML file for each component and names them after the web components naming conventions described above.
In the first step, the skeleton for the creation of *Virtual Nodes* is visualized and following this a skeleton for the creation of a DOM element is described.

Virtual Node Skeleton

Figure 4.7 visualizes the skeleton of a HTML template used to create Web Components on the basis of a *Virtual Node*. The style part consists of the *className* followed by its style properties. Following style, the element is rendered with its *className* as element type and its unique identifier. If the component has a content part the text is put between the opening and closing tag brackets.

[14]https://www.polymer-project.org/2.0/start/toolbox/set-up
[15]https://www.webcomponents.org/community/articles/how-should-i-name-my-element

```
<template>
    <style >
    className{
      style.properties
    }
    </style>
    <className id="id">contet.text</className>
  </template>
```

Figure 4.7: Virtual Node Skeleton of HTML Template

Figure 4.8 shows an exemplary HTML template of a button component. The style properties were adapted to the respective syntax. The *classNames* were resolved to *button*. The element's identifier is defined by the component's identifier. Lastly, the button component has a text attached and therefore this text is also mapped to the rendered button.

```
<template>
    <style >
     button{
      transform: translate(-156, 90);
      color: #2F80ED;
      background-color: #FFFFFF;
     }
    </style>
    <button id="button1">Submit</button>
</template>
```

Figure 4.8: HTML Template Button Component

DOM Element Skeleton

As described in the DOM abstraction part in section 4.3, components with children are mapped to DOM elements. Figure 4.9 visualizes the template for a component with two children. Obviously a component can inhere more children but for simplicity reasons this template features just two children.
The *wrapper* may have some style properties in the respective style section of the template. The style properties of the children are also attached to the style section. In the element rendering part, just after the closing style tag, the wrapper encapsulates its children. Both children may have properties according to their element type but always an unique identifier.

```
<template>
 <style>
  div{
   wrapper.style.properties
  }
  className1{
   child1.style.properties
  }
  className2{
   child2.style.properties
  }
  </style>
  <div id="wrapper">
   <className1 id="child1.id">child1.content.text</className1>
   <className2 id="child2.id">child2.content.text</className2>
  </div>
</template>
```

Figure 4.9: DOM Element Skeleton of HTML Template

4.4.2 Component Rendering

The rendering part of the components is greatly supported by Polymer 2.0[16]. The framework is well-suited to support the generation of custom elements inside its own shadow tree by working with a *dom-template*[17]. This *dom-template* allows the developer to specify the HTML template completely in markup, which is according to their documentation the most efficient way of defining HTML imports. By default, if an element has a shadow DOM, the shadow tree is rendered instead of the element's children. Through this setup tree scoping and all the neat features of shadow trees are enabled for the element.
Figure 4.10 shows the complete skeleton of the used *dom-module* template.

```
<dom-module id="my-header">
  <template>
    <style>...</style>
    <header>
      <h1>I'm a header</h1>
      <button>Menu</button>
    </header>
  </template>
</dom-module>
```

Figure 4.10: Template Definition using Polymers' *dom-module*

[16]https://www.polymer-project.org/2.0/start/
[17]https://www.polymer-project.org/2.0/docs/devguide/dom-template

JavaScript Part

The defined HTML template at the current stage could be imported to other HTML documents and could be rendered as custom elements in a DOM environment e.g. in modern browsers. For the scope of this thesis the decision was made to render those custom elements in the Polymer 2.0 starter kit. The reasons why this approach was chosen is explained in further detail in the following section 4.4.3. Because of this approach there has to be some JavaScript added to render the single components and in the end the render the bundled component inside of an application.
Figure 4.11 illustrates the complete skeleton of the used approach. On top of the file there is a HTML import, which includes the *Polymer Element* that is later used to be extended by the component. Following the import the *dom-module* is defined. This section is already described above. After the closing template tag the beginning of the JavaScript part is signaled by the opening script tag. Inside the tag, the component is defined as a class and extends the imported *Polymer Element* from the top of the file. Through the static "is"-getter Polymer is able to retrieve the *dom-module* for the element and to clone the current template's contents into the element's shadow DOM[18].
Lastly, the new class is associated with an element name[19]. Through the class syntax, which is primarily syntactical sugar over JavaScript's existing prototype-based inheritance[20], the developer could also expand created HTML templates by extending the class, in this case *MyComponent*.

[18]https://www.polymer-project.org/2.0/docs/devguide/shadow-dom
[19]https://www.polymer-project.org/2.0/docs/devguide/registering-elements
[20]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

```
<link rel="import"
  href="../bower_components/polymer/polymer-element.html">
<dom-module id="my-component">
 <template>
  <style>
   className{
    style.properties
   }
  </style>
  <className id="component">content.text</className>
 </template>
 <script>

 class MyComponent extends Polymer.Element {
  static get is() {
   return "my-component";
  }
 }
 window.customElements.define(MyComponent.is,
 MyComponent)
 </script>
</dom-module>
```

Figure 4.11: Complete Skeleton HTML Template

Exemplary HTML Template

Figure 4.12 shows a complete example of a generated button component. The generic placeholders such as *className*, *style.properties*, *text* and *my-component* were replaced by their respective values. This results in a custom element wrapped inside a HTML template that could be imported into any application and could be rendered in a DOM environment. All implementation details and style properties are scoped to its shadow root and are therefore not affecting or affected by other elements. Only some generic properties are inherited from components that are higher in the component hierarchy.

```
<link rel="import"
    href="../bower_components/polymer/polymer-element.html">
<dom-module id="my-button1">
 <template>
  <style>
   button{
    transform: translate(-156, 90);
    color: #2F80ED;
    background-color:#FFFFFF;
   }
  </style>
  <button id="button1">Submit</button>
 </template>
 <script>
 class MyButton1 extends Polymer.Element {
  static get is() {
   return "my-button1";
  }
 }
 window.customElements.define(MyButton1.is,
 MyButton1)
 </script>
</dom-module>
```

Figure 4.12: Complete Example of Generated Button Component

4.4.3 Polymer Project Structure

The Polymer 2.0 starter kit was used for the view part of the implementation part, because it is well-suited to quickly deploy and display generated components. The project template is set up to follow the PRLP pattern for efficient and progressive loading of applications[21]. The PRLP pattern stands for:

- Push critical resources for the initial route.
- Render initial route.
- Pre-cache remaining routes.
- Lazy-load and create remaining routes on demand[22].

The project is generated by the Polymer-CLI and the output is visualized in figure 4.13. This figure is adapted from the official documentation of the toolbox setup[23]. The most important files of this project are:

[21]https://www.polymer-project.org/2.0/toolbox/templates
[22]https://www.polymer-project.org/2.0/toolbox/prpl
[23]https://www.polymer-project.org/2.0/start/toolbox/set-up

- index.html
  This is the main entry point for the application.

- src folder
  All generated components should be in this folder. Inside this example the my-app.html file is the top-level element and all other elements are bundled inside of this file. Consequently the my-app.html file is imported to the index.html. This is a well-known approach for decoupling components.

- service-worker.js and sw-prechage.config.js
  This is the generated service worker and its configuration file. The service worker is responsible for offline caching critical resources[24].

```
bower.json
images
index.html
manifest.json
package.json
polymer.json
service-worker.js
src
    my-app.html
    my-icons.html
    my-view1.html
    my-view2.html
    my-view3.html
    my-view404.html
    shared-styles.html
sw-precache-config.js
test
```

Figure 4.13: Overview Polymer Project Structure

Adaption of Polymer 2.0 Starter Kit

The Polymer starter kit is used as a template in the implementation part, because it was easy to set up and greatly displays newly generated components in an overview. The components are generated by the *componentGenerator*-module. After their generation the components are written to a file and saved inside the source folder of the Polymer application. Moreover, the my-app.html is generated too, since all generated components should be rendered inside the application. The Polymer-CLI provides a command for serving the components from the source folder. Shortly after the generation of the components they are visualized after in the application. The only components of the Polymer's source folder that are not generated are:

[24]https://www.polymer-project.org/2.0/toolbox/service-worker

- my.icons.html
  This file contains icons that are required by the view. These icons are for instance used in the application's header.

- my-view404.html
  This is just a fallback view if anything goes wrong. Should loading the generated components fail, it does not break the application.

Figure 4.14 visualizes the source folder of the Polymer project after the components were generated. As stated before the my-icons.html and my-view404-html are the only files that were not generated. The rest of the source folder is generated by the *componentGenerator*-module.

```
src
├── my-app.html
├── my-button1.html
├── my copyrightComponent.html
├── my-icons.html
├── my-loginFormWrapper.html
├── my-Register.html
└── my-view404.html
```

Figure 4.14: Overview Polymer Source Folder after Component Generation

After serving the components the single components are visualized in the application. Figure 4.15 shows the application with some generated components. On the left there is a selection view where every component can be selected and viewed singled up and on the bottom there is the bundled component. This component contains all generated components and is therefore the result of the whole extraction and generation process. The view displayed in the screenshot is situated after clicking the *View button1* reference. The main page then loads the HTML template and renders the custom element underneath the MyApp-header. In this case the Submit-Button is rendered.

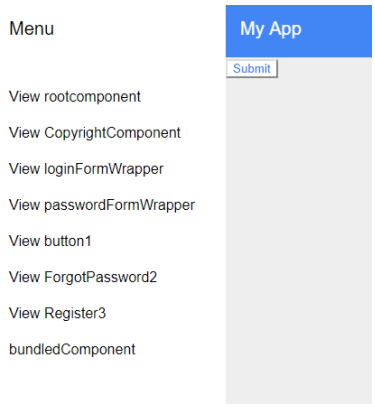


Figure 4.15: Screenshot Entry Point MyApp

5 Evaluation Use Cases

The goal of the evaluation is to assess the capabilities of the implemented extraction tool in context of user interfaces that were designed with either Sketch or Figma and were accessible online free of charge. The evaluation focuses on the extracted view model and how comprehensive this model represents the real world example.

5.1 Evaluation Approach

The evaluation of each use case is based on the expected and the actual view model. The expected view model was crafted by hand on the basis of the *rawJSON* since at this point of the process no transformation has been performed yet. All elements in the expected view model were counted and compared to the actual view model. The evaluation differentiates between identified components, extracted components and extracted properties. This approach was chosen, because firstly identified and extracted components are not always equal. Secondly, the properties are the core part of the styling part, which are very important for rendering the components according to the W3C specification. Moreover, each phase of the extraction process faced many challenges concerning the mapping of the components to the view model. To explain these issues every section of the extraction process features its own problem identification section. Those problems are divided in the part they arose, namely problems identified with structure, content and style.

5.2 Fridge App Login View

This subsection features a walk-through explanation of the first use case, namely an input form. The example[1] was taken from *uplabs.com*, which is a community website for graphical user interfaces. Designers present their creations and some of them are available free of charge. The example features different views of an application for iOS. As a starting point the login view was chosen.

5.2.1 Fridge App Login View

Figure 5.1 displays a screenshot of the user interface. This UI prototype was designed with the design tool Figma, using its power explained in section 2.3.2. It features a simple login view where the users can submit their credentials like phone number and password. Moreover, it has one button to submit the credentials, as well as a header. The following consists of two links, one is for registering for the application and one helping in the case of a forgotten password. On the bottom of the UI a copyright tag is located.

[1] https://www.uplabs.com/posts/fridge

This prototype can be exported as SVG via Figma and then passed to the extraction process.

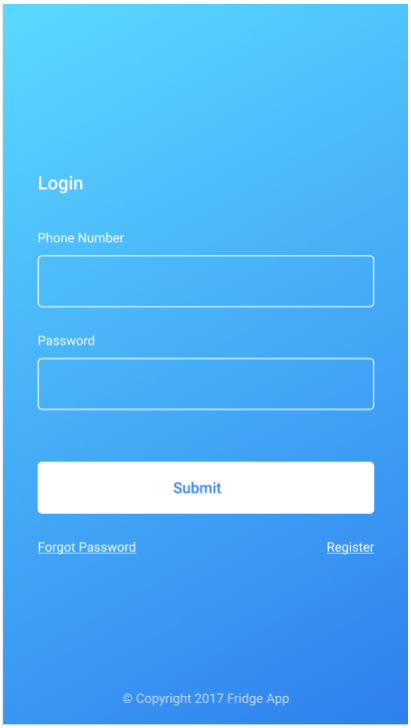


Figure 5.1: Fridge App Login View

5.2.2 SVG and rawJSON

As stated in section 4.2.1 the exported SVG follows certain rules that support its parsing. Figure 5.2 shows an object model of the structure of a SVG file and which part of the SVG is applied to which rule. Since the SVG header is only instantiated once, it is not modeled in the same syntax as the following group elements (g). Despite containing some information about the size and a description, the SVG header is removed completely from the object as it does not contribute knowledge to the view model.
The first instance of a group element, g1, is removed by the second rule, as it also does not contribute any information to the view model. It is an element created by Figma to wrap everything inside one element. The second instance, g2, is identified as the wrapper for the following components. This is why it is the first element which is extracted as wrapper and added to the view model. Its identifier is often a hint towards the view, that is prototyped, and is therefore extracted as well. Moreover, it may contain some attached style, as for instance *clip-path*-keys. Those clip-paths are meant to restrict a region, where the paint shall be applied[2]. Although this feature is also implemented in CSS, it was removed because of its reference to resources inside the SVG. This reference though is the *path* and can not be mapped to a component and therefore the *clip-path*-properties were removed completely.

[2]https://developer.mozilla.org/en-US/docs/Web/SVG/Element/clipPath

The next element is another group element, but this time it is an array filled with objects. Since this part of object wraps the whole component tree, it is called *rootComponent* and is applied to the respective extractor. This component marks the root of the DOM, which has later children added to it. Referring back to the object diagram, the *rootComponent's* first element is often a rectangle, which is in case of the prototypical UI representing its outer bounds. Those rectangles have style information attached, which is specified by the use object in this case . The use object is tailored to duplicate nodes from the inside of a SVG document to some other place in the document[3]. Since, cross-referencing is not part of this thesis it was skipped but the style attributes of the use object can be used and were therefore extracted as style information and attached to the *rootComponent*.
In the example object diagram 5.2 there is also a first component to be seen, which leads to the first problem, discussed in the following. In general this component would be extracted by a specific rule tailored towards the requirements of this component.
The extraction of the *rootComponent* marks the end of the first phase of the extraction process as defined in figure 3.7.

Problems identified with structure

Copyright component inside of *rootComponent*

It was stated that the *rootComponent* is on top of the tree and the child components are following it. In the example the UI prototype 5.2 already indicates that it can not be generally admitted. The copyright component is inside the *rootComponent* although being located on the bottom of the application as visualized in figure 5.1. This issue is a big problem for the structure of the view model as it messes up the hierarchy that is met by all other components. Eventually a workaround for this problem has to be developed when mapping the components to W3C Components.

Problems identified with style

SVG's XML pointers

Many style properties inside the components point to resources as it is part of the XML design. Nevertheless, this resource can not be extracted as they are vectors that describe in detail how for instance a rectangle has to be drawn. Since this kind of drawing is not possible in component based rendering, the solution is mapping. Referring back to figure 5.2 some of the style properties can be used and are therefore extracted. To be precise for the example, the property of *fill* can be used later on, as it is part of CSS. On the contrary, *xlink:href* is another XML pointer to a resource and is therefore removed.

Mapping of fill property

The fill property is the most interesting style characteristic when mapping the SVG to the view model, because it is the only way that color properties are defined in the SVG. This means that for example a button has two fills, where one needs to be mapped to the

[3]https://developer.mozilla.org/en-US/docs/Web/SVG/Element/use

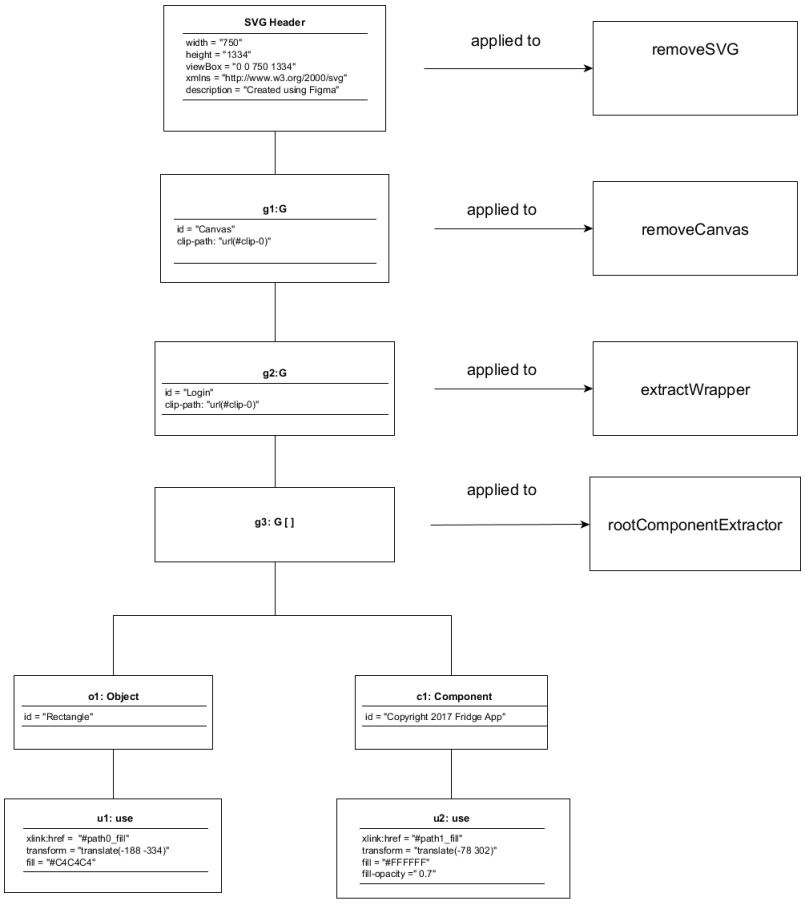


Figure 5.2: Object Diagram rawJSON

button's background color and the other fill needs to be mapped to its font color. The same problem arises when dealing with different type of components. To solve this problem two steps have to completed in order to guarantee a valid mapping:

1. Identification of the component type
   In order to establish the right mapping, the component type has to be identified correctly. Otherwise step two can not be executed.

2. Proper mapping of style properties.
   If the component type was identified correctly, the respective extraction rule will handle the mapping of the style properties from the SVG to the view model.

These style issues are a general challenge in the extraction process and this is why they were placed inside this section, although they are explicitly handled in the second phase of the extraction process. This placement is due to the structure of the following use case evaluation where every phase of the extraction process is explained in detail and many of

the extracted components and their respective problems with style, structure and content are presented one after another.

5.2.3 *Preprocessed JSON*

The *preprocessed JSON* marks the beginning of the second phase of the extraction process, which is the application of the specific rules. Figure 5.3 shows an object diagram of a form component taken from the FridgeApp login view. The component is introduced by an object (o2) with the identifier ”Group” and followed by a group array, wrapping the single parts of the component. Since the component is of type form it is obvious that the left part of the diagram has to be the form part. It is an object with object identifier ”Rectangle” followed by some style information. On the other side the label part is located, which is introduced by an object (o3) with the identifier ”Password”. This property is not only the identifier of the object but also the content of the form component. It can be mapped to the label that is located above the form as visualized in the login view figure 5.1. The label part has also some style information attached which can be extracted.

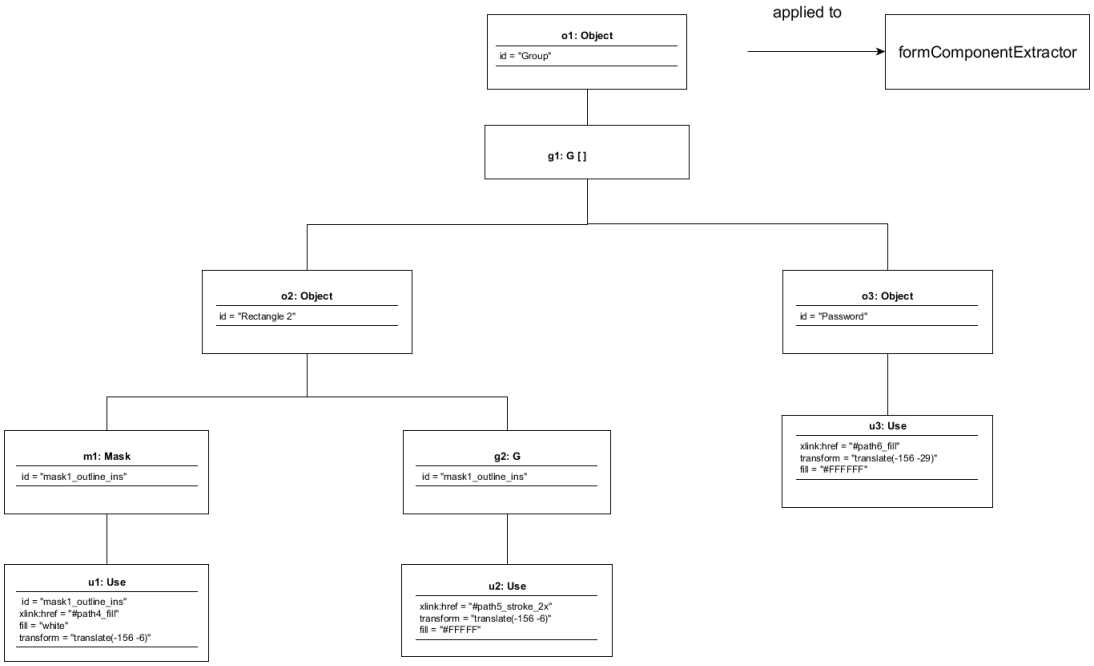


Figure 5.3: Object Diagram Form Component

Problems identified with structure

Header inside of the first form

As shown above, the extraction of a form component can be an easy match. A counter example is visualized in figure 5.4. The figure is very similar to figure 5.2. It clearly consists of a form part (o2) having an identifier ”Rectangle” with some style attached, as well as a label part. The issue arising is, that both *o3* and *o4* could be the label part of the form

component. There is no way of differentiating between those two objects. By looking at the FridgeApp login view it is obvious that *o4* is the label part of the component and *o3* is a kind of header for the following components. This issue is a clear limitation of the SVG's structure and in this example there is no clean way of matching the elements to its respective components without looking at the login view.

Problems identified with content

Identification of text inside the SVG

The problem identified with the content part of the view model is, that the SVG does not directly supply any content. The string *Phone Number* from the label of the form component for instance is saved in the identifier of the object as visualized in figure 5.4. This could be problematic when facing more complex user interfaces than the analyzed example.

Problems identified with style

Different style structures in form parts

Comparing the form parts of the two object diagrams, namely *o2* in 5.3 and *o2* in 5.4, there is a clear difference between the structure of their attached style parts. One is wrapping its style information inside of a use object, whereas the problematic form component structure in figure 5.4 is more complicated. It contains a group and mask object, which is the first reason it is more difficult to identify. Moreover, the mask and group objects wrap use objects, which themself have nearly identical style contents. The only difference are the various XML references (xlink:href), whereas the fill properties *"white"* and *#FFFFF* are identical. This problem leads to a higher complexity of writing a comprehensive rule to identify and extract form components, as they do not follow a general structure and can vary.

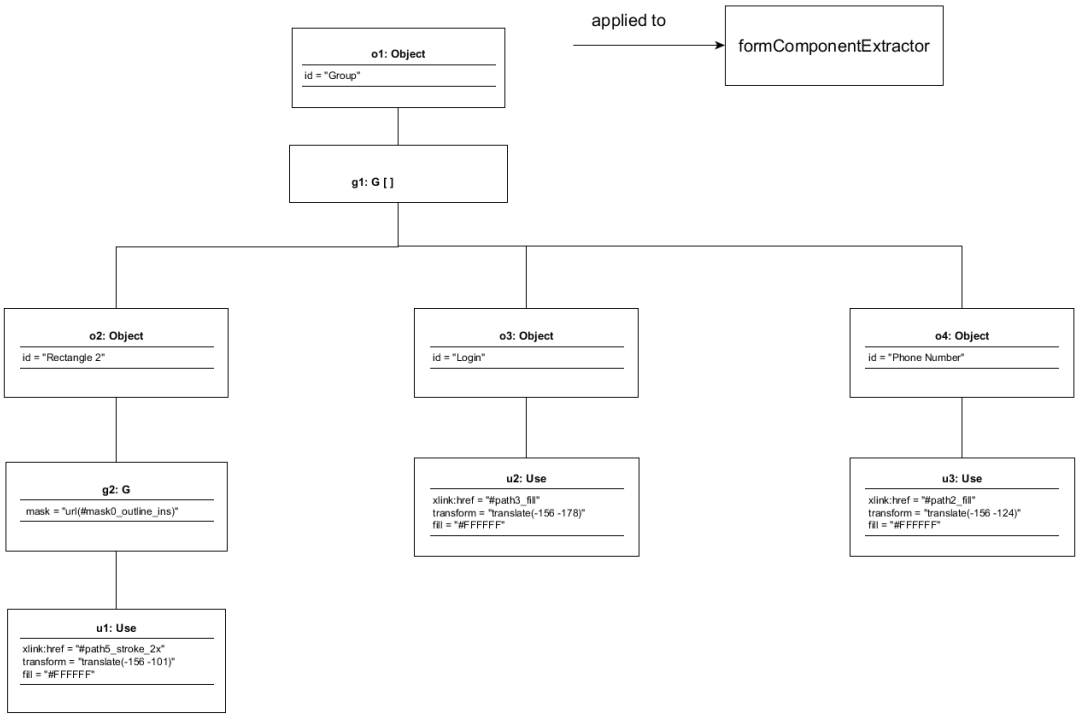


Figure 5.4: Object Diagram Form Component with Problematic Structure

5.2.4 View Model

The view model is the result of the first and second phase of the extraction process. It contains all information about the SVG that could be extracted.

View Model Header and Copyright Component

Figure 5.5 visualizes the origin of the view model.
The first element is the *rootComponent* which has an identical identifier and has its style and structure attached. The class name *div* is self-evident, as it is the wrapper of the component tree and this is one of its features in HTML documents. The style part has been extracted from the first rectangle of the SVG, which is responsible for the outer bounds of the user interface.
Following style and structure is an array of *rootComponent's* children, which are all components that could be extracted. In the case of the example the first component is a copyright component. As stated in 5.2.2 the copyright component is on top of the hierarchy, although it is located on the bottom of the view, which is obviously a problem. Nevertheless, the copyright component gets extracted smoothly with all its properties, namely structure, content and style. The combination of the structure and the text of content, the component could already be rendered in pure HTML, since the beginning of the text *&copy* references a copyright tag in HTML[4].

[4]https://wiki.selfhtml.org/wiki/Referenz:HTML/Zeichenreferenz

Since the copyright component has no children attached, the view model continues with the component tree.

```
{
    id: "rootComponent",
    structure: {
        className: "div"
    },
    style: {
        properties: {
            transform: "translate(-188 -334)",
            fill: "#C4C4C4"
        }
    },
    children: [
        {
            CopyrightComponent: {
                id: "CopyrightComponent",
                structure: {
                    className: "p"
                },
                content: {
                    text: "&copy; Copyright 2017 Fridge App"
                },
                style: {
                    properties: {
                        transform: "translate(-78 302)",
                        color: "#FFFFFF",
                        fill-opacity: "0.7"
                    }
                }
            }
        },
.....
```

Figure 5.5: View Model of Header and First Component

Extracted Form Component

Figure 5.6 visualizes the next child of the *rootComponent*. The component features its *formComponentWrapper*, which is responsible for wrapping the component's form and label part. This is why it is defined again as a *div*.
The first child is the label part. Its identifier is generated from the content and the term label to make its assignment easier. The structure is defined as *label* as it is HTML's way of defining a label for its respective form. Because of that a similar approach for this view model was chosen. The content part is the extracted identifier of the object from the *preprocessedJSON*.
The form component is described by a similar identifier which is the label string appended with the term form. As it is conceivable that a form component may exists without a label part, the form identifier in this case would be form plus a number, which will be incre-

mented each time a form is extracted. This process is exemplary shown in the flow chart of the extractor in section 4.2.2. The style part of the form is an easy extraction, skipping the useless XML pointer. Lastly, the form component is self-evidently of type form, since it is HTML's way of defining forms. The identifier of the wrapping component is defined by the label's content string appended with the term "FormWrapper". Through this naming convention it is self-evident which part is the wrapper the form or the label.

```
{
    formComponentWrapper: {
        id: "passwordFormWrapper",
        structure: {
            className: "div"
        },
         children: [
            {
                labelComponent: {
                    id: "passwordLabel",
                    style: {
                        properties: {
                            transform: "translate(-156 -29)",
                            color: "#FFFFFF"
                        }
                    },
                    structure: {
                        className: "label"
                    },
                    content: {
                        "text": "Password"
                    }
                }
            },
            {
                formComponent: {
                    id: "passwordForm",
                    style: {
                        properties: {
                            fill: "white",
                            transform: "translate(-156 -6)"
                        }
                    },
                    structure: {
                        className: "form"
                    }
                }
            }
        ]
    }
}
```

Figure 5.6: View Model of Form Component

Faulty Extracted Form Component

As stated in subsection 5.2.3 one problem identified with the structure was the header inside the form component. Figure 5.7 displays exactly the result of this problem. The first element in the object array, where the form component was extracted from, is the login header followed by the form part. There is no way of differentiating between the header and label part of the form component. This is why the result in figure 5.7 is faulty as the header is accidentally identified as the label part. Therefore, the wrong label part is extracted although the valid label component is present.

```
formComponentWrapper: {
    id: "loginFormWrapper",
    structure: {
        className: "div"
    },
    children: [
        {
            labelComponent: {
                id: "loginLabel",
                style: {
                    properties: {
                        transform: "translate(-156 -178)",
                        color: "#FFFFFF"
                    }
                },
                structure: {
                    className: "label"
                },
                content: {
                    text: "Login"
                }
            }
        },
        {
            formComponent: {
                id: "loginForm",
                style: {
                    properties: {
                        transform: "translate(-156 -101)",
                        fill: "#FFFFFF"
                    }
                },
                structure: {
                    className: "form"
                }
            }
        }
    ]
}
```

Figure 5.7: View Model of Form Component with Faulty Content

Extracted Button Component

The next element in the component hierarchy is identified as a button component. Figure 5.8 shows the object diagram of the mentioned button component as it is saved inside the component array. Similar to a form component, a button consist of two different elements. The first one, is an object with an ”Rectangle” identifier and an object with a label like identifier. Both elements have style information attached in form of an use object.
Figure 5.9 shows the extracted view model of a button component. The identifier of the object is incremented each time a button component could be successfully extracted to ensure globally unique identifiers.
The structure part can easily be mapped to a button in HTML. The description of the button is kept in the content part as well as the style part is attached in its respective object. The mapping of the colors has been solved in the following way:
The color property, which is responsible for the description of the button is extracted from the fill property of the label part. The background color on the other hand is mapped from the rectangle object‘s fill, as it is responsible for the color of the button itself. This mapping works as long as both label and button part can be identified successfully.

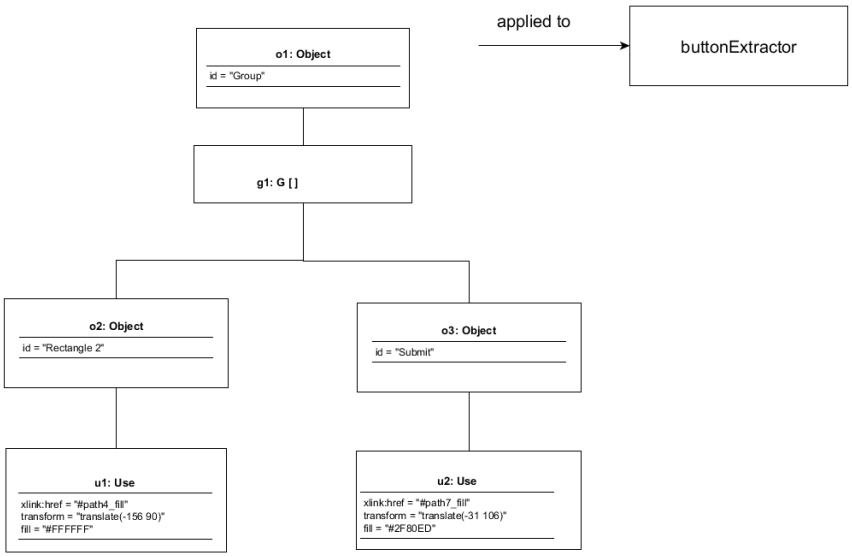


Figure 5.8: Object Diagram Button Component

Problems identified with structure

Identical structure as a form component

When comparing the object diagram of a button visualized in figure 5.8 and a form component 5.3 it is obvious to conclude that both components are very similar. Both have a kind of label part and rely on an object with a ”Rectangle” identifier. This is a huge problem, as the identification of components is a necessary step to extract them. The only difference is the identifier of the label part. Inside a button component the identifier of the label part is associated with a button description like for instance submit, cancel, abort, etc. This

problem is solved in the implementation part by matching the input component against an array of known button strings as named before. If an identifier that could be matched to a button component is present, the *buttonExtractor* will be invoked, otherwise the component will be passed to the *formComponentExtractor*.
As this workaround may work for the scope of this use case, it is self-evident that this could cause huge problems in more complex use cases and greatly effect the extraction quality. The array that is matched against the identifiers on the other hand can be extended easily, as it is defined as a variable.

```
{
    buttonComponent: {
        id: "button0",
        structure: {
            className: "button"
        },
        content: {
            text: "Submit"
        },
        style: {
            properties: {
                transform: "translate(-156 90)",
                color: "#2F80ED",
                background-color: "#FFFFFF"
            }
        }
    }
}
```

Figure 5.9: View Model of Button Component

Extracted HTML Link Component

The last elements that are inclosed in the component array are two HTML link components. Figure 5.10 shows an object diagram of the components inside the component array. Components of this kind can be identified by their use object array, which contains all attached style properties. Their respective identifiers can be mapped to their label, which is in the case of the view model is mapped to the content part.
Figure 5.11 visualizes the extracted view model of the object diagram. Since two components could be identified, the workaround implemented for this case was to combine both components in one array. This also leads to the first problem identified with structure, but is not content of this paragraph. Both components are very similar, as they only differ in their identifiers and marginally in their style properties. Their content and identifiers are derived from their object identifiers as well as their style information. The mapping to HTML links can be made, because of their very similar characteristics. Both should be click-able and should trigger and action, for instance direct the user to a dialog, where he can register.

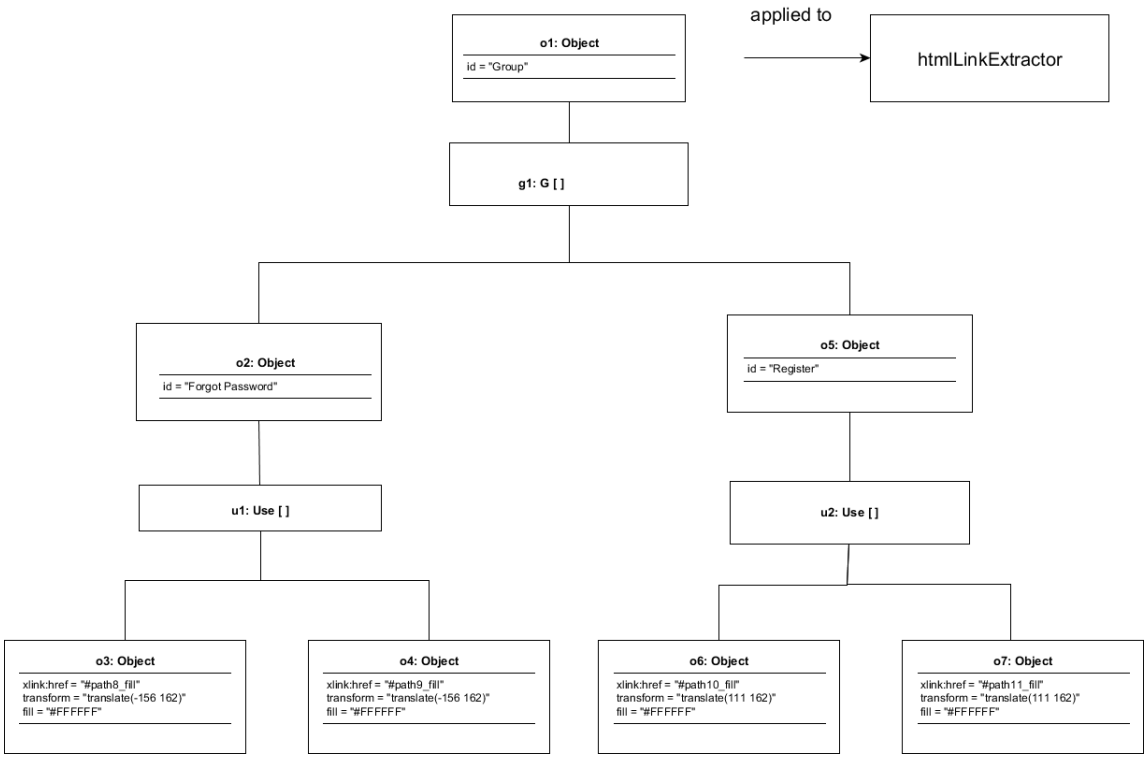


Figure 5.10: Object Diagram HTML Link Component

Problems identified with structure

Two elements inside one identified component

This is a severe problem and limitation since all other components are singled up inside of an group object array. HTML links differ, as they are wrapped inside one group object array and therefore the rule has to be implemented differently. Moreover, the mapping to the view model is a problem, as it does not conform to the the view model's structure.

Difficult component identification

The second problem is due to its different structure compared to other components. HTML links may be difficult to identify in other use cases. They are not containing much information as visualized in the object diagram 5.10, what makes it tough on the extraction tool to identify the valid kind of component or even extract usable information. The figure shows the two components (o2, o5) with their respective content information, which is in this case their identifier and their style properties encapsulated in their respective use array(u1, u2). The properties of the use array are in both cases very similar and will be explained in problems identified with style. Nevertheless, this slight difference in structure are a big challenge for the identification of the right component type.

Problems identified with style

Two use objects with nearly identical properties

This problem is not only related to style but also to the structure of the object. The only possibility to identify HTML link components is their respective use objects array. By analyzing those objects, it becomes self-evident that the information included in the use object arrays is very limited. Both elements inside the array contain nearly identical information, as only the XML pointer has an different identifier. All other properties such as fill and transform are identical. This may be a limitation but has to be checked by evaluating a second use case.

```
{
  htmlLinks: [
  {
    htmlLinkComponent: {
      id: 'Forgot Password',
      structure: {
        className: 'a'
      },
      content: {
        text: 'Forgot Password'
      },
      style: {
        properties: {
          transform: 'translate(-156 162)',
          color: '#FFFFFF'
        }
      }
    }
  },
  {
    htmlLinkComponent: {
      id: 'Register',
      structure: {
        className: 'a'
      },
      content: {
        text: 'Register'
      },
      style: {
        properties: {
          transform: 'translate(111 162)',
          color: '#FFFFFF'
        }
      }
    }
  }
  ];
}
```

Figure 5.11: View Model of HTML Link

5.2.5 DOM Abstraction

This subsection is contemplated to show examples of two typical components from the view model and how they were mapped to the DOM abstraction.

Form Component

Referring to figure 5.6 figure 5.12 visualizes how the extracted form component is mapped to the DOM abstraction. It has to be noted that only properties which were enriched by

the view model are modeled.
Since the component has children attached, the root of the component is mapped to a DOM element. This element is enriched with the component's identifier as tag and node name. Moreover, it has its class name attached, which is a div element. Two virtual nodes extend the root node, namely the label part (v1) and the form part (v2).
The first virtual node's tag name is again the identifier of the label part of the form component. This node has properties (p1) and children (c1) attached. The specific properties contain the class name, which is label and a style object which itself contains the respective style information that have been extracted to the view model. As stated before, the content part is mapped as Virtual Text (vt1) to the virtual node, which explains the extra children the right side of the tree.
The second virtual node on the right side of the tree represents the form part of the form component. It is very similar to the label part, unless having different style properties, a Virtual Text (vt2) and obviously a different class name.
Elements that have not been modeled but are enriched by the DOM framework are the parent nodes for instance. Every element in the DOM abstraction has a parent node attached. The label part (v1) and form part (v2) have their wrapper (d1) referenced in the parent node and the DOM element itself has the *rootComponent* attached as reference.

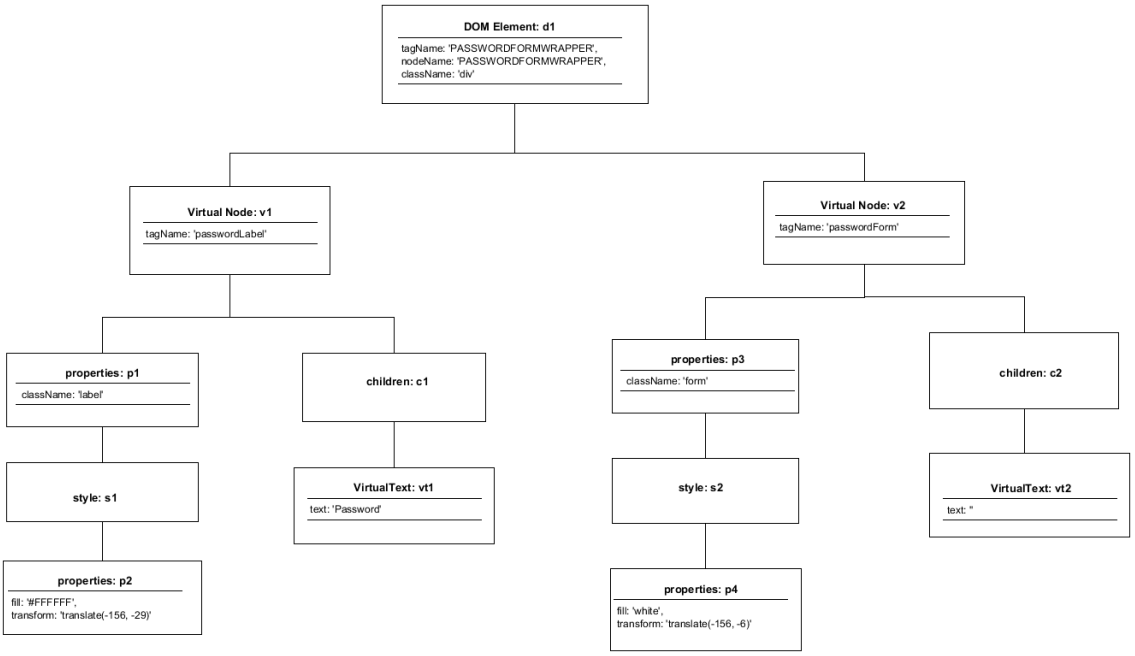


Figure 5.12: Object Diagram of DOM Abstraction Form Component

Button Component

The next example that is given is the mapping from a button component to a virtual node. Figure 5.13 illustrates the model of a mapped button component. The first thing that stands out is the simpler structure compared to the form component. This arises due to the fact that the button component has no additional children attached but its content in form of a

VText. The remaining properties are mapped similar to the form component.
The virtual node (v1) is identified by its unique tag name, which is again extracted from the button component's identifier. The attached child contains the content part instantiated as *VText* (vt1). On the left side of the tree the property section is located. It contains the button class name as well as some extracted style properties.
Obviously this virtual node has more properties in its specific implementation as only the most important ones are modeled. These properties are for example a reference to its parent node, hooks and namespaces.

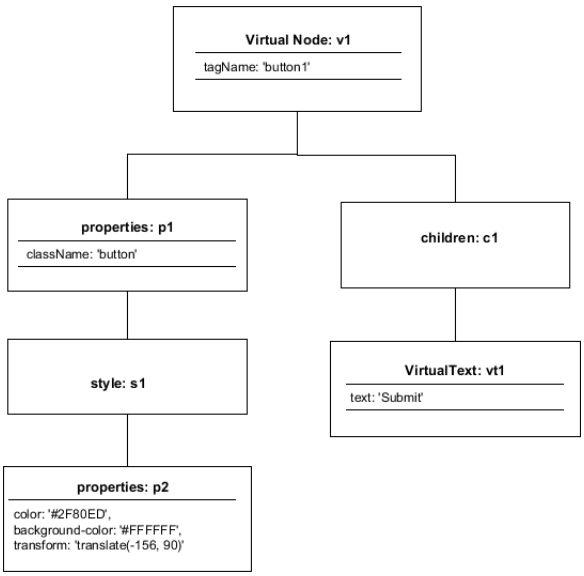


Figure 5.13: Object Diagram of DOM Abstraction Button Component

5.2.6 Generated Components

The following will presented two of the generated components. The respective sections will feature their HTML document as well as limitations, that came up during the evaluation.

Generated HTML Link Component

The first generated component presented is a HTML link component. In the FridgeApp login view the button is located beyond the submit button, visualized in figure 5.1. The generated HTML is represented by figure 5.14.
The HTML is generated on the basis of the *Virtual Node* skeleton, described in section 4.4.1. On top of the generated HTML there is as always the import of the Polymer element. The remaining part of the structure is also according to the template. The interesting part is the rendering part of the the link element (a). The identifier is chosen according to the component's identifier. The *href* property is hardcoded and referring to the root node inside the Polymer application, as there is obviously no reference defined in the UI prototype. This approach was chosen, since when there is no *href* defined inside the link element, the rendered element is not click-able and therefore is not visualized correctly. This property has obviously to be modified by the developer, when reusing the component.
The following script and Polymer part is again generated according to the template.

```
<link rel="import"
    href="../bower_components/polymer/polymer-element.html">
<dom-module id="my-forgotpassword2">
 <template>
  <style>
   a{
     transform: translate(-156, 162);
     color: #FFFFFF;
    }
  </style>
  <a id="ForgotPassword2"
  href="http://127.0.0.1:8081/rootcomponent">
  Forgot Password
  </a>
 </template>
 <script>
 class MyForgotpassword2 extends Polymer.Element {
  static get is() {
   return "my-forgotpassword2";
  }
 }
 window.customElements.define(MyForgotpassword2.is,
 MyForgotpassword2)
 </script>
</dom-module>
```

Figure 5.14: Generated HTML Link Component

Generated Form Component

Figure 5.15 visualizes the generated HTML document of a form component. It adapts to the DOM element skeleton defined in 3.3.4. Each of the parts, namley form and label part, have their own style definition inside the script tags. The difference in the element definition (beginning of the closing script tag) is that the *div* is wrapping label and form part. Inside the form tags an input element is defined. This is where the developer can adapt the input types. There are three different input types[5]:

- text
  Defines an one-line input field.
- radio
  Defines a radio button, which allows selecting one of many choices.
- submit
  Defines a submit button, which is self-evidently used to submit a form.

[5]https://www.w3schools.com/html/html_forms.asp

```
<link rel="import"
href="../bower_components/polymer/polymer-element.html">
<dom-module id="my-passwordformwrapper">
  <template>
    <style>
     label{
      transform: translate(-156, -29);
      color: #FFFFFF;
     }
     form{
      fill: white;
      transform: translate(-156, -6);
     }
    </style>
    <div id="passwordFormWrapper">
      <label id="passwordLabel">Password</label>
      <form id="passwordForm"> <input> </form>
    </div>
  </template>
  <script>
    class MyPasswordformwrapper extends Polymer.Element {
      static get is() {
        return 'my-passwordformwrapper';
      }
    }
    window.customElements.define(
    MyPasswordformwrapper.is,
    MyPasswordformwrapper
    );
  </script>
</dom-module>
```

Figure 5.15: Rendered Form Component

5.2.7 Use Case Evaluation Summary

In the following a summary of the evaluated use case will be given.

Extraction Process

The extraction process will be measured by its successfully extracted components and their respective properties. Table 5.1 shows the success of the extraction process. The total number of components from the *rawJSON* were taken as available components and were counted. The extraction quality of components has been divided into two rows, since the first step is the identification and the second the extraction. Obviously a component can

not be extracted if it has not been identified before. Nevertheless, it is an important quality criteria of the extraction process.
Properties are only listed once, as when a component has been identified there is an implicit extraction of their properties. Only those properties are counted, that can be used in a W3C Components context. This means that all XML related properties that can not be used in the view model are not counted for the total column.
The table shows that every component from the use case could be identified correctly. This obviously correlates with the fact, that the extraction tool has been developed by the means of this example and therefore is well suited for its structure and contents. The rate will likely greatly differ, when the extraction tool is applied to a different use case. Nevertheless, the rate is a good indicator for successful mapping to the view model.
It may be surprising that the rate of identified and successfully extracted components differ. This evolved from the problem described in section 5.2.4. The header inside the form component can only be identified by looking at the login view itself. There is no way of differentiating between the right label components just with the extraction tool, as they look exactly the same. This issue is responsible for the difference between successfully identified and extracted components.
The rate of successfully extracted properties is also indicating a decent success rate. The incorrect extracted properties are easy to explain, as they have been skipped for several reasons. For instance, two properties of the HTML link components have been skipped, as they are important to identify the components, but do not contribute any style information to the view model. This is because the style properties are identical as described in section 5.2.4. Moreover, two properties are missing due to the issue concerning the faulty identified form component. The last missing property has evolved from the different structure of form parts inside the form component.

	Correct	Incorrect	Missing	Total	Rate
Identified components	6	0	0	6	100 %
Extracted components	5	1	0	6	83 %
Extracted properties	19	0	5	26	73 %

Table 5.1: Evaluation of Extracted Components Login View

DOM Abstraction

The DOM abstraction part is not expected to cause problems as long as the extraction part was executed successfully. Every component that has been identified will be mapped to the virtual DOM with no exception. This is the reason why a evaluation of this part of the process is not necessary.

W3C Components Generation

Finally, the last step of the evaluation is the result of the component generation. Note that the extraction tool has been built around this use case and the result is most probably the best result this approach can produce. Figure 5.16 pictures a screenshot of the bundled component inside the Polymer project. All of the individual generated components

are bundled, imported into the HTML and then rendered. The result clearly pictures the limitations of the chosen approach and many of the problems, that have been identified during the implementation and have carried out through the whole process. In the following every single component will be considered individually naming all issues and their cause.

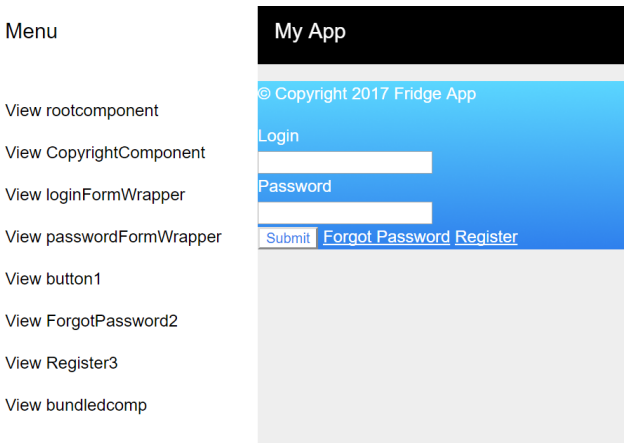


Figure 5.16: Screenshot Bundled Components FridgeApp

rootComponent and Background

The first thing that immediately pops up is the background of the *bundledComponent*. The background is computed by the linear gradient[6] which happens to follow the same rules in CSS and SVG. Obviously this makes the mapping easy and the visualization of the background is pretty good. The only problem is the design of the first phase of the extraction process, where the section that contains the linear gradient information was removed completely. By design the first phase is not extensible and therefore a workaround was implemented to ensure a proper background mapping.

Copyright Component

The next component in the hierarchy is the copyright component. The mapping of the component itself and its style worked good but obviously the placement of the component is false. This issue was already addressed in subsection 5.2.2 as the component is not placed at the expected position and therefore is also rendered at the wrong position inside the bundled component.

Form Components

The next two elements are the form components and there are several issues. Although nearly all style information from the SVG could be extracted to the respective components, there are still obvious issues with the style part. Firstly the rectangles do not have soft edges but hard ones. The reason for this is that the rectangles in the SVG are defined vectors. Those vectors draw the lines and therefore create the soft edges of the rectangles.

[6] https://developer.mozilla.org/en-US/docs/Web/CSS/linear-gradient#Gradient_with_multiple_color_stops

Sadly there is no way to map those vectors to the components and hence the form part does not have any soft edges.
Secondly the mapping of the rectangles' fills did not work properly. In the UI prototype the only the rectangles' borders are white as in the generated component the whole fill of the rectangle is white. This issue is again caused by the way rectangles are defined in the SVG.
Thirdly there is also a structural issue. The first form component has a label named "Login" which is self-evidently the wrong label. As described in subsection 5.2.2 the issue is the login string, which is of type header and placed inside the form component. As stated before there is no way of differentiating between the header and the label in this particular case. Since the first element inside the form component is the login header and its structure is identical to a label, the responsible extractor expects the header to be the label and therefore maps it to the label part of the form. The correct label is skipped, because there is no second label expected inside of a form component. This issue is not present inside of the second form component (Password) and therefore the mapping works for this component.

Submit Button

The next component in the figure is the submit button component. Since a button component is very similar to a form component and its rectangle part is identical to the form's rectangle, the same issue with the soft edges can be identified. Moreover, the size and dimension of the button are not even close to the button of the UI prototype. This issue is also caused by the vector definition of rectangles in SVG data structures. On the other hand the extraction of the remaining style part worked pretty well, since the background color and font color could be extracted and mapped successfully to the button component.

HTML Links

On the bottom of view, the last two components are the HTML link components. In the implementation parts there were already big problems with the structure of the links. They appear inside the SVG as one component, but need to be mapped to two single components for the rendering part. This workaround was implemented and may not work for other use cases, since the assumption that every HTML link is defined as single component may not be true generally. Nevertheless, by the means of the workaround the mapping to the rendered components inside the bundled component worked pretty well. The content information, which is basically the displayed text, and style properties could be extracted and rendered successfully. The only issue of the rendering part is the placement of the components. In the UI prototype the links were put beyond the submit button. In the resulting component the links are placed next to the submit button. This issue is caused by the lack of structural information of the SVG. During the implementation no properties could be found that justify the placement of the links and therefore are also not present in the rendered component.

Summary

Before the summary of the evaluation some assumptions that have been made during the implementation have to be emphasized. Firstly the extraction tool was designed around this use case and therefore no other UI prototype will yield a better result than the *FridgeAppLoginView*. Every assumption of how elements from a SVG can be mapped to components are based on the content and structure of this use case. Since this may not generally be true for other use cases the scope of the implemented extraction tool may be small but has to be evaluated by assessing another use case. Nevertheless even for this use case many workarounds had to be implemented which already question the generally applicability of the chosen approach. This applicability obviously has also to be evaluated by assessing another use case.
Moreover, not even all properties that could be extracted and were attached to the components can be rendered in browser such as the transform property that nearly every component has. The transform property defines a list of transform definitions that are applied to an element and the element's children[7]. As the property works with coordinates it obviously can not be rendered as a component and is skipped at DOM build time, although it was extracted and mapped successfully to its respective component. This issue exemplifies the loss of information when transforming a SVG file to a HTML document.
The evaluation can be summarized by stating that a mapping from SVG elements to web components is possible in general but many issues exist although the tool was designed around this use case. Those issues have to be solved by implementing workarounds which themselves narrow the general applicability scope of the extraction tool. The applicability to real world examples can already be questioned.

5.3 Stripe Checkout

The next use case that was analyzed is a view from an UI prototype that features forms for pay checkouts[8]. Those checkouts are for instance used when buying something online and then being redirect to the paying part. This use case was chosen after a long search for a well-suited example to elaborate the limitations of the implemented tool. Even if the evaluation will disclose huge limitations, it was exactly chosen for that reason. The main issue is that there is nothing like design guidelines and even if the prototypes are designed with the same tool, they greatly differ in their structure and their properties. Figure 5.17 shows the UI prototype. At first sight it looks very similar to the FridgeApp login view. There are forms and their respective labels and on the bottom a button to trigger the checkout. Nevertheless, its structure and properties greatly differ from those of the FridgeApp and will be discussed in the following.

5.3.1 SVG and rawJSON

The first step of the process is not causing problems. This is not surprising as it only parses the SVG to JSON. But looking at the structure of the *rawJSON* there are the first big

[7] https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/transform
[8] https://www.uplabs.com/posts/stripe-checkout-form-web-mobile-responsive

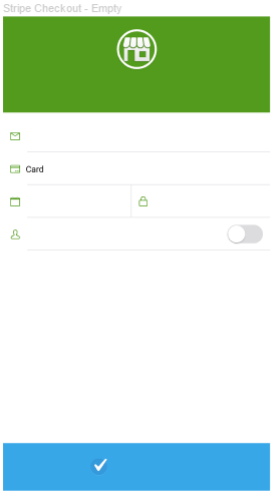


Figure 5.17: Screenshot Movie Detail View

problems arising. On basis of the FridgeApp the assumption was made that the structure of the header components is always identical. This is certainly not the case for this use case. Figure 5.18 shows the beginning of the passed object. As the first part of the extraction process is fixed, structural problems cause the extraction tool to fail.

Problems identified with structure

Different component identifiers

The first four objects are as expected (SVG, g1, g2, g3), but beginning with g4 are great and not expected difference in the structure. In the FridgeApp, components were introduced wrapped in an array with the identifier ”Group” and every single component was itself wrapped after a group identifier and inside of an array. The object diagram clearly shows that this is not the case for checkout view. This issue leads to failure of the extraction, before any extraction could be started. Of course a workaround for this problem could be implemented, but the idea of this thesis was to create a generic tool and a another workaround would greatly contradict this approach. This is why even if the extraction process is basically over at this point, the evaluation of the use case will continue, as it shows limitations of the whole approach which the attempt to generate components from SVG UI prototypes.

Problems identified with style

The problems that were identified with style properties are very similar to the ones of the FridgeApp. Those issues are generic problems when trying to transform SVG, which again is well-formed XML, to a data structure with lower abstraction level. This will always lead to loss of information, which is already described in section 5.2.2.

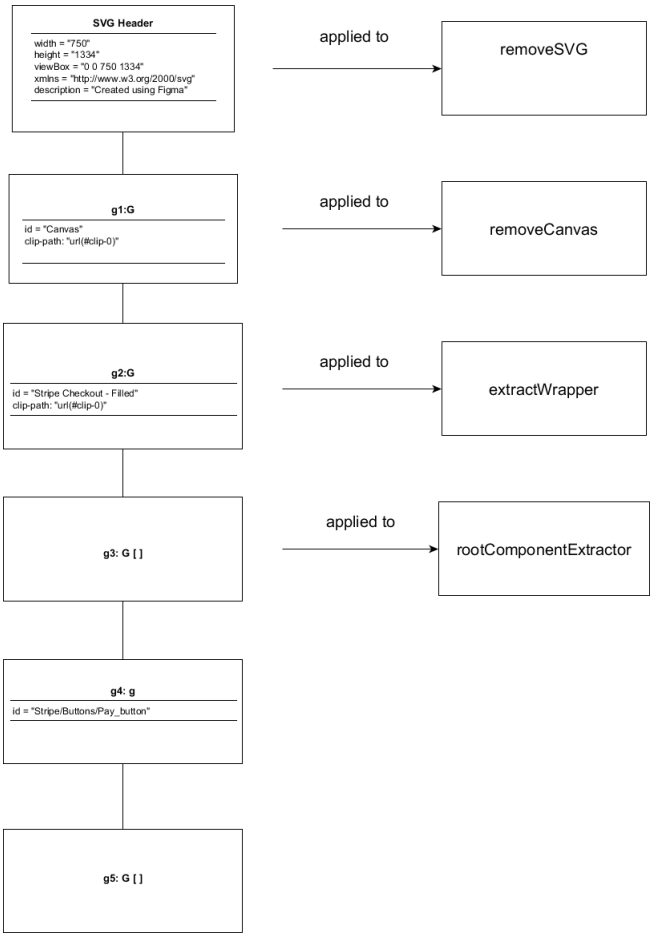


Figure 5.18: Object Diagram rawJson of Stripe Checkout

5.3.2 *preprocessedJSON* and View Model Header

The next step in the extraction process is the generation of the *preprocessedJSON*. This step is applied after the parsing and generates a *rootComponent* with its respective properties. Moreover this step creates an array with all components that are included in the SVG and attaches them as children to the *rootComponent*. Figure 5.19 shows the JavaScript object, after the generic rules have been applied. Although a component array was extracted successfully, a look at the first element of the *componentArray* will immediately rise big concerns if any information can be extracted.

```
{
    rootComponent: {
        id: "rootComponent",
        structure: {
            className: "div"
        }
    },
    componentArray: [
        {
            id: "Stripe/Buttons/Pay_button",
            g: [
                {
                    id: "base_button",
                    use: {
                        xlink:href: "#path0_fill",
                        transform: "translate(978 733)",
                        fill: "#66C226"
                    }
                },
                {
                    id: "Ic_check",
                    use: {
                        xlink:href: "#path1_fill",
                        fill: "url(#pattern0)",
                        transform: "translate(1154 755)"
                    }
                }
            ]
        },.....
    ]
}
```

Figure 5.19: preprocessedJSON of Stripe Checkout View

Problems identified with Structure

No Component Identifiers

Components inside the Stripe Checkout view are different than in the FridgeApp. Therefore, all of the rules will fail as the components need to be identified firstly and only if this condition evaluates to true, the extraction of the component can start. This means if the component can not be identified the extraction process will also fail.

Exemplary Button Component

Figure 5.19 shows the beginning of the *preprocessedJSON*. It shows the *rootComponent* and following this the component array. The first element in the array is representing a button component, which can be assumed by opening the UI prototype in Figma. This example greatly shows the problem that was stead above that components following different

structural patterns than in the FridgeApp example. The component is neither introduced by a group identifier, nor is the button component similar to button components of the FrigeApp example. A button component in the FridgeApp example always had an rectangle, which was representing the button itself, and a labeling part which was responsible for the content. This structure is not upheld by the Stripe Checkout example. Therefore, an extraction of the component is not possible by the means of the implemented extraction tool.

Exemplary Form Component

To elaborate even more limitations figure 5.20 shows what was manually identified to be a form component in the Stripe Checkout example. A form component was assumed to have at least a form part and optional a label part. The form was identified by being a rectangle with respective style properties and label by an identifier which was extracted as text and also some style properties. Those two elements were wrapped inside of an array, which identified itself as a component. Not one of these identifiers is present in the Stripe Checkout view. There is clearly another structure. Obliviously the object with identifier label is contemplated to be the label part and also the icon is pretty clear at first sight. Of course it would have been possible to implement a button component identifier that follows the corresponding rules of the Stripe Checkout view but again this would greatly decrease the generic approach the extraction tool was designed to. During design and implementation there had been no clue that the structure would differ by this amount.

5.3.3 View Model

The issued problems result in a nearly empty view model visualized in figure 5.21. The only component that was extracted successfully was the *rootComponent*. But not even this is a big accomplishment, as the rules for *rootComponents* were defined light, since the component is obligatory to start any type of extraction or even rendering. All of the stated problems lead to an empty children array, which makes the view model completely useless.

5.3.4 DOM Abstraction and Generated Components

Since the view model is useless neither a mapping to the virtual DOM can be realized nor components can be generated. Therefore, a further analysis will not yield anymore findings.

5.3.5 Use Case Evaluation Summary

The analysis of this use case greatly shows the limitation of the implemented extraction tool as well as the chosen approach. If the tool works similar to the FridgeApp use case and is further developed it may lead to positive result in the process of prototype driven development. As this is not the case for different use cases, the chosen approach is not comprehensive enough. The easiest way and probably most useful way would be to create design guidelines which represent how which type of element should be modeled. This would greatly simplify the identification of components and hence the extraction.

```
{
    id: "Stripe/Inputs/Input_single",
    g: [
        {
            id: "h_rule",
            use: {
                xlink:href: "#path2_fill",
                transform: "translate(1012 461)",
                fill: "#E5E5E8"
            }
        },
        {
            id: "Stripe/Inputs/Input_text_placeholder",
            $t: ""
        },
        {
            id: "Label",
            use: {
                xlink:href: "#path6_fill",
                transform: "translate(1010 430)"
            }
        },
        {
            id: "Stripe/Icons/UI/Email",
            g: {
                id: "ic_location",
                use: {
                    xlink:href: "#path7_fill",
                    transform: "translate(989 431)",
                    fill: "#559A28"
                }
            }
        }
    ]
}
```

Figure 5.20: Form Component of Stripe Checkout View

```
{
    id: "rootComponent",
    structure: {
        className: "div"
    },
    children: []
}
```

Figure 5.21: View Model of Stripe Checkout View

5.4 Fridge App Add Item View

To emphasize the applicability of the extraction tool to SVGs with an expected structure one more view of the FrigeApp use case will be evaluated to a minor extend.

5.4.1 Fridge App Add Item View

Figure 5.22 displays the screenshot of the UI prototype. It features a view to add items to the fridge. Therefore, it consists of a header and three forms. One to type determine the item's name, the second one for quantity and the third one for the quantity's unit. On the bottom a submit button is located, to save the typed in information. While extracting and rendering the components from the SVG no adjustments had been made to the extraction tool but adding one rule, namely the header extraction. The result directly yield the power of the extraction, when the structure is as expected.

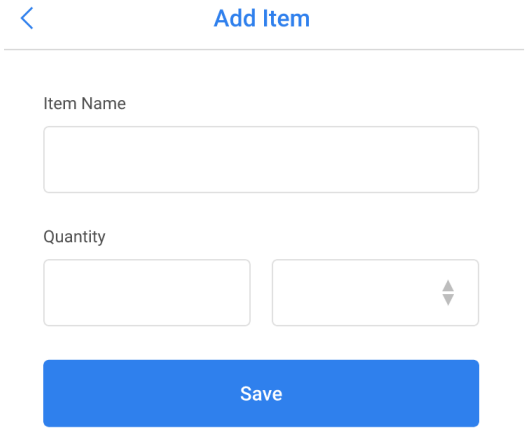


Figure 5.22: Screenshot Add Item View FridgeApp

5.4.2 *preprocessed JSON*

Exemplary for the *preprocessed JSON* figure 5.23 displays the first component that has a different type than any component identified in the login view. Its type is header and it is located at the very top of the UI prototype. It features a rectangle to define the structure of the underlying header, a label part and the icon part. The implementation of the rule currently features only the extraction of the label part, since this is the most important part of the header. As the rules are designed to be extendable it would be easy to add the missing parts to the extraction process and make it available for the rendering part.

5.4.3 View Model

In this section one component of the view model is shown to emphasize the quality of the extraction and the extensibilty of the specific rules.

```
{
    id: "Header",
    g: [
        {
            id: "Rectangle 3",
            filter: "url(#filter0_d)",
            use: {
                xlink:href: "#path0_fill",
                transform: "translate(267 1276)",
                fill: "#FFFFFF"
            }
        },
        {
            id: "angle-left",
            g: {
                id: "Vector",
                use: {
                    xlink:href: "#path1_fill",
                    transform: "translate(282.656 1290)",
                    fill: "#2F80ED"
                }
            }
        },
        {
            id: "Add Item",
            use: {
                xlink:href: "#path2_fill",
                transform: "translate(420 1288)",
                fill: "#2F80ED"
            }
        }
    ]
}
```

Figure 5.23: Header Component Add Item View

Header Component

Figure 5.24 visualized the view model of the header component. As stated above only the extraction of the label part is implemented and therefore is the only part that is extracted. The component is successfully identified and moreover all properties are mapped correctly. This will perfectly work with the process that follows, namley mapping to virtual DOM as well as the rendering and generating part.

```
headerComponent: {
    id: "header0",
    style: {
        transform: "translate(420 1288)",
        color: "#2F80ED"
    },
    content: {
        text: "Add Item"
    },
    structure: {
        "className": "header"
    }
}
```

Figure 5.24: View Model Header Component of Add Item View

Bundled Component

Figure 5.25 displays the screenshot of the bundled and rendered component. Note that the background has been hidden as the white background is not easy to differentiate from the native background of the application. As the screenshot clearly shows, the core part of the components could be successfully extracted and rendered inside the application.

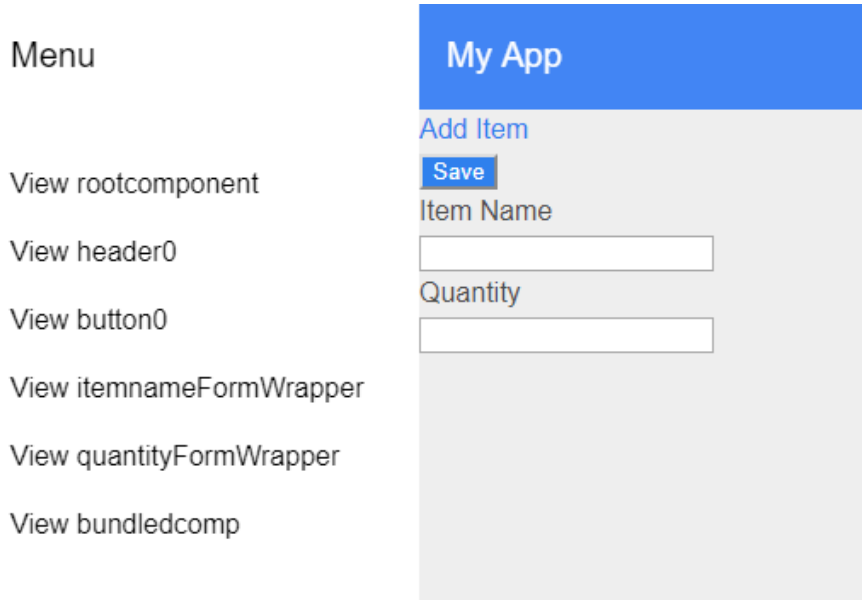


Figure 5.25: Screenshot Add Item View FridgeApp

5.4.4 Use Case Evaluation Summary

Table 5.2 displays the overview of identified and extracted components as well as extracted properties. The missing component is the display of the unit button, as the form component extractor only expects one form inside of a component array and everything else is ignored. To summarize the findings, it can be stated that the extraction tool does work to a satisfying extend when an expected structure is imposed to the input SVG.

	Correct	Incorrect	Missing	Total	Rate
Identified components	4	0	1	5	80 %
Extracted components	4	0	1	5	80 %
Extracted properties	10	0	5	15	67 %

Table 5.2: Evaluation of Extracted Components Add Item

5.5 Analysis of Evaluation

The evaluation showed that the quality of the model extraction by the implemented tool is not sufficient enough to be applied to real world use cases. The quality differs a lot between the use case, that the tool was built around and a different use case. As this may not to be too surprising, it is still notable that the model extraction could work, when the prototyping tool would follow predefined rules or some kind of annotations. By doing so, the complexity of the extraction tool would be greatly decreased, since it has only to extract the components and is not challenged by also identifying the components. Nevertheless, the extraction tool completely failed with the second use case because of the unexpected structure, so its applicability to real world use cases is not given.
In addition, the quality of the generated components can be questioned as well, since they are not very complicated in case of the evaluated examples. It is impossible to formulate an assumption on how the quality of the components would respond to an increasing complexity of the use case, but taken all findings into account it can be highly doubted that the quality of the extracted components would increase.
To conclude the evaluation taking all evaluated use cases into account there are some major improvements to be made, when further developing the implemented tool:

1. Annotate or identify components before passing then to their respective rule.
   This would decrease each rule's complexity and an assumption of the extracted components quality could be taken much earlier.

2. Prototyping tools have to be extended or developed to increase support for component extraction and generation.
   This suggestion would greatly contribute to the first item, as it would make components identification more trivial and reliable.

3. Enable extensibility in appropriate components.
   This suggestion refers to the templates for component generation. It would be beneficial, if components that are extendable, could easier be enhanced by the developer.

An example would be a button component that receives its on-click function as properties. This would probably increase the usage by developers, as they only need to define the HTML import, render the element at the desired position and them pass the function as a property to the component.

4. Increase extensibility of the first phase of the extraction process.
   In contrary to the second phase of the extraction process, the first phase is hardly extensible at all. The assumptions made during that phase were important for the following process, but as a result of evaluating the second use case were not generally applicable. Therefore, the first phase which is basically a preprocessor needs to be redesigned according to the findings of the evaluation.

6 Conclusion and Further Work

It was successfully shown that it is possible to map elements from an UI prototype to W3C Components. The defined view model clearly has its own limitations, since it only allows style, content and structure properties. Nevertheless, it was sufficient to generate and render web components based on the view model. On contrary it was conducted that assuming certain structures is not sufficient to cover a variety of use cases. This issue could be dealt with by introducing design guidelines or maybe to annotate the components inside the prototyping tool. By doing so, the modification would greatly increase the extraction coverage of the tool and therefore increase the quality of the generated components. Since it is unrealistic to assume, that the introduced prototyping tools tailor their output to simplify component identification and generation the usability and future of the chosen approach has to be questioned. Not only because of the identified limitations but on the basis of different approaches. Those different approaches often include machine learning such as the pix2code project[1]. The tool generates code on the basis of a graphical user interface's screenshot and is far more comprehensive than the model based approach that was used in this thesis, although the machine learning approach also relies on some kind of model.
The identification of components turned out to be far more complex than anticipated. The prototypical implementation of the view model extractor was greatly impacted by that issue. To avoid this and because of the lack of documentation of how the prototyping tools export crafted user interfaces, many structural characteristics of the first use case had to be taken as general assumptions for other use cases. As it turned out, this was not the case and lead to failure of the tool. Nevertheless, it was successfully shown that when UI prototypes follow certain structural rules and components can be identified correctly, the generation of the components happened to be less complex. In addition, although the analyzed prototyping tools are used by designers because of their similar component-based structure compared to the current major front-end frameworks (Vue.js, React.js, Angular.js), they could not fulfill the expectations of predictable SVG exports. This is still the biggest issue when dealing with this kind of model driven approach.
All in all it can be stated the chosen architecture was well-suited for the kind of process. The limitations of the first phase of the extraction lead to redesigning this step according to passed SVG's structure. To accomplish this, a deep analysis and evaluation of how SVGs are exported from prototyping tools has to be conducted and modeled. This kind of model would greatly impact the development of tools, that take a similar approach than the one presented in this thesis.

[1] https://github.com/tonybeltramelli/pix2code

7 Appendix

7.1 FridgeApp Login View: Simplified Handcrafted View Model used for the Evaluation

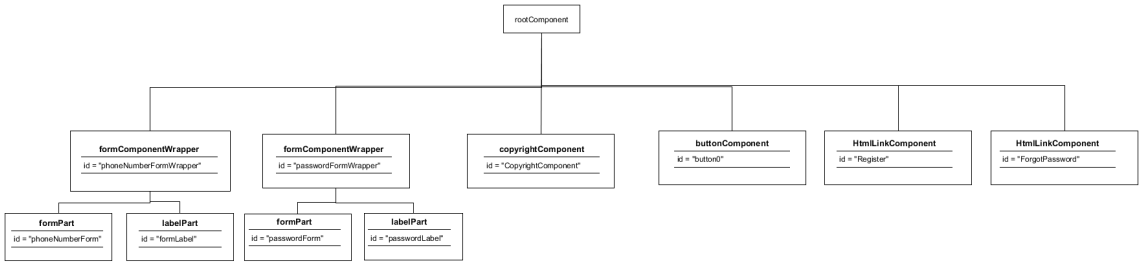


Figure 7.1: Hand Crafted Overview View Model of Login View

7.2 FridgeApp Add Item View: Simplified Handcrafted View Model used for the Evaluation

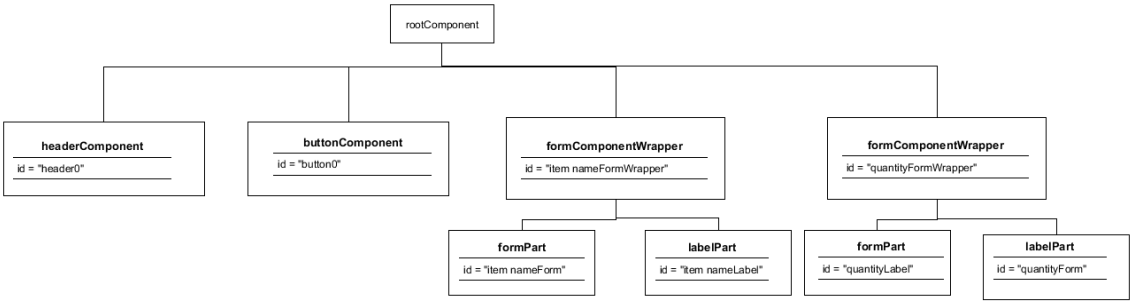


Figure 7.2: Hand Crafted Overview View Model of Add Item View

List of Figures

List of Tables

Bibliography

[Aho+14] P. Aho et al. "Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 2014, pp. 343–348. DOI: 10.1109/ICSTW.2014.39.

[AMS06] Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi, eds. *Extreme Programming and Agile Processes in Software Engineering: 7th International Conference, XP 2006, Oulu, Finland, June 17-22, 2006. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-35095-8.

[CC90] E. J. Chikofsky and J. H. Cross. "Reverse engineering and design recovery: A taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17. ISSN: 0740-7459.

[CKV07] Adrien Coyette, Suzanne Kieffer, and Jean Vanderdonckt. "Multi-fidelity Prototyping of User Interfaces". In: *Human-Computer Interaction – INTERACT 2007: 11th IFIP TC 13 International Conference, Rio de Janeiro, Brazil, September 10-14, 2007, Proceedings, Part I*. Ed. by Cécilia Baranauskas et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 150–164. ISBN: 978-3-540-74796-3.

[HC10] Alan Hevner and Samir Chatterjee. "Design Science Research in Information Systems". In: *Design Research in Information Systems: Theory and Practice*. Boston, MA: Springer US, 2010, pp. 9–22. ISBN: 978-1-4419-5653-8.

[Jac09] Julie A. Jacko, ed. *Human-Computer Interaction. New Trends: 13th International Conference, HCI International 2009, San Diego, CA, USA, July 19-24, 2009, Proceedings, Part I*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-02574-7.

[LSM13] Stefania Leone, Alexandre de Spindler, and Dennis McLeod. "Model-Driven Composition of Information Systems from Shared Components and Connectors". In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences: Confederated International Conferences: CoopIS, DOA-Trusted Cloud, and ODBASE 2013, Graz, Austria, September 9-13, 2013. Proceedings*. Ed. by Robert Meersman et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 204–221. ISBN: 978-3-642-41030-7.

[mob16a] mobgen. *High-fidelity prototyping: What, When, Why and How?* 2016. URL: https://mobgen.com/high-fidelity-prototyping/ (Retrieved June 6, 2017).

[mob16b] mobgen. *Low-fi prototyping: What, Why and How?* 2016. URL: https://mobgen.com/low-fi-prototyping/ (Retrieved June 6, 2017).

[MW17] Aaron Marcus and Wentao Wang, eds. *Design, User Experience, and Usability: Theory, Methodology, and Management: 6th International Conference, DUXU 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part I*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-58634-2.

[OAS08] OASIS. *User Interface Markup Language (UIML) Version 4.0*. 2008. URL: https://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf (Retrieved Aug. 3, 2017).

[Dyl17] Dylan Field. *Introducing Figma's Integration with Framer*. blog.figma.com, 2017. URL: https://blog.figma.com/introducing-figmas-integration-with-framer-c69a747aeee2 (Retrieved July 16, 2017).

[Eri13a] Eric Bidelman. *HTML Imports*. 2013. URL: https://www.html5rocks.com/en/tutorials/webcomponents/imports/ (Retrieved July 26, 2017).

[Eri13b] Eric Bidelman. *HTML's New Template Tag*. 2013. URL: https://www.html5rocks.com/en/tutorials/webcomponents/template/ (Retrieved July 26, 2017).

[Eri17a] Eric Bidelman. *Custom Elements v1: Reusable Web Components*. 2017. URL: https://developers.google.com/web/fundamentals/getting-started/primers/customelements (Retrieved July 26, 2017).

[Eri17b] Eric Bidelman. *Shadow DOM v1: Self-Contained Web Components*. 2017. URL: https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom (Retrieved July 26, 2017).

[Isa16] Isaac Lyman. *Developer-driven development*. 2016. URL: https://hackernoon.com/development-driven-development-75c01b2afca1 (Retrieved June 6, 2017).

[Jea13] Jean-Marc Denis. *Discovering Sketch*. 2013. URL: https://medium.com/sketch-app/discovering-sketch-25545f6cb161 (Retrieved July 26, 2017).

[Joh12] John Mueller. *10 Reasons Development Teams Don't Communicate*. 2012. URL: https://blog.smartbear.com/management/10-reasons-development-teams-dont-communicate/ (Retrieved June 6, 2017).