Lappeenranta University of Technology

School of Engineering Science

Master's Degree Programme in Computer Science

Rafael A. Aguilar Umaña

Modeling Features of Standard Web Components

Examiners: Professor Ajantha Dahanayake
MSc Jiri Musto

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Master's Degree Programme in Computer Science

Rafael A. Aguilar Umaña

Modeling Features of Standard Web Components

2018

80 pages, 32 figures, 9 tables and 0 appendices

Examiners: Professor Ajantha Dahanayake
MSc Jiri Musto

Keywords: Modeling, Web Components, Front-End, Client Side, HTML, ECMAScript.

World Wide Web Consortium (W3C) is developing a set of standards for front-end web development since 2014, known as Web Components. The additional functionality addresses many of the problems of an increasingly complex font-end, creating necessity of new modeling tools in this field. Existing efforts in terms of modeling web such as WebML, and more recently Interaction Flow Modeling Language (IFML) are capable of reflecting significant aspects of web development. However, Web Components introduce features that none of the existing initiatives contemplate by default. This work aims to contribute by proposing an extension of IFML that is capable of describing structural aspects of web components standards that include: Shadow DOM, HTML Custom Elements and HTML Templates.

ACKNOWLEDGEMENTS

The present work would not be possible without the wise guidance of Professor Ajantha, to whom I express my most sincere gratitude and my immense admiration for such keen researcher eye, her work ethics and professionalism. Thanks to Julio del Valle, Gloriana Zamora, Carlos Naranjo, and Michael Rojas whose vast experience in Front-End Engineering provided this thesis with valuable insights. Finally, I cannot imagine reaching this point without the extraordinary support of my family in Costa Rica, and of a group of amazing people who made of Lappeenranta my second home: Amila, Alejandro, Abhishek, Saeid, Suomo, Sohail, Olya, Kees, Franz, Khaled, Sashas, and of course Dani.

Table of Contents

Figures 7

Tables 8

1 Introduction 9

1.1 Objectives 10

1.2 Research Questions 10

2 Research Methods 11

2.1 Literature Review 11

2.2 Interviews 11

2.3 Limitations of the Research 12

3 Modeling in Software Engineering 13

3.1 Purpose of modeling 13

3.2 Defining the scope of Model-Driven Engineering 14

3.2.1 Definition of Model-Driven Engineering (MDE/MDSE) 15

3.2.2 Definition of Model-Driven Development (MDD) 15

3.2.3 Definition of Model-Driven Architecture (MDA) 15

3.2.4 Definition of Agile Model-Driven Development (AMDD) 15

3.2.5 Model-Driven Software Engineering 16

3.2.6 Transformations 17

3.2.7 Agile and Model-Driven Development 18

3.3 Challenges of Modeling 19

4 Front-end Web Development 21

4.1 Background 21

4.2 Web Components 22

4.2.1 Custom Elements 22

4.2.2 Shadow DOM 25

4.2.3 HTML Templates 27

4.2.4 HTML Imports 27

4.2.5 Disadvantages 28

4.3 Front-End Frameworks 30

4.3.1 Advantages 30

4.3.2 Disadvantages 31

4.4 Atomic Design 31

5 Existing Modeling Languages for Web 32

5.1 Web Application Extension (WAE) for UML 32

5.2 WebML 32

5.3 Interaction Flow Modeling Language 33

6 Industry Practices (Interviews) 34

6.1 About interviewees 34

6.2 Techniques and Technologies 34

6.3 Challenges when Developing Front-End Reusable Components 35

6.4 Role of Modeling in Front-End 36

6.5 Viability of Usage of Models in the Industry 37

6.6 Modeling and Agile 38

7 Interaction Flow Modeling Language (IFML) 39

7.1 Scope of IFML 39

7.2 Principles 39

7.3 Key features 41

7.3.1 Modularization 41

7.3.2 Extensibility 41

7.3.3 Personalization 42

7.4 Artifacts 43

7.5 Visual Syntax 44

7.5.1 Concepts 44

7.5.2 Standard IFML extensions 46

7.6 Challenges of Modeling Front-End with IFML 48

7.6.1 Implementability and support 48

7.6.2 Graphic design aspects 48

8 Extending IFML with Web Components Features 49

8.1 Justification 49

8.2 Extending IFML to Model Web Components 49

8.2.1 Custom Elements 50

8.2.2 Shadow DOM 50

8.2.3 HTML Templates 52

8.2.4 HTML Imports 52

9 Use Case: DayPlanner Application 52

9.1 DayPlanner Description 53

9.2 Application Design and Wireframes 53

9.3 Modeling Application with IFML **58**
9.3.1 Refactoring and Module Definitions 62
9.4 Transformation to code **65**
9.4.1 From Specific to General 65
9.4.2 Polyfills 67
9.4.3 Source Code and Live Application 67
9.5 Use Case Results **68**
9.5.1 Benefits 68
9.5.2 Drawbacks and Challenges 69
9.5.3 General considerations 69
9.6 Limitations of the Use Case **69**
10 Discussion **71**
10.1 Recommendations for Future Research **73**
10.2 Conclusions **74**
11 References **77**

Figures

Figure 1: Classification of Modeling Purposes 14
Figure 2: Model-Driven Engineering approaches 16
Figure 3: Transformation concepts 18
Figure 4: Transformation between languages 18
Figure 5: IFML principles 40
Figure 6: Examples of Standard Extensions in IFML 42
Figure 7: IFML Visual Syntax scope(IFML Specification, 2015) 43
Figure 8: ViewComponent representing a Custom Element 50
Figure 9: Landmark and Shadow ViewContainer 51
Figure 10: Shadow ViewContainer with nested elements 51
Figure 11: Add New Goal 53
Figure 12: Initial state 53
Figure 14: Add Another Goal 54
Figure 13: Add Tasks 54
Figure 15: Hover State 55
Figure 16: DayPlanner Mark as Done 56
Figure 17: DayPlanner Mark as Not Done 57
Figure 18: DayPlanner Delete Confirmation Modal 57
Figure 19: Add Button ViewContainer 58
Figure 20: Add Task Action Flow 58
Figure 21: Modeling Add New Task Action 59
Figure 22: Two Hover States 59
Figure 23: Modeling Task Hover States 60
Figure 24: Modeling Task 60
Figure 25 Modeling Completion Ratio 60
Figure 26: Application in IFML, Iteration 1 61
Figure 27: TaskComponent Module Definition 62
Figure 28: AddTaskAction Module Definition 63
Figure 29: Application in IFML, Iteration 2 64
Figure 30: Creation of AddComponent 65
Figure 31: Implementing AddIcon Template 66
Figure 32: Creating Task Component 67

Tables

Table 1: Example of Autonomous Custom Element 24
Table 2: Example of Customized Built-in Element 24
Table 3: Usage of Shadow DOM 26
Table 4: Challenges of Front-End Reusable Components 35
Table 5: Usage of Models by Interviewees 36
Table 6: Basic IFML Concepts 44
Table 7: View Container Types 45
Table 8: Examples of Standard View Component extensions 47
Table 9: Mapping Web Components to IFML 49

1 Introduction

This thesis proposes a way to model features of Web Components. Web Components is a terminology that comprises a set of standards by W3C. They bring component-based software engineering to client-side web development (commonly known as front-end), by implementing “a single, canonical API for declaring and consuming reusable components” (House, 2015).

Existing efforts in web modeling such as WebML, or Interaction Flow Modeling Language (IFML) are important leaps to model the front-end of web applications (Marco Brambilla, Mauri and Umuhoza, 2014). However, they lack the support for Web Components.

The present work aims to help web developers to model reusable web components by providing a simple extension to IFML. This is achieved by analyzing existing modeling efforts and identifying key aspects of Web Components, to finally produce an extension to IFML that maps the model with the specific implementation aspects of Web Components. This document includes a use case application, called “*DayPlanner*”, that employs the extensions proposed, in order to validate them by documenting pros and cons.

The increase of internet adoption worldwide (ITU, 2017), and the introduction of standards such as HTML5 (W3C, 2017), and ECMAScript, provide a wider range of tools, that also increase the complexity in front-end web development. For instance, ECMAScript’s originally was a simple scripting standard, and is now a “fully-featured general-purpose programming language” (ECMA, 2018). Another example is NPM repository ecosystem (for JavaScript), a platform widely used in client-side development, is growing at a much higher rate in comparison to the most popular back-end repositories such as Maven, for java, or Nuget, for .Net (ModuleCounts, 2018).

This shift is allowing front-end development to grow in number of features and complexity, which also requires more skills. This reflecting in the labour market, where the average position as web developer is paying $88,488 yearly (Glassdoor, 2018b) in the United States, while front-end web developers is slightly higher at $88,680 a year in average (Glassdoor, 2018a).

Despite the growth in importance and complexity of client-side development, there is an important gap in front-end web development in academia, and a complete absence of research when it comes to modeling Web Components. Based on the level of sophistication and growth in front-end software engineering, this author considers highly relevant the usage of models to describe Web Components in order to resolve complex problems and effectively communicate front-end architectures.

In addition, this work includes experiences of expert front-end web developers to provide insights regarding industry practices. This additional context is essential to analyze how modeling can play a role in improving the existing practices.

1.1 Objectives

- Analyze existing efforts to model web applications.
- Identify challenges of modeling Web Components.
- Extend IFML to include Web Components features.

1.2 Research Questions

- What are the outcomes of previous research concerning web modeling?
- What are the problems of existing modeling languages in front-end development?
- How to extend which describes the structure of web components?

2 Research Methods

2.1 Literature Review

A critical literature review is necessary in order to obtain the existing state of knowledge in the topics of modeling software, and the current efforts in terms of modeling of web elements, as well as insights in web components in general.

Based on the information obtained in the Literature Review, the research will propose a modeling language that reflects structural aspects of web components.

2.2 Interviews

Target Interviewees: Professionals with at least 4 years' experience in the field of front-end web development.

Objective:

- Get qualitative information about industry practices. For instance: needs, common practices, best practices, etc.

The participants will be contacted via e-mail, and the interviews will be conducted through e-mail or voice call. They will be ***semi-structured*** interviews, which are widely used in qualitative research. The semi-structured nature will provide prepared questions as a guidance, while being able to adapt or add new questions in order to contemplate circumstances related to the organizations the subjects are currently working on.

The interview guide will be based on the findings of the literature review. This will establish a framework of discussion where the information obtained by analysing and comparing against the existing work.

The interviews play a key role in the modeling proposal, since they context to the applicability of IFML (and the extension proposed) in real world scenarios.

2.3 Limitations of the Research

The development of the Use Case application "*DayPlanner*" tests the feasibility of the extension to IFML present in this document. However, it needs further validation in a wider variety of development situations. For instance, document usage and experience in other types of web applications.

Another aspect that will not be tested in the scope of this thesis will be the experience of a larger scale project, that require more resources such as time, money and development teams.

Ideally, transformation tools such as a code generator, should be in charge of the Platform Independent Model to Platform-Specific Model transformation, in order to grasp the full potential of the proposed language. Even though the structure diagram to be proposed is an advance, it is insufficient to generate executable code from it, since behavioural aspects might require their own diagram.

Modern web browsers are not supporting native web components to the same extent. Although it is not an optimal solution, *polyfills* are a reasonable solution for cross-browser compatibility with the latest standards.

Even though the interviews provide valuable qualitative information regarding front-end practices, they do not intent to deliver statistically valid information, given the reduced number of subjects interviews.

3 Modeling in Software Engineering

The main goal of this thesis is proposing a mechanism model the structure of web components. In order to achieve this goal, it is important to review basic concepts and analyze previous work. However, this is not a trivial endeavour, since the nature of the work requires a critical review of a wide spectrum of concepts within Software Engineering, including model-driven engineering, web engineering, front-end development, and agile methodologies.

First, literature review intents to describe the context and specify the area that the proposal will occupy within Software Engineering. Then, it establishes a base of knowledge used as framework to justify the relevance of the proposal. This research provides the necessary toolkit (previous work, metamodels, transformations, etc.) to build the extension to IFML present in later sections.

3.1 Purpose of modeling

Models are abstractions representing the reality. Models have multiple purposes such as making generalizations, classifications and abstractions (Brambilla, 2017)

According to Brambilla (2017), the necessity of using these representations in Software Engineering rely on the following:

- Software artifacts are becoming more complex, and for that reason they need to be abstracted in a simplified, in a way they can be easily discussed.
- Increasingly integration of software within people's life is constantly requiring new software components or the evolution of existing ones.
- Software development requires interaction with external entities such as management, customers, or any other that not necessarily possess programming skills.
- Shortage of software professionals when compared to the job market.

Martin Fowler proposes three different perspectives regarding the usage of UML: UML as a sketch, UML as a blueprint, and UML as a programming language. The first one refers to the

models that communicate specific aspects of the system to other developers. UML as a blueprint refers to models created to be a guide for developers to base their code upon. Finally, UML as a programming language is applied when aspects of the software can be entirely represented by UML, and therefore the code can be automatically generated based on their graphical representation. (Fowler, 2003)

Since the classification does not refer to the standards themselves, it can be extended beyond UML, and can describe different purposes for modeling software in a more general sense. The following diagram summarizes the three main perspectives and their characteristics, based on Fowler's classification.

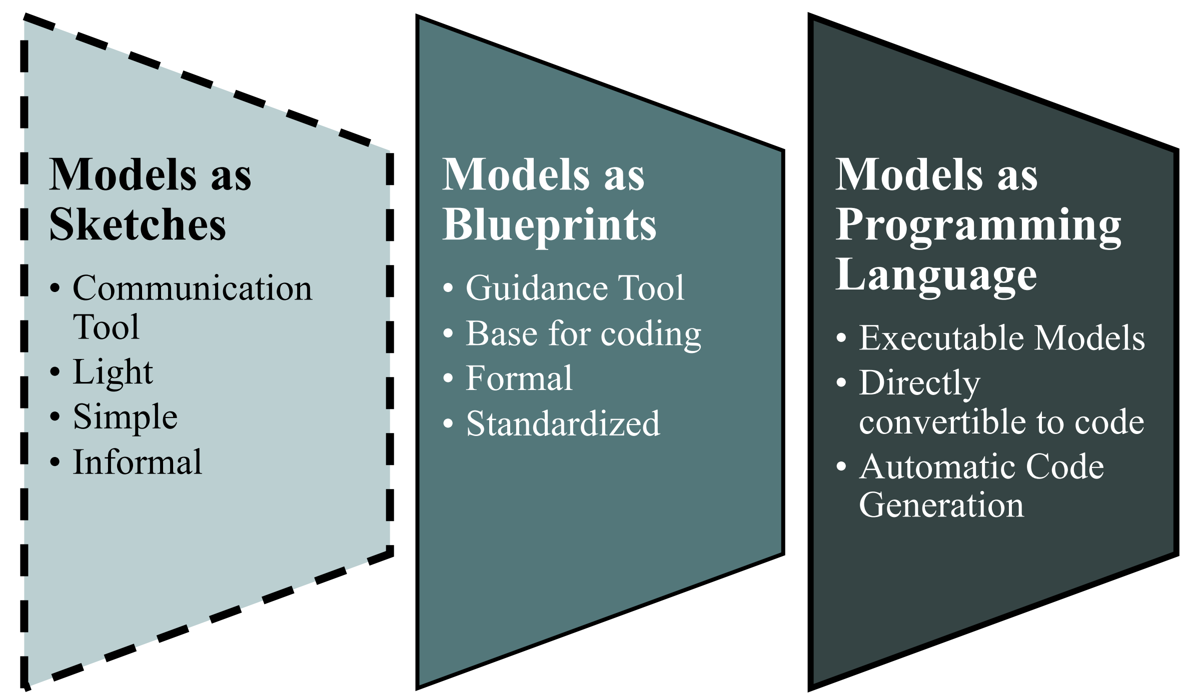


Figure 1: Classification of Modeling Purposes

3.2 Defining the scope of Model-Driven Engineering

When it comes to modeling in Software Engineering, there are recurrent definitions in the literature that are employed to refer to methodologies, standards or good practices. Some

concepts contain smaller ones, while others are part of broader categorizations. Since the present work refers to many of them, they are worth clarifying in terms of scope, in order to pursue a better understanding of the field and avoid confusion from the reader.

3.2.1 Definition of Model-Driven Engineering (MDE/MDSE)

Often referred as Model-Driven Software Engineering (MDSE), MDE is an approach to software engineering in which the main artifacts are the models and their corresponding transformations (Beydeda *et al.*, 2005). This definition suggests that MDE is a superset of other classifications where models play an important role in the software engineering practice.

3.2.2 Definition of Model-Driven Development (MDD)

Model-Driven Development (MDD) is the notion that models can be built in order to later transform them into working software (Mellor, Clark and Futagami, 2003). Based on this affirmation, MDD is a subset of MDE. However, it can comprise many other concepts due to the broadness of its definition. Modeling standards, good practices or specific tools to models fall under this classification.

3.2.3 Definition of Model-Driven Architecture (MDA)

It is a set of standards introduced in 2001 by the Object Management Group (OMG). MDA explains relationships between concepts such as models, metamodels and transformations (*The MDA Foundation Model, an ORMSC DRAFT*, 2010). It implements MDE tools by heavily relying in other OMG standards such as UML.

3.2.4 Definition of Agile Model-Driven Development (AMDD)

According to Scott W. Ambler, “AMDD is the agile version of model-driven development (MDD)” (Ambler, 2004). The main difference is that AMDD promotes the creation of lightweight models that are acceptable enough, in order to start coding shortly after.

3.2.4.1 Definition of Agile Model-Driven Architecture (Agile MDA)

Agile MDA encompasses notion that code and executable models can be the same in an operational level (Mellor, 2004). In other words, models can be executable products, instead of mere documentation.

The Figure 2 summarizes the concepts previously described and illustrates the relationship between them in terms of their scope.

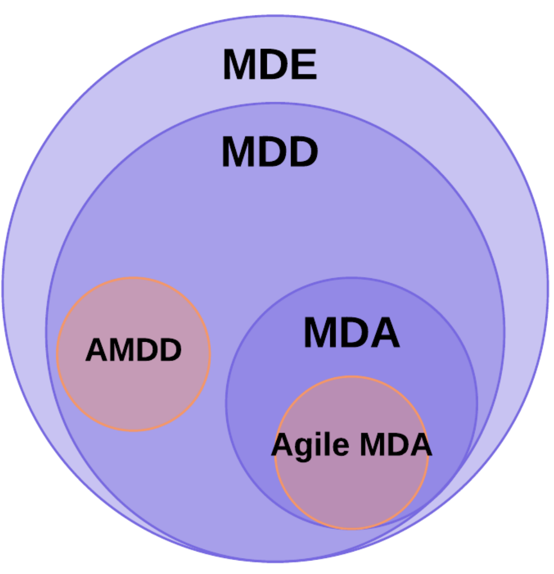


Figure 2: Model-Driven Engineering approaches

3.2.5 Model-Driven Software Engineering

Model-Driven Software Engineering (MDE) promotes the use of modeling tools such as diagrams, meta-models, model to model transformations, and model to text transformations. This is a broad categorization that encompasses many approaches towards the practice of modeling, all of which can be classified under any of Fowler's three categories, mentioned in section 3.1. Use of models prove to be time-effective and economical (Acerbis *et al.*, 2007), according to MDE supporters.

MDE encompasses a series of tools that are commonly used by many model-driven approaches. This toolkit includes: metamodels and transformations.

3.2.6 Transformations

The main focus for software engineers, when working with modeling frameworks such as MDA, is generating a Platform Independent Models (PIM). Platform-Specific Models (PSM), in the other hand, describe in much more detail a number of technical aspects PIMs do not, due to the fact that any specific platform, such as .NET, ECMAScript (JavaScript) or Java, introduce different and very specific constraints and specifications. Transformations are processes aiming to close the gap between PIMs and PSM.

It is important to make a distinction between three related and interconnected concepts: transformations, transformation tools, transformation definitions and transformation rules, all summarized in Figure 3. Transformations are processes that translate PIMs into PSMs and vice versa. According to Kleppe, a transformation definition is a group of transformation rules that must be unambiguous in a way that they have the ability to an existing model (or part of it) to create a another model. (Kleppe et al., 2003)

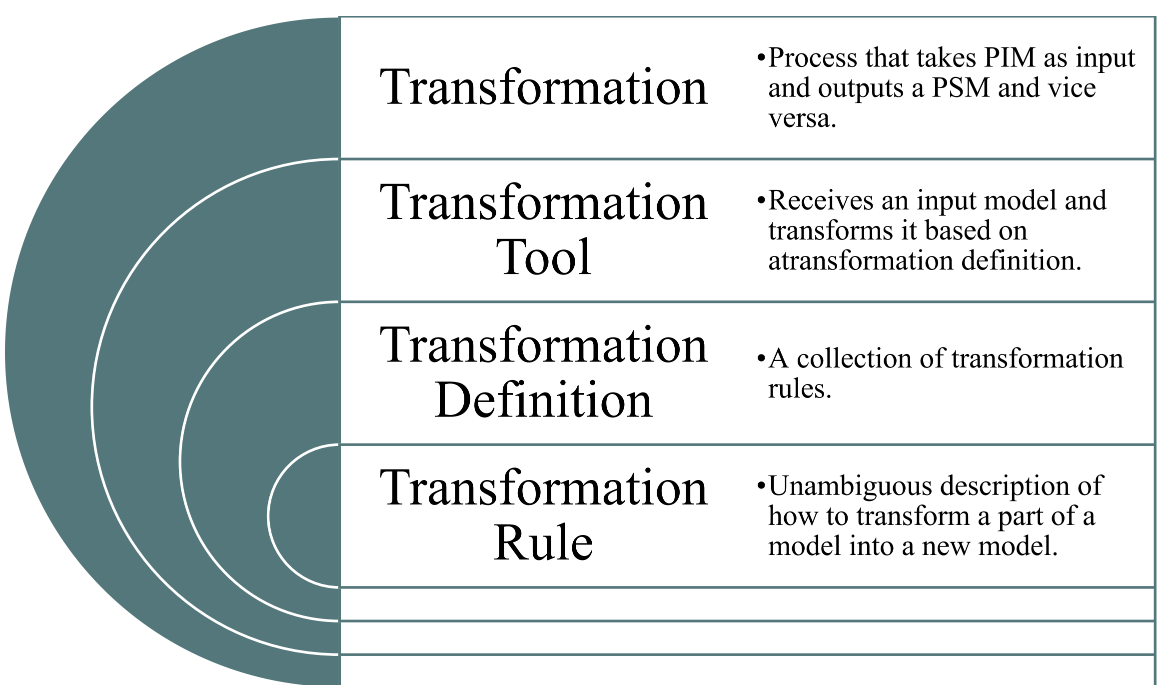


Figure 3: Transformation concepts

Transformation tools can be employed to ease the transformation by either assisting in the process, or by entirely automating it. These tools internally contain a transformation definition, which describe the rules and guidelines of how to transform pre-determined input models into new platform-specific models.

It is important to note that transformation definitions are defined between languages. For instance, a transformation definition can be described between UML and C#, or between IFML and ECMAScript, as depicted in Figure 4. Furthermore, multiple transformation between intermediate languages might be necessary to finally output code as a result.

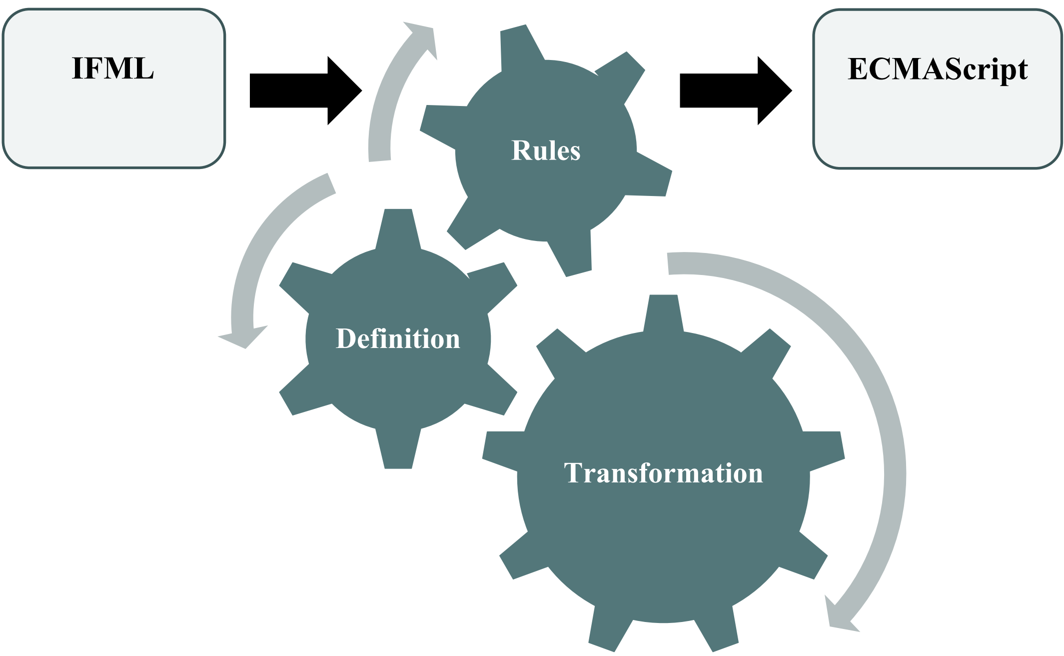


Figure 4: Transformation between languages

3.2.7 Agile and Model-Driven Development

Agile values and MDD are contradictory according to some point of views. Agile practitioners often perceive models as “heavyweight” artifacts, thus delaying the more important executable

results. On the other hand, model-driven development practitioners often consider agile methodologies as not formal enough, and sometimes contrary to the good practices of Software Engineering, which can lead to all sorts of trouble from security to maintainability. One of the reasons for the disconnection between the two paradigms is the "verification gap" (Mellor, 2004). Stephen J. Mellor describes this gap as the one that occurs when someone creates a document that cannot be executed.

In the literature there are two different approaches regarding how to resolve the problem of reconciliating both agile and traditional model driven practices. One approach is called Agile Modeling (AM), proposed by Scott W. Ambler.

AM comprises practices based on a set of principles and values, aiming to guide on how to be an effective modeler. It follows an Agile Model–Driven Development (AMDD), an agile variation of the MDD. AM adopts the iterative nature introduced by the Rational Unified Process. More importantly, AM proposes to write "light" models, instead of the extensive and rich modeling proposed by MDD. According to Ambler, this approach allows the team to spend a minimum amount of time sketching a decent enough model in order to rapidly start coding (Ambler, 2004).

Another very different approach is described by Stephen J. Mellor. He proposes Agile MDA as a way to reconcile MDA and Agile values by creating executable models. Due to the executable nature of these models, they are artifacts that produce tangible results, in alignment with Agile Manifesto values. Agile MDA achieves this by modeling the software from many different perspectives and creating relationships between those models. These relationships, along with the executable nature of the models, eliminate the necessity of implementing transformations, in contrast with MDA, which depends on transformations to produce executable artifacts.

3.3 Challenges of Modeling

The lack of agreement regarding the purpose of UML, and other modeling languages, is source of discussion in both academia and the practice in the industry. Depending on the viewpoint, a modeling language can be considered either too simplistic or overly verbose.

Some authors see UML rather sceptically, given their empirical evidence of implementations of modeling languages hurting in many ways the practice of Software Engineering. For example, according to Alex E. Bell, some potential problems of modeling are: focusing on a narrow static software design and ignoring flexible aspects such as concurrency, data coherency or latency that are often ignored; models lacking understanding stakeholders' needs; models becoming the final product, instead of a mean to an end; false expectations of UML artifacts; and the usage of modeling languages to represent other aspects than which they are originally conceived for (Bell, 2005).

Scott W. Ambler states that even though UML is a robust framework, it is not sufficient for all real-life scenarios, and problems may arise when UML diagrams start to be used for other purposes that they are intended for. (Ambler, 2014) As a solution, he proposes to either extend UML, or use models outside the framework.

Many methodologies based on the Agile Manifesto principles such as Scrum or eXtreme Programming, value "working software over comprehensive documentation" (Beck *et al.*, 2001). If modeling languages are treated as documentation, then they are not a priority. Conversely, if they are executable artifacts, they are justified by the principles of the Agile Manifesto.

MDE supporters address some of these problems by arguing that model-driven development is beneficial as long as the models belong to Fowler's third category: UML as a programming language. Taking into account that designing is less costly than coding, it would be reasonable to invest more effort designing executable UML, by implementing code generators that take models as inputs. (Brambilla, 2017)

Scott W. Ambler suggests that the rigidness of models can be solved if they are light and not too verbose, in order to make rapid iterative changes to them and to the code. Hence, Ambler's approach focuses on models as sketches, possibly even as blueprints, but never as programming languages.

4 Front-end Web Development

4.1 Background

In Software Engineering, the terms back end and front end have taken different meanings. Currently, they are employed to refer to a separation of concerns between data access and presentation layers. In client-server architectures, such as web applications, Front-end is often used to refer exclusively to the presentation parts of the software that run in the client side. (Codesido, 2009)

Examples of the software components that are categorized under Front-end are: HTML, CSS, JavaScript code running on the client side. These elements often interact and make extensive usage of visual design assets such as images and fonts.

Rich Internet applications (RIA) during the decade of the 2000's are present predominantly through installable internet applications, which often required the installation of a plug-in the browser in order to run. These plug-ins allow web applications to provide the end users with rich visual experiences such as animations, video streaming and sound. They are often based in proprietary software that do not comply with industry-wide standards. Some of the most popular installable plug-in-based technologies are: Adobe Flash, Java applet, JavaFX, and Microsoft Silverlight.

The decay in support and usage of plug-in based web applications, arguably driven by Steve Jobs and his lack of support of Adobe Flash (Jobs, 2010), pivoted the entire internet towards open standards. Standards such as HTML, CSS and JavaScript are currently the base of today's rich internet applications. These specifications are in constant evolution, and the support for more complex features is gradually being adopted by the most popular web browsers (Leenheer, 2015).

This increasing number of standard features supported by the browsers, along with the increasing capabilities of end-user devices, allow the web developers to create a more complex and heavier client-side. Web components is a set of features by W3E that aims to deal with the

increasing complexity of rich front-end web applications by providing a standard mechanism to create customizable and reusable building blocks.

4.2 Web Components

In traditional software engineering, reusability is a common practice. However, this is not necessarily true for HTML markup, resulting in very complex structures that are hard to maintain. As an answer to the problem, Web Components aim to bring component-based software engineering to front-end web development, by the implementation of encapsulated reusable components. (Mills, 2018)

Web components are a set of standards by World Wide Web Consortium (W3E) that comprises four features: Custom Elements, Shadow DOM, HTML Templates and HTML Imports. These features consist of HTML specifications and JavaScript APIs in order to achieve the following:

- Separation of concerns (clear division between markup structure, visual design and behaviour)
- Encapsulation
- Reusability

4.2.1 Custom Elements

HTML specification offers a fixed set of DOM elements, that provide semantic meaning to the content. DOM elements such as <p>, <img>, and <input> provide a clear semantic description that can be easily interpreted by search engines to offer more relevant results, and by and dictation engines for accessibility. This semantic meaning is even more to web browsers, which provide these elements with a default behaviour and a default visual representation.

It is very common for developers often modify or completely rewrite their behaviour and visual style to match business requirements. Furthermore, many of these default DOM elements are being modified in such a way, that they end up being used for completely different purposes than their original purpose, which often leads to semantic problems, which leads to other

problems such as accessibility issues (Connor, 2012). Another frequent scenario is adding non-standard DOM elements and providing them with custom functionality through JavaScript, which create non-conforming elements that are not very functional (W3C, 2018a).

Custom Elements provide a mechanism to create custom DOM elements that allow the author to expose extensibility points in order to explain platform features in a similar fashion elements in HTML do (W3C, 2018a). The specification explains two scenarios when creating custom elements: *autonomous custom element*, and *customized built-in element*.

The creation of an *autonomous custom element* is aims to define a new element, which is independent to existing HTML elements. As seen in Table 1, a new JavaScript class should be created, extending from *HTMLElement* class. The constructor definition is required, along with a call to *super()* function as the first action in the controller. The description of the custom functionality is achieved by adding more statements to the constructor and/or adding more functions to the class. Once the description of the custom element is done, the final step consists in adding it to the existing available elements with the function *customElements.define()* as shown in Table 1.

The definition of a *customized built-in element*, the second scenario when creating Custom Elements, has the purpose of reusing behaviour of existing HTML elements, by extending them and modifying their behaviour (W3C, 2018a). The implementation consists of defining a new class that extends from a class corresponding to an existing element, as shown in Table 2. Its usage slightly differs from the *autonomous custom element* in two details: first, *customElements.define()* function must include a third parameter specifying the extend option, as also described in Table 2; second, the HTML markup element must include the attribute “is”, with the value of the first parameter in the *customElements.define()* function of the previous step.

Table 1: Example of Autonomous Custom Element

	Code/Markup	Description

Class definition	class AlertPopUp extends HTMLElement {
	  constructor() {
	    // Always call super first in constructor
	    super();
	    this.message = null;
	    // add element functionality here
	    ...
	  }
	}
		Class definition in JavaScript, extending from HTMLElement. Custom functionality such as event listeners and default values can be added to the constructor.

Component usage	customElements.define(“alert-popup”, AlertPopUp);
	
	const alertPopUp = new AlertPopUp ();
	alertPopUp.message = "successful";
	document.body.appendChild(alertPopUp);
		Adding defined class to the elements registry. First parameter defines the DOM markup element name, the second references the class name.

HTML markup	<alert-popup message="successful"></ alert-popup >	This is the outcome in DOM markup.

Table 2: Example of Customized Built-in Element

	Code/Markup	Description

Class definition	class MagicDropdown extends HTMLSelectElement {
	  constructor() {
	    super();
	
	    this.addEventListener("click", () => {
	      // Draw nice custom animation effect here!
	    });
	  }
	}
		Class definition in JavaScript, extending from an existing HTML element class. In this example, a custom animation can be defined when user clicks the element.

Component usage	customElements.define("magic-dropdown", MagicDropdown, { extends: "select" });	Adding custom element to registry, and then

	`const magicDropDown = document.createElement("select", { is:" magic-dropdown" });`	programmatically inserting the HTML markup.
HTML markup	`<select is="magic-dropdown"> …</select>`	Resulting markup.

4.2.2 Shadow DOM

Encapsulation is the most important feature that Shadow DOM API adds to web components. Shadow DOM is part of the following specifications: DOM Specification, HTML Specification, CSS Scoping Module, and UI Events Specification. W3C claims that the application of this method results in a better composition of the DOM (W3C, 2018c).

Object-oriented software engineering provides mature solutions to the problem of isolating code, which is a common practice in the industry since the 1980's (Snyder, 1986). However, this is not the case for web development, where the most common solution to this problem is the implementation of HTML *iframes* (Glazkov, 2011), which creates a sandbox to protect its content from script injection and style propagation. Iframes original purpose is embedding documents in a web page. However, its usage as mechanism for code encapsulation is highly discouraged due to security (Mavrommatis and Monrose, 2008), performance and semantic reasons, which can even affect the content relevance for search engines (Mavridis and Symeonidis, 2015). In addition, iframes often results too restrictive for most encapsulation purposes.

Shadow DOM tackles the problem of isolating code, by establishing a clear boundary between the encapsulated code and the code that makes use of it. This API provides a mechanism to attach a hidden and independent DOM (known as Shadow Tree) to an element in the existing structure, having as a result encapsulated markup structure (Torres *et al.*, 2018).

The root element of the Shadow Tree is called Shadow Root. The usage of the *Element.attachShadow()* in JavaScript allows the developer to attach a shadow root to any element.

Table 3: Usage of Shadow DOM

Name	Code	Description

Shadow Root Attachment	`let shadow = element.attachShadow({mode: ‘open’});`
	
	`let shadow = element.attachShadow({mode: ‘closed’});`
		Attaching shadow root to an existing element. The main page context can access the Shadow DOM (through JavaScript) if the mode is set to ‘*open*’.

Referencing a shadow DOM	`let shadowDOM = element.shadowRoot;`	Method to reference a shadow DOM after its attachment to an existing element. This is possible if the shadow DOM attachment mode is open.

Adding elements to shadow DOM	`let newSpanElement = document.createElement('span');`
	
	`shadowDOM.appendChild(newSpanElement);`
		Attaching new elements to the shadow DOM is possible by implementing traditional methods.

Attaching style to the shadow DOM	`let style = document.createElement('style');`
	
	`style.textContent = ‘{.alert{‘ +`
	`    ‘position: relative}’ +`
	`    ‘span {‘ +`
	`      ‘color: red;’ +`
	`    ‘}’;`
	
	`shadowDOM.appendChild(style);`
		Adding encapsulated styling, by creating a <style> element and adding it to the shadow DOM.

The most important benefit of using Shadow DOM API is that the encapsulation that shadow structures create, isolates the code (and styling) inside of them, effectively avoiding conflicts with the rest of the scope.

Shadow DOM and custom elements can work together to create custom encapsulated elements, which according to Mozilla's resources for developers (MDN) is the most useful application of the shadow DOM (Torres *et al.*, 2018). Attaching the shadow DOM to the custom element in the constructor is possible by implementing a method similar to the following in the custom element's constructor:

```
let shadow = this.attachShadow({mode: 'open'});
```

The structure underneath and its styling can then be defined as part of the custom element by using the methods described in Table 3.

4.2.3 HTML Templates

HTML Templates add reusability features to HTML. The mechanism consists in defining a *template* HTML element, which can hold content that browsers do not render by default. The purpose of templates is holding entire HTML structures for rendering them at a later stage, in runtime, by adding them to the DOM via JavaScript. This is ideal for pieces of content that needs to be reused and rendered multiple times throughout the application lifetime.

As part of the standard, custom elements have the ability to define HTML templates in their constructor. Furthermore, shadow DOM structures can interact with both HTML templates and custom elements in order to accomplish modular, encapsulated and reusable HTML structures. (West, 2016)

4.2.4 HTML Imports

HTML Imports, or simply imports from here on, is a feature that contributes to the modularization of the markup and everything it contains (such as references to stylesheets and to scripts). This is achieved by the possibility of dividing the HTML into different files, and then referencing them from independent HTML files in order to make use of them (W3C, 2018b).

Imports are particularly useful in web components scenario, where HTML templates and custom components can be defined, implemented and kept in separate files, which allows for a better separation of concerns and better maintainability.

Without imports, developers need to resort to tools such as asynchronous JavaScript (Ajax), *iframes* or the implementation of *hacks* to dynamically import markup, which is not ideal. (Bidelman, 2013)

The usage of imports requires declaring a *link* element, setting the keyword *import* as the value of the attribute *rel*, in the following fashion:

```
<link rel="import">
```

The *head* section is where definitions of imports exist within a HTML file. It is also a requirement to define the path of the external HTML resource as part of the href attribute. For example:

```
<head>
    <link rel="import" href="/components/customPopup.html">
</head>
```

4.2.5 Disadvantages

Despite helping to modularize, encapsulate and reuse HTML, web components also introduce a set of challenges that the interface developer needs to consider when working with them. This section describes the main problems and some possible solutions for them.

The semantic problem with autonomous custom elements

As the Custom Elements section describes, there are two types: customized built-in elements, and autonomous custom elements. The first one extends from existing HTML elements; hence the browsers and search engines can treat the resulting custom element according to the

semantics added by the standard parent element. In the case of autonomous custom elements, this is not possible, since they are entirely new elements for which browsers or search engines do not have default semantic information for.

In order to mitigate the problem, W3C recommends a set of practices that add semantic to autonomous custom elements:

If the custom element is an interactive element (i.e. a custom button or a dialog element), the inclusion of *tabindex* attribute will make the element focusable.

The inclusion of WAI-ARIA attributes adds semantic related to accessibility. For instance, setting the ARIA attribute role to "button", adds semantic meaning to the custom element so it can be correctly interpreted as a button by accessibility technology such as screen readers. Joshue O Connor addresses these and other accessibility concerns in his book Pro HTML5 Accessibility (Connor, 2012).

Include behaviour for commonly used event handlers, such as keydown and click will provide the browser with more information regarding the default behaviour for the custom element.

Partial support of web components

Not all standards under the umbrella of "web components" are universally supported by all browsers. This reality is caused to a number of different factors. Documentation suggests the most important one is technical disagreements over the standard solutions proposed (WebKit, 2018).

Web standards by W3C are in constant revision and they subject to all kinds of technical criticism and debate. This is particularly true for the consortium members who are also browser vendors, such as Microsoft, Apple and Mozilla. For instance, both WebKit, from Apple, and Mozilla Foundation do not support HTML Imports (described in section 4.2.4), due to concerns over the way it currently works, alleging that the behaviour as described in not entirely

compatible with ECMAScript 6 Modules (van Kesteren, 2014), and they believe that there are better ways to package web components.

As browser vendors do not universally support all web components features, developers need to rely on polyfills. Polyfills are JavaScript libraries that simulate browsers' missing features, in order to support these behaviours as closely as possible to the standard specifications (WebComponents.org, 2018). Even though they are not the ideal solution, they are viable solutions to deliver websites that take advantage of the latest standard features.

4.3 Front-End Frameworks

The standard mechanisms to program reusable encapsulated components are yet in development and are object to debate and change in future iterations. However, large companies such as Facebook and Google and smaller development companies as well, have developed their own set of tools, from now own referred as frameworks, that allow component-oriented front-end development. Some examples of this are React, Angular, Vue, Backbone, etc. All of them with different scopes and with their own advantages and drawbacks. Each offers a library of tools that allow the developer to easily define, instantiate and reuse components, by providing a transparent way of interacting with the DOM.

In addition, they are open source, which allow the developers to modify the behaviour of the framework to custom it according to any particular project's needs.

4.3.1 Advantages

Support by the community: The open source nature of these framework attracts thousands of developers willing to contribute by documenting solutions to common problems, and by proposing improvements to the framework.

Repository of components: The most popular frameworks have a vast repository of commonly used components shared by developers, that are made available to the community, so other developers can reuse them for their own projects.

Support by the vendor: In addition of the support by the community, some of them also have the support of a big company, that counts with the financial means to commit to the continuous improvement of the framework. Such is the case of React, by Facebook, and Angular, by Google.

4.3.2 Disadvantages

Obsolescence: Third party frameworks are in constant evolution, releasing new features and security patches. A common recommendation is to keep the production code up to date with the most recent versions of third party libraries, including the mentioned frameworks. This is a challenge in projects with reduced amount of resources for maintenance, especially if the updates in the framework require many changes and updates to the application existing code. The lack of updates may lead to other problems such as security and compatibility.

Migration: If the team decides to move to a different technology, migration times can be high, in which cases a rewrite of the code will probably be the best solution.

Familiarity: A team that does not possess the proper training to take advantage of the framework in question, can face all kind of different problems such as: very steep learning curve, low code quality and higher development times.

4.4 Atomic Design

In web design field, design systems and particularly atomic design is crucial for front-end web development. Atomic Design brings consistency and reusability to web design. Brad Frost describes it as the process of building larger, more complex visual elements by using smaller building blocks (Frost, 2016). These building blocks can be combined in any way to form new elements that are visually consistent.

His classification for these elements goes form simpler to more complex: atoms, molecules, organisms, templates, and pages; where molecules are formed by a group of atoms, organisms by a group of molecules and so on.

A developer can take this design approach and easily translate it to front-end component-based development, where smaller components can form more complex components in a hierarchical

way, since it provides a clear and consistent way of reusing existing smaller components. Atomic design is technology agnostic, so standard Web Components or any of the existing components-oriented front-end frameworks have the ability to map atomic designs.

5 Existing Modeling Languages for Web

5.1 Web Application Extension (WAE) for UML

Jim Conallen proposes Web Application Extensions for UML in 1999, in his book Building Web Applications with UML (Conallen, 1999). His original work consists in the introduction of new semantic elements to UML that aim to reflect architectural aspects of web applications of the late 90s and early 2000s. He proposes new stereotypes, tagged values, and constraints that, even though they lack official support by Object Management Group, they gained a considerable amount of traction among practitioners. However, it also requires several types of UML diagrams to model different aspects of a single functionality, and not even then is it possible to map all aspects with the extensions proposed. Despite that this makes it difficult to create an executable model, many practitioners find it useful as a starting point that can generate web pages templates, in order to use them as a starting point (Hennicker and Koch, 2001).

5.2 WebML

The introduction of the notation Web Modeling Language (WebML) is in the year 2000. It addresses the necessity of modeling web applications of the time. The specification of a website expressed in WebML contains four aspects: a Structural Model, which expresses data content entities and the relationships between them; a Hypertext Model, describing hypertext related information such as the composition and navigation between pages; a Presentation Model, that is responsible for the layout, able to contain graphic appearance information; and a Personalization Model, that reflects user and user group customization aspects, in order to store specific information depending in the user or user group. (Ceri, Fraternali and Bongio, 2000) Interestingly, WebML also supported XML syntax, in an attempt to facilitate the support of software generators.

5.3 Interaction Flow Modeling Language

The Interaction Flow Modeling Language (IFML) uses the experience acquired by the usage of WebML in order to specify a new up-to-date language. It was developed between 2012 and 2013, and in March 2014 it is fully-recognized and adopted as a standard by the Object Management Group (OMG). It is a platform independent model (PIM), which aims to model user interface aspects of a web application. IFML includes tools to model front-end characteristics such as composition of the view, content of the view, interactions, actions, the effects of the interactions and the communication between elements of the user interface (UI). (Brambilla and Fraternali, 2014)

Due to the relevance of this modeling language to the present work, the section 7 explains in greater detail key aspects of IFML such as scope, principles, artifacts and other relevant features.

6 Industry Practices (Interviews)

This section describes the most important challenges practitioners face when developing components in front-end. In addition, determines which techniques or technologies they employ to achieve reusable components, and finally analyze which role modeling has in their daily practice in the industry.

The information below is a result of four interviews to professionals working in different technology organizations based in Costa Rica and the United Kingdom. The interviews are conducted in a semi-structured manner, where questions are formulated according to the findings in the literature survey also present in this document.

6.1 About interviewees

Each participant has at least five years' experience in front-end web development. They develop reusable web components for a diverse range of industries: retail, music and health services. They usually work in teams of 4 to 8 people, and employ agile tools for their development processes. In addition, they have wide experience in the creation of reusable components in front-end web development.

6.2 Techniques and Technologies

The interviewees use different approaches and technologies when it comes to the development of reusable components. Most of them are currently making use of front-end component-based frameworks (see section 4.3 for more details).

Two of the participants are developing React components, another one is coding by taking advantage of Vue framework, and the last one primarily employs custom-made components by taking advantage of classes and other features ECMAScript provides. In addition, one of them reports the usage of custom elements (web components) in the past, by using the library *Polymer* to achieve cross-browser compatibility.

6.3 Challenges when Developing Front-End Reusable Components

Each participant describes how challenging the following problems are, by ordering them from most challenging to least challenging. The least challenging has the lower score (1), conversely the most challenging is the one receiving the highest weight (8). The results are:

Table 4: Challenges of Front-End Reusable Components

Challenge	Interviewee A	Interviewee B	Interviewee C	Interviewee D	Total Score
Reusability	6	7	6	6	**25**
Share information across development team.	8	8	4	5	**25**
Cross-browser support.	2	2	8	8	**20**
Markup consistency	7	3	3	7	**20**
Encapsulation	4	6	5	3	**18**
Packaging	1	5	7	4	**17**
Styling	3	4	2	1	**10**
Isolation of pieces of code	5	1	1	2	**9**

Reusability is one of the two most challenging problems when it comes to development of reusable components. Participants mention that in the common day to day practice it is challenging to determine whether an existent component can be reused in new scenarios. The engineering team has to determine whether to add new features to an existent component, or to

create a new component altogether. Even if the team decides to modify existent component, the developers need to ensure that it is still compatible with existing usages.

Sharing information across development team is the other most challenging problem. The participants often mentioned that repetition of work can happen when existing components are not being reused due problems of communication.

One of the participants, in addition to the list provided, mentions two additional challenges: offline front-end development, and performance. Both being the most challenging part of his practice in his opinion. In contrast, this participant also mentions that the usage of appropriate tooling is crucial to automate the exchange of information, improving team's awareness and communication as a result. Based on this participant's experience, there are other aspects that a good workflow and tooling have the ability to address challenges such as: consistency in markup, styling and isolation of pieces of code are being address.

6.4 Role of Modeling in Front-End

To discover the role modeling has in the industry, an unordered list of actions related to possible motivations for modeling was presented, so the interviewees could identify how and why they use models. The list provides actions that correspond to one of the three Fowler's classification, present in greater detail in section 3.1. The results are present in Table 5.

Table 5: Usage of Models by Interviewees

Models Classification	Purpose	Interviewee A	Interviewee B	Interviewee C	Interviewee D
Sketches	A way of communicating aspects of the solution with co-workers.	X	X	X	X
	Informal sketches for personal use.	X	X	X	X

Blueprints	As guidance for coding.	X	X	X	
	As a formal document	X		X	
	As standardized language			X	
Programming Language	For automatic code generation			X	X
	As executable models				

It is clear that interviewees employ models sometimes as Blueprints, but mostly as Sketches, according to Fowler's categories of purposes for modeling. This is relevant to this research because the modeling proposal will probably be employed as a way to communicate ideas or as a guide to base the code development on.

Two of the participants use some tooling for code generation, based on some type of models representing, for instance, graphic design aspects. However, these tools only generate a small portion of the code, and they do not have the ability of representing all aspects of components, or all the necessary aspects of a complete web application.

6.5 Viability of Usage of Models in the Industry

As general consensus, interviewees completely agree with the following statement: "Front-end web development has become richer in features, therefore, higher in complexity". The manifest that even though there certainly is a quick development of software and capabilities that provides developers with more tools, it is also true that it comes as an answer to the increasingly complexity of client-side development.

Due to the increased complexity described, all participants see the many potential benefits of modeling in front-end. From 1 to 5, being 5 the highest possible score in usefulness, the final

average score is 4.75. They consider modeling front-end components of high importance because they believe in models as a way of achieve better communication across the development team. In addition, they manifest that modeling promotes better design and planning, which becomes key when coding.

They recognize the high complexity of modeling web components, especially if some kind of standard and consistent modeling is the goal. Basing their analysis in the complexity, from 1 (trivial) to 5 (extremely difficult), the average complexity of achieving a standard modeling language for components is 3.75. They consider it an achievable goal, but a highly difficult one. They manifest that this task becomes higher in complexity as the modeling should include all possible scenarios for all possible components. In addition, the maintenance and constant evolution of such language can become a task that requires a high level of attention and a high amount of resources, because it would need to keep up to date with new requirements and scenarios.

6.6 Modeling and Agile

In the context of agile tools and processes, they score the feasibility of models' usage as 3.5 in average, being 1 not feasible and 5 highly feasible. Fitting modeling into short development cycles are the main challenge they mention. However, they consider it relatively feasible if they can prove that modeling can effectively reduce development time and help to show working code in less time.

Under the assumption of the existence of an automatic code generator, based on a model, the feasibility averages 5 among all participants, because they claim that under this scenario modeling time can also be considered time that is effectively being used to develop the final product.

7 Interaction Flow Modeling Language (IFML)

This section starts with a general overview of the language and then describes in more detail the features that are relevant for the objectives of this thesis. The main goal of this work is not to provide an extensive description of all the domains covered by IFML, but to highlight only relevant parts of the language in order to successfully be able to model the structural aspects of HTML web components.

7.1 Scope of IFML

IFML is concerned about the user-interface, including visual components and their related user interactions that may trigger events. IFML standards comprises five different domains of the user interface(*IFML Specification*, 2015):

1. View structure: Models the hierarchical organization (nesting), visibility and reachability of UI elements in terms of *View Containers*.
2. View content: In a similar fashion to ViewContainers, it expresses nesting, visibility and reachability, but in terms of *View Components*. However, unlike View Containers, they represent elements for content display or data entry. Content display can also be linked to a *ContentBinding* to specify the origin of the data.
3. Events: They can be triggered by a user, the application or by an external agent. They have the ability to change the state of the user interface.
4. Event transitions: Describes the effects produced as a result of an occurrence of an event. It includes changes in the View Containers, on the content displayed, or in the triggering of further events.
5. Parameter binding: Declares dependencies in terms of inputs and outputs between View Containers, View Components and events.

7.2 Principles

IFML was created based on a set of principles summarized in the following figure:

1. Conciseness	•Minimum number of diagram types •Minimum number of concepts
2. Inference from Context	•Avoidance of redundant information •Inference of information from other parts of the model
3. Extensibility	•Adaptation to new requirements •Standard to define new concepts
4. Implementability	•Executability •Transformations and automatic code generation
5. Not everything in the model	•Avoidance of presentation aspects •Delegation to other models of non UI-related aspects.

Figure 5: IFML principles

- The principle of *conciseness (1)* aims to avoid redundancy, and advocates for the creation of the lowest number of diagram types and concepts as possible, leaving out unnecessary details that can be detrimental to the readability or implementability of the model. (*IFML Specification*, 2015)
- *Inference from the context (2)* refers to the ability of making use of information from other parts of the model, instead of resorting to repetition of information (Brambilla and Fraternali, 2014).
- IFML also specifies a standard manner for *extending (3)* the language, which permits a high level of customizability. This characteristic makes this language highly adaptable to new technologies or devices. For instance, it allows the creation of new interface components or event types (*IFML Specification*, 2015).
- Even though IFML is a platform-independent language, it procures the usage of models that can be easily transformed into executable code. This principle is called *implementability (4)*, and it heavily depends on the support of external tools to be achieved. If a model cannot be supported by tools and cannot be easily executed, then it is avoided (Brambilla and Fraternali, 2014).

- The last principle, *not everything in the model (5),* suggests the possibility of using other languages or external applications to represent other areas of the application (*IFML Specification*, 2015), especially those that are not part of the graphic user interface (Brambilla and Fraternali, 2014). For instance, UML can be used instead of IFML in order to model back-end processes, sequence of operations, etc.

7.3 Key features

As mentioned in the section 7.1 about the scope of IFML, this language models a wide spectrum of domains, all concerning a user interface. However, this author considers the following features the most important ones in order to reflect structural aspects of a user interface, which is the main objective of this thesis:

7.3.1 Modularization

According to the specification, this modeling language ensure model-level reusability, which permits the storage, searchability an reuse of the models within the same application or even in other applications. IFML achieves this mainly through the use of Reusable Modules, which can encapsulate structure and functionality, and that communicate with external entities through input and output *ports*, in order to ensure the flow of information.

7.3.2 Extensibility

IFML includes a number of core concepts that can be extended to add more semantic details to the language, to improve readability and understandability, or to add meta-information that can be useful for code generation tools (for instance, adding extra model checking). By default, the specification contains some standard extensions, as seen on Figure 6.

In that example, Window is an extension of View Container; Form, List and Field extend from View Component or View Component Part; Submit Event, Select Event are extension from Event; Device, Position and User Role are extensions of ContextDimension.

In addition to the mentioned standard extensions, custom ones can be defined depending on the specific necessities of the application.

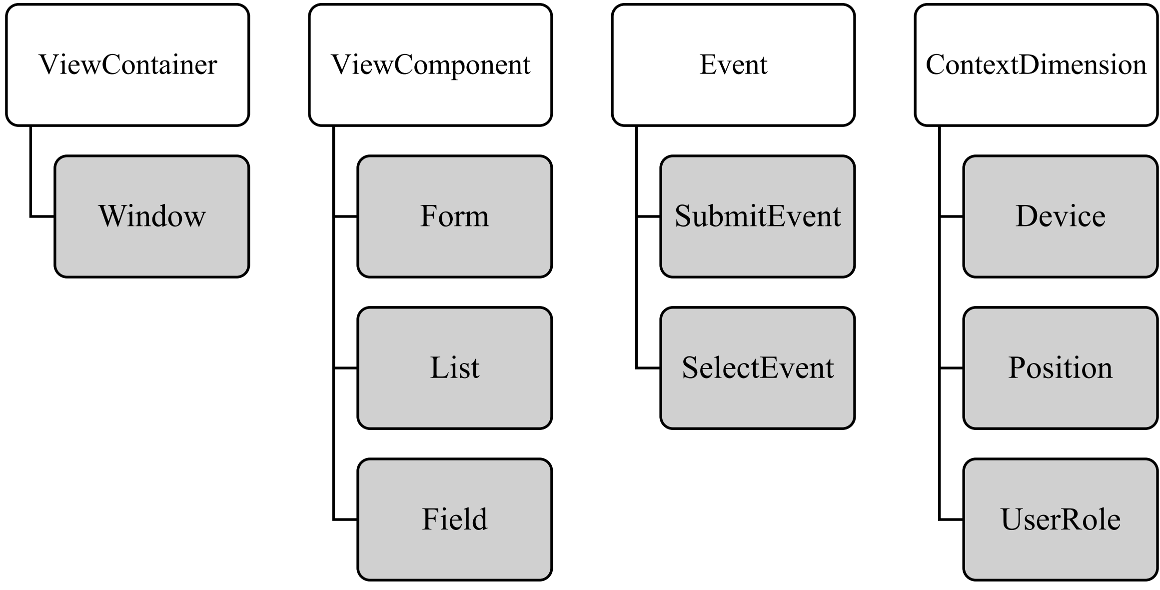


Figure 6: Examples of Standard Extensions in IFML

7.3.3 Personalization

Composition of an interface cannot be a static definition. The composition can vary depending on user actions, information, or context. IFML defines the concept of *ContextDimension* to support such adaptations in terms of composition. This is a purposely very broad concept, in order to encompass a wide variety of circumstances. For example, as seen in Figure 6: Examples of Standard Extensions in IFML, *Device*, *Position* and *UserRole* are all standard extensions of ContextDimension, and they all contain key information that will determine the necessary adaptations depending on the type of device (i.e.: PC, tablet, mobile phone), the position of the device (i.e.: portrait or landscape mode) or by the role a particular user assumes (i.e.: an

administrator, an editor or a reader). The importance of this feature relies in the fact that more extensions can be defined, and they can be as specific or as broad as needed.

7.4 Artifacts

IFML consists of four artifacts. The first is the metamodel, which contains the semantics and structure information of the language. The second one is a UML profile, which defines a UML-based syntax to express IFML models. The third artifact is the *visual syntax* itself, that comprises aspects of the GUI in a concise way, which would otherwise require several UML diagrams to represent (see Figure 7: IFML Visual Syntax scope below). Finally, XMI is the artifact that provides the exchange format for IFML portability to different tools. (*IFML Specification*, 2015).

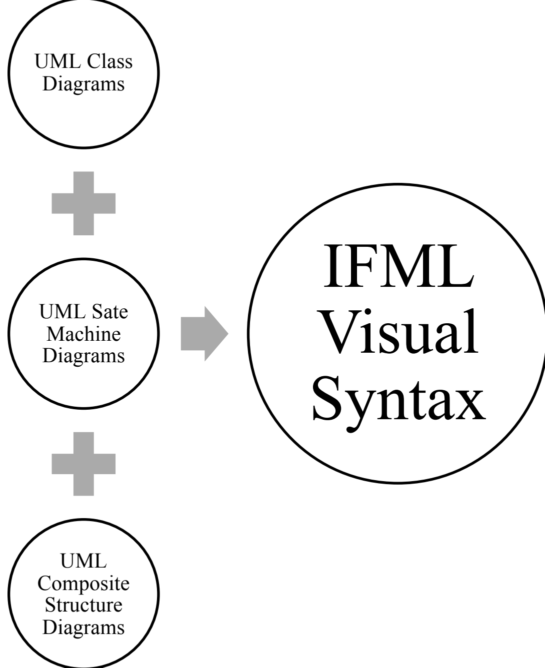


Figure 7: IFML Visual Syntax scope(*IFML Specification*, 2015)

7.5 Visual Syntax

In accordance to the other sections describing IFML, this one does not aim to give an exhaustive description of its entire visual syntax. However, it describes the basic concepts that will be used as the building blocks for later sections of this work.

7.5.1 Concepts

Table 6: Basic IFML Concepts

Name	Notation	Description
View Container	View Container	Basic container. It can represent a Window (desktop applications) or a Page (web applications). They possess the ability of nesting, in order to describe composition or hierarchy.
View Component	«Form» View Component	An element that is able to display content. It is also capable of accepting input data. The extension "Form" is presented in the notation example.
View Component Part (Parameter Binding)	«DataBinding» [Name]	Regularly used to express input or outputs between View Components. Contains a type and a name.

Navigation Flow		Used to express Navigation Flow between View Components or View Containers.
Data Flow		Express the flow of information. It is often used to describe parameters sent from a supplier to a target.
Module	Module	Reusable piece of interface. It can also encapsulate actions, and it is often used for maintainability purposes in a model.

The View Container is one of the most important concepts in IFML, since it is the basic unit used as a base to contain other elements and nested View Containers. A View Containers can be a Default[D], a Landmark [L], or an XOR. Refer to Table 7: View Container Types for a description of their function.

Table 7: View Container Types

Name	Notation	Description
XOR View Container	[XOR]TabsContainer	This is a parent container that comprises a set of View Containers, that can be displayed, but only one at the time. This means that any nested View

		Containers are mutually exclusive to their siblings.
Default View Container	[D] FirstTab	A View Container is marked as Default when it the default view among a set of siblings contained under a common parent View Container.
Landmark View Container	[L] Header	A Landmark View Container can be accessed by any sibling View Container or any children sub containers. For instance, Header Section in a website containing the website title and logout options can be marked as a landmark and can be reached globally.

7.5.2 Standard IFML extensions

In addition to the *View Container* types described above, IFML defines standard extensions for the View Container, with the purpose of adding semantic and non-functional details to the container. Listed below are some examples and their respective notation:

- «Window» MainPage (i.e.: for a main HTML page)

- «Modal» Warning (i.e.: an HTML pop-up alert)
- «Modeless» Toolbar (i.e.: an HTML modeless draggable component)

In addition, IFML standard extensions for *View Components* in order to model aspects of the components such as *Type* and *DataBinding*. These extensions are not only useful for automatic code generation tools, but they are also key for the readability and maintainability of the model. Table 8 describes examples of the mentioned standard View Component extensions.

Table 8: Examples of Standard View Component extensions

Name	Notation	Description
Form	«Form» User Registration «TextField» username: String «PasswordField»password: String «EmailField» email: String	Example of HTML form for user registration.
List	«List» ProductsList «DataBinding» Product	View Component displaying a list of instances of the type specified as DataBinding
Details	«Details» ProductDetails «DataBinding» Product	Component shows details of the instance specified as DataBinding.

7.6 Challenges of Modeling Front-End with IFML

Amongst the existing efforts, IFML is the modeling standard that has the better capacity of abstracting front-end elements and interactions. Nonetheless, it still has a number of limitations:

7.6.1 Implementability and support

As describing in section 7.2, one of the IFML principles is implementability, which procures to employ models that have the ability to easily be transformed to executables. However, only a few tools provide automatic model to code transformations out of the box. And even tools that support it, such as WebRatio, might not have all the features needed. Therefore, the automatic transformation tools will need customization (WebRatio, 2014), or they need to be developed in its entirety.

7.6.2 Graphic design aspects

Graphic design aspects, (i.e.: font-sizes, elements positioning on the screen, or colors) are antagonistic to abstraction (Brambilla and Fraternali, 2014). This means that front-end elements have the ability to be modeled only up to a certain extent, leaving out a high number of other concrete elements that cannot be abstracted. Thus, executable models or automatic code generation will not be sufficient, and styling aspects of the website need to be manually coded in order to achieve a highly customized user interface.

8 Extending IFML with Web Components Features

8.1 Justification

Web Components specification comprises both behavioural and structural aspects of web development. This section extends the existing tools that IFML provides, in order to map the structural aspects of web components. This proposal also portrays some behavioural aspects that are tightly coupled with the structural ones such as parameters.

Even though this author recognizes the importance of web components behaviour, he also considers that the effort of focusing on structure presents enough complexity to be object of study on its own. This is also a valuable starting point for any future work that aims to either improve the existing tools or to introduce improvements in the modeling of behavioural features.

8.2 Extending IFML to Model Web Components

The following is a model that summarizes the mapping that this author considers appropriate to represent HTML Components in terms of IFML. More details about each mapping present in the table below is present in the following sections.

Table 9: Mapping Web Components to IFML

W3C Specification	IFML Artifact	**IFML Extension**	Description
Custom Elements	ViewComponent	«custom-tag» CustomComponent	A custom HTML tag will be specified as a custom ViewComponent.
Shadow DOM			The new Shadow ViewContainer

	ViewContainer	[S] ViewContainer	type will be identified with an [S].
Templates	Module	Map to HTML Template. No change in visual representation.	Any module will be automatically mapped to an HTML template.
HTML Imports	ModulePackages	No extension needed	No adaptations should be applied.

8.2.1 Custom Elements

IFML already provides a mechanism to define custom ViewComponents. Furthermore, as described in section 7.5.2, IFML defines its own standard extensions. Extending ViewComponents is ideal to model custom elements in HTML. Adding the custom html tag between Guille-mets (« ») can define a new custom element in terms of IFML.

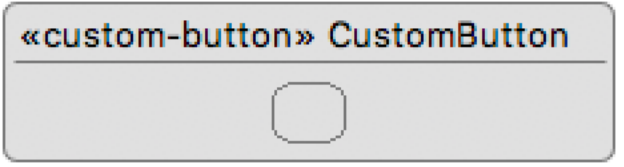


Figure 8: ViewComponent representing a Custom Element

This author also considers ViewComponents ideal to represent custom HTML elements because, by default, they possess other relevant features such as incoming parameters. *ViewComponents* is also capable of nesting *ViewComponentParts* with information such as *DataBindings* or other special sub-components that are mappable to nested html structures. For instance, a custom form can contain *ViewComponentParts* specifying special fields.

8.2.2 Shadow DOM

Shadow DOM's main feature is defining boundaries in order to isolate entire hierarchies (4.2.2). As section 7.5.1 discusses, *ViewContainers* besides representing Windows or Pages, are also

capable of nesting other ViewContainers, which makes them suited for representing entire visual hierarchies. This feature alone is enough reason to use *ViewContainers* as tools to represent Shadow DOM boundaries. See an example of a ViewContainer with nested elements in Figure 10.

However, not all ViewContainers should define boundaries of isolated sections of code, since web developers are often interested in sharing features such as cascading styling or in extending scripts' reach across many nodes in the DOM. In order to control where to set Shadow DOM boundaries, a new IFML *ViewContainer* type for this purpose alone.

This author suggests to define Shadow DOM boundaries by creating a *Shadow ViewContainer*. This new type can be an extension of existing types (described in Table 7), and it can be identified by using the label "[S]". Figure 9 is an example of using a Landmark type along with a Shadow type. Figure 10 is an example of using a Shadow ViewContainer on its own, which isolates all elements defined underneath from the rest of the code.

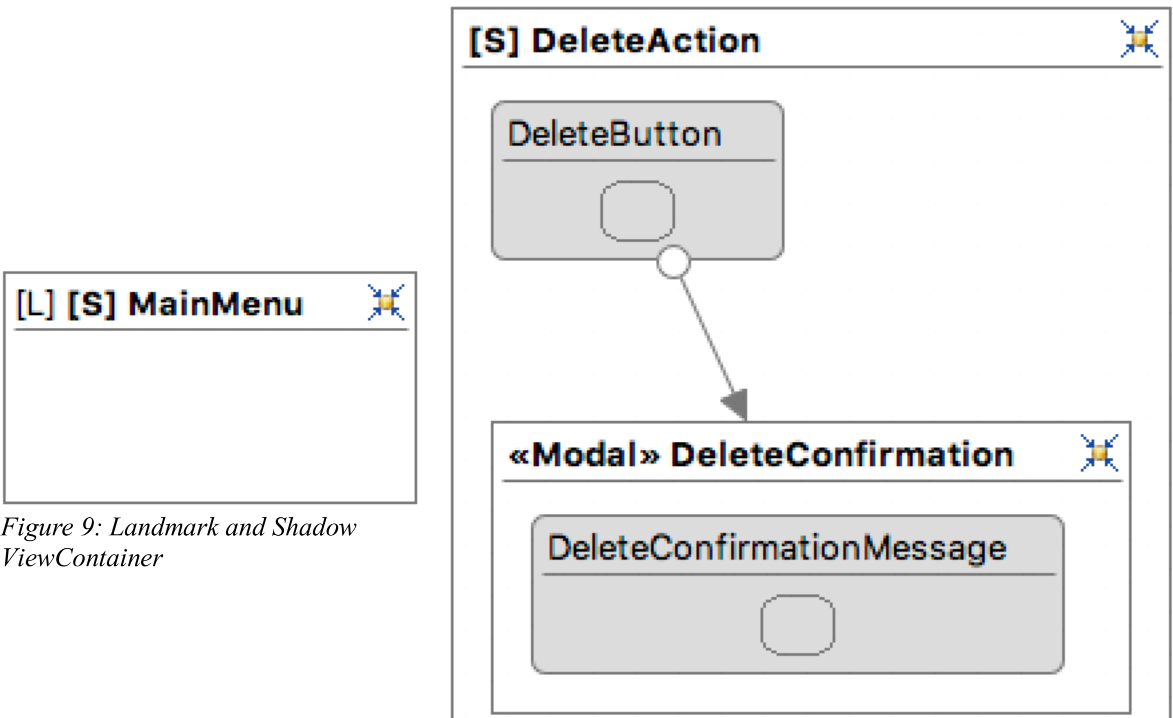


Figure 9: Landmark and Shadow ViewContainer

Figure 10: Shadow ViewContainer with nested elements

8.2.3 HTML Templates

As section 4.2.3 describes, HTML Templates main purpose is to provide reusability features to web development. On the other hand, IFML introduced modules for the same purpose, as show in Table 6. For that reason, IFML Modules is the evident tool to model HTML Template features. Modules can, in addition, define input and output ports that can be mapped into JavaScript code to provide templates with customized features.

8.2.4 HTML Imports

In the *HTML Imports* specification, as section 4.2.4 explains in greater detail, aims to package components in independent HTML files that other HTML files can import and use. IFML's Module Packages are the evident choice to map HTML Imports, since "ModuleDefinitions can be aggregated in a hierarchical structure of ModulePackages" (*IFML Specification*, 2015).

Even though *HTML Imports* is part of the standard specification, the present work does not focus on Imports, since there is an ongoing technical current debate in the industry (for more details, see section 4.2.5).

9 Use Case: DayPlanner Application

An entire process of designing, modeling and developing a web application is described in this section in order to test the extensions described in Table 9. The main goal is to validate the original work and to have a better understanding of the capabilities and limitations of the tools by undertaking the development of a small web application.

The validation is done in three main stages. Each stage outputs an artifact that serves as input for the next stage. The initial stage is creation and definition of wireframes, that will be the base for the second stage: the modeling in IFML. This model will finally be the input for developing the code. This transformation from model to code is not done by automatic generation tools but is accomplished manually according to the mapping defined.

9.1 DayPlanner Description

The validation of the extensions proposed is achieved by the implementation of a "Day Planner" web app, that follows the style of a To-Do Lists application. In addition, it displays to the user the information of tasks completion, which automatically updates depending on the user's input.

This validation does not include interactions with the server. Even though IFML is capable of modeling such interactions, this particular feature goes beyond the scope of the present work. Hence, the life cycle of the application lives within the lifespan of a web browser's window life cycle. Therefore, this application does not send or retrieve information from the server, other than the initial HTTP request. Once loaded, the application runs and dies entirely in the client-side.

9.2 Application Design and Wireframes

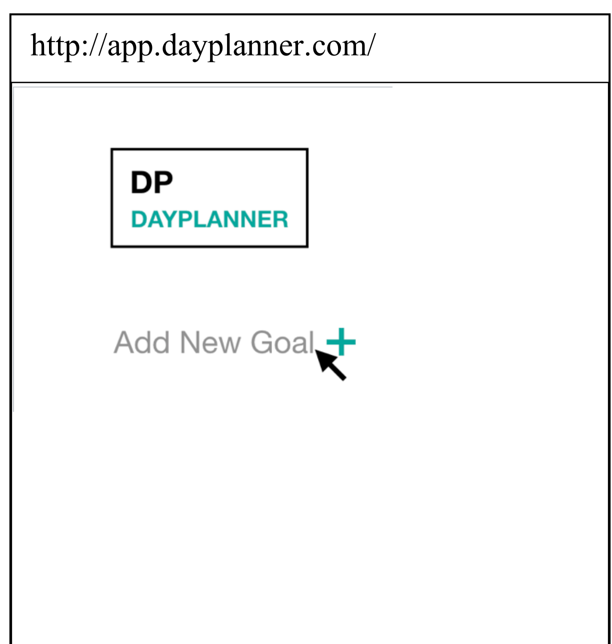


Figure 12: Initial state

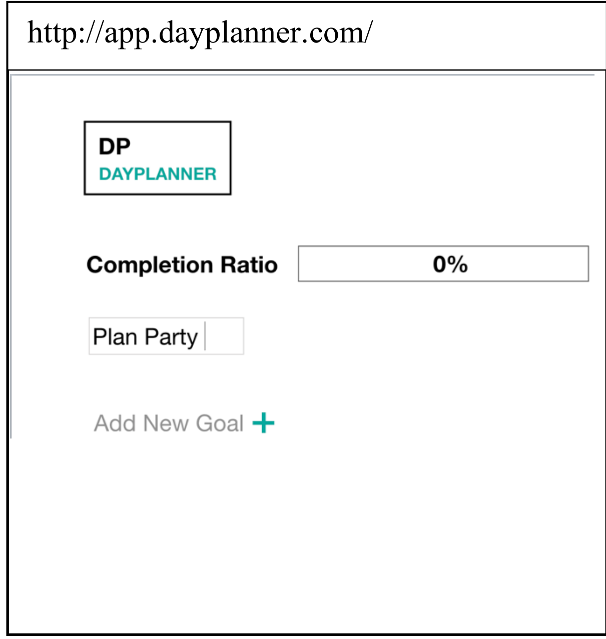


Figure 11: Add New Goal

This section contains wireframes that represent the developing application before modeling. These artifacts are the main input for creating the model.

Figure 12 represents the initial state of the application. It contains the logo and a button to add a new goal. Figure 11 describes the result after adding a new goal with the name "Plan Party".

Note that a new section with information about the completion percentage is now visible, with an initial state of 0% completion.

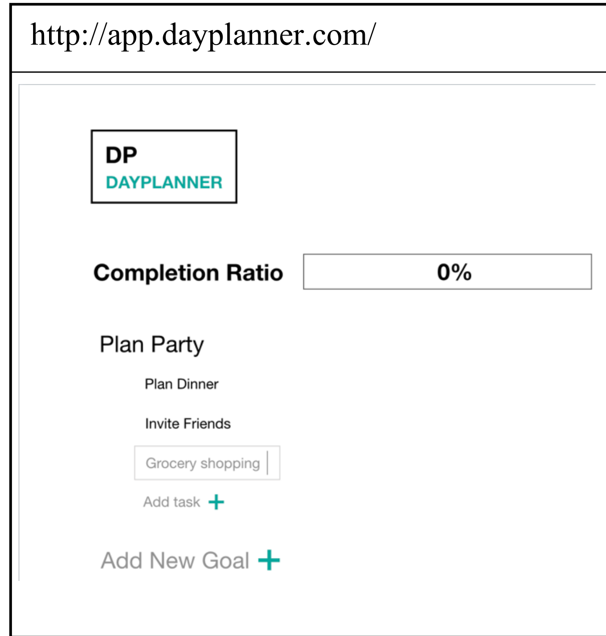


Figure 14: Add Tasks

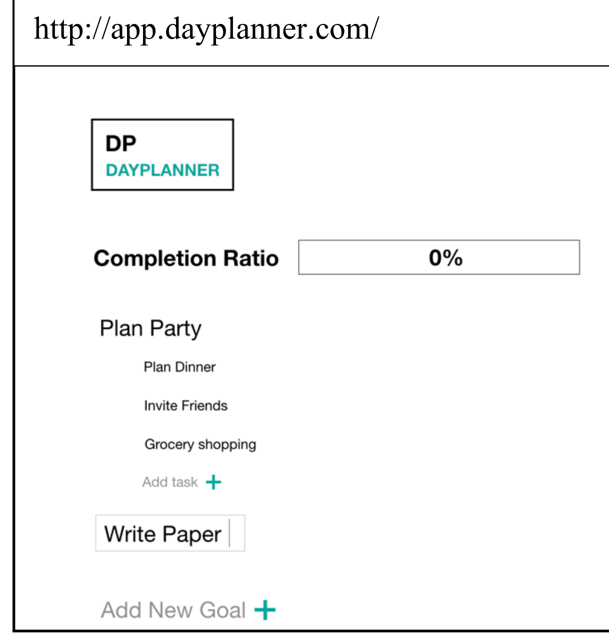
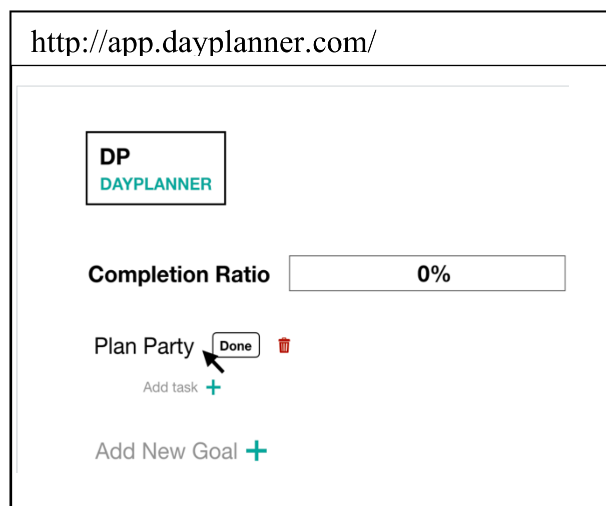


Figure 13: Add Another Goal

The user is capable of introducing new tasks to accomplish the goal by clicking on the "Add Task" button located under the goal created (Figure 13).

In **Error! Reference source not found.**, the user is adding a new goal. If desired, the user can



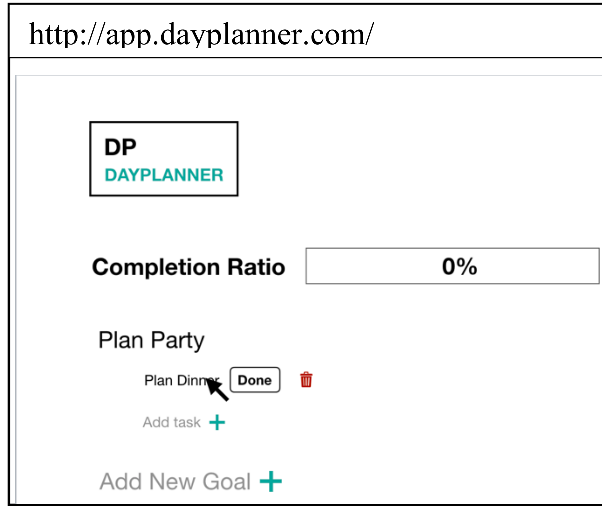
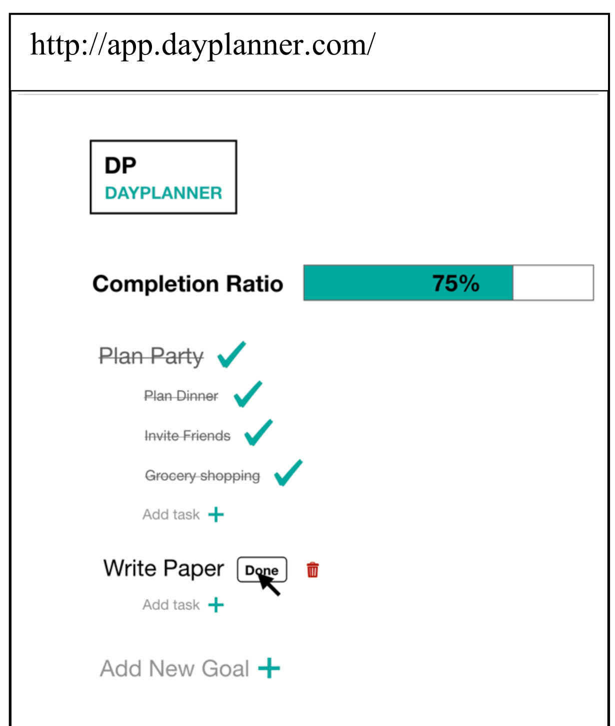


Figure 15: Hover State

add more tasks underneath each goal. Goals are capable to live in their own, and they can act as regular tasks if no tasks are defined underneath.

Tasks will always have hover states as Figure 15 shows, that will display different actions depending on whether the task is pending or if the task is done. If there are no tasks defined under a goal, it will act as a task, showing the same hover states.

In a pending task (or goal), the hover state consists of a button that marks the task as done, and an option to delete the task. Figure 16 describes the hover state actions and how the interface updates after marking a task as done. Notice that a checkmark will appear next to each complete task, and the completion ratio bar and percentages change.



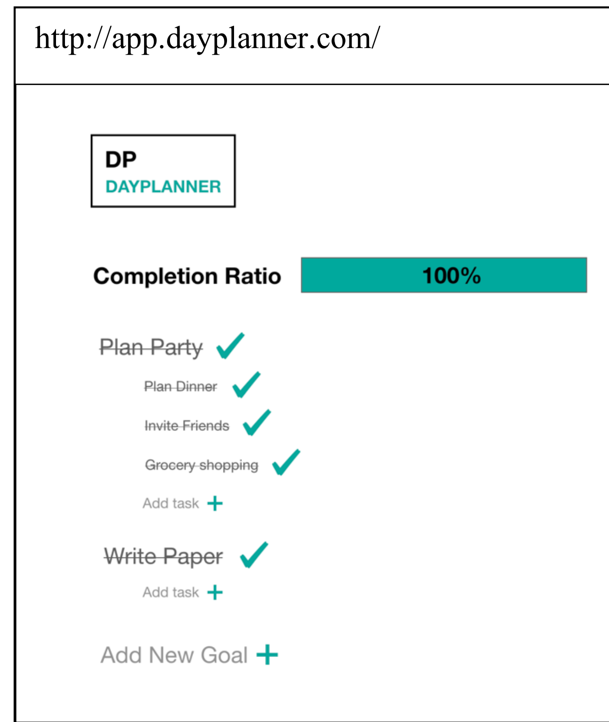
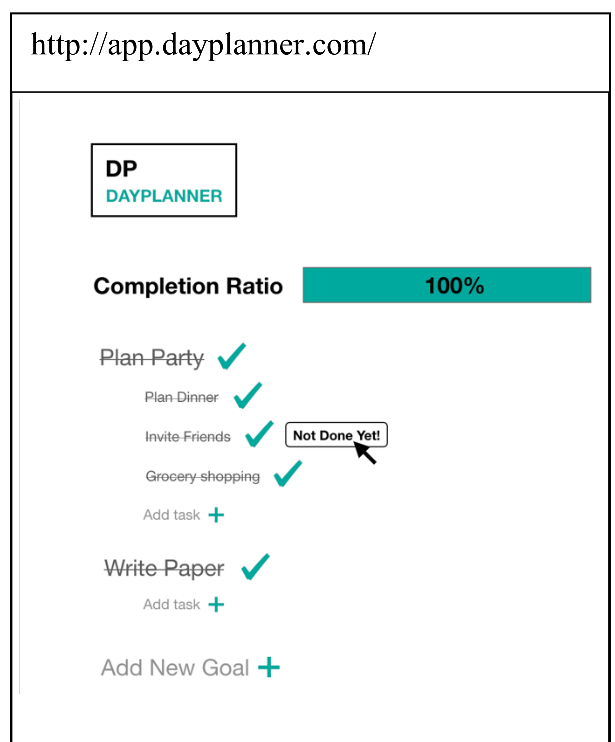


Figure 16: DayPlanner Mark as Done

Hover state actions change once the task (or goal) is done. Notice that inFigure 17, the hover state reveals a button with an action to mark the task's state back to pending. After changing the state, the parent goal state is also undone, and the completion ration updates accordingly.



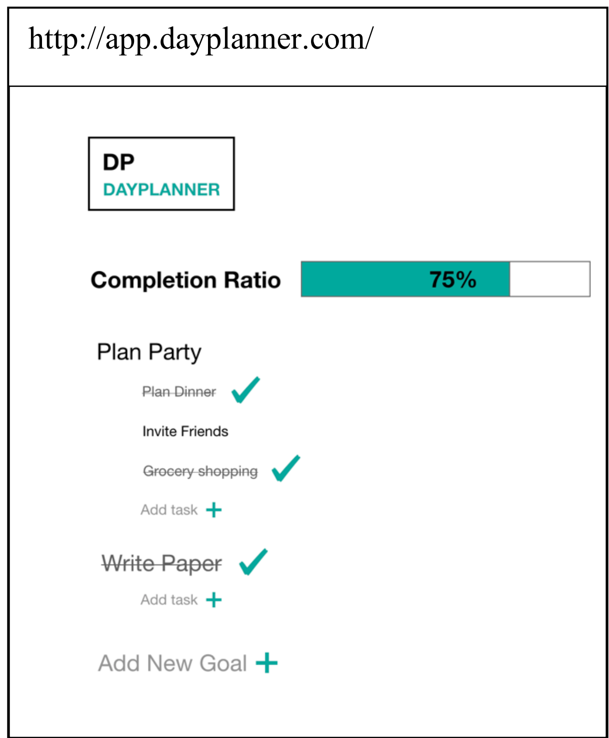


Figure 17: DayPlanner Mark as Not Done

Before deleting a task, a confirmation dialog pops in order to prevent accidental deletions. This dialog will be presented as a modal on top of the rest of the content. See Figure 18.

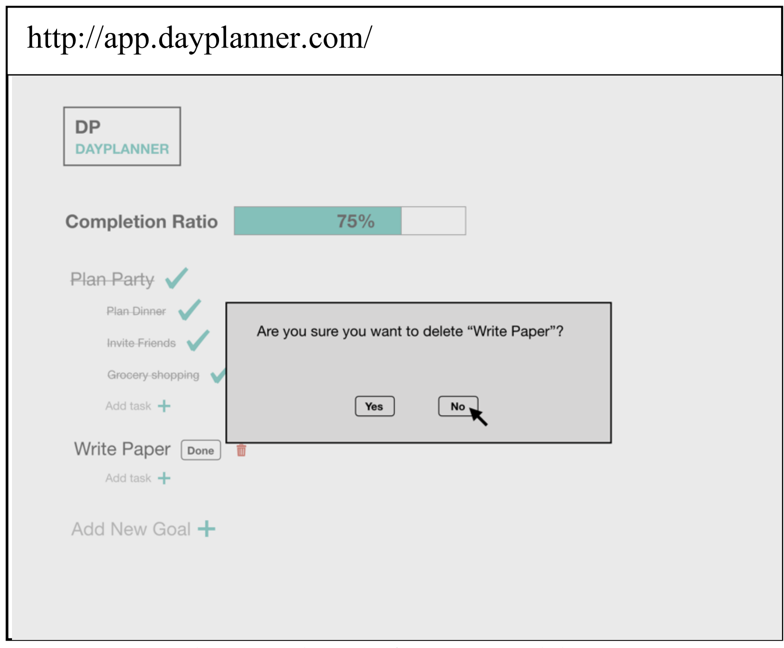


Figure 18: DayPlanner Delete Confirmation Modal

9.3 Modeling Application with IFML

To model the application described in the previous section, the strategy is first modeling small pieces of the interface, and then showing the integration between them all. Finally, the model is improved by showing the steps to refactor the model in order for it to reflect the extensions proposed for web components.

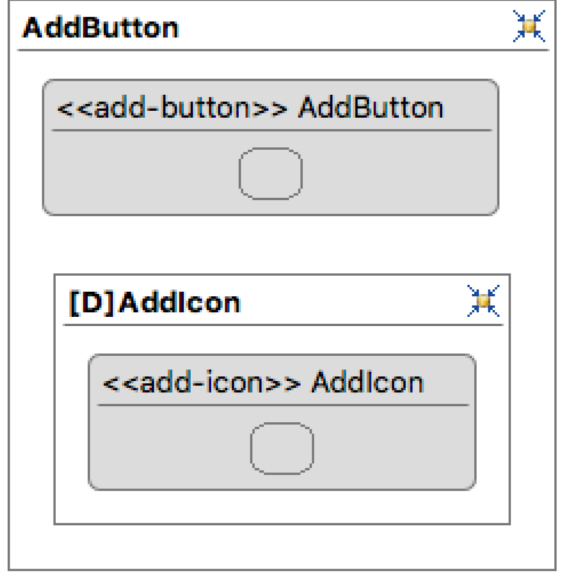


Figure 19: Add Button ViewContainer

The first element to model is the "Add New Goal" button, that is present in the initial state, and that continues to be accessible throughout the application life cycle regardless of the different states (See all figures from Figure 12 to Figure 17).

In order to model this element, a new ViewContainer is created. Notice that the view container *AddButton* has both a ViewComponent, also called "AddButton", which is identified as a custom element because of the tag <<add-button>>. This custom button also has a plus icon, which is present by default [D]. The icon is a custom element, that in alignment with the extension to IFML proposed, it is denoted by the type <<add-icon>>.

Figure 20: Add Task Action Flow

The next aspect to model is the portion of adding more tasks. Figure 20 shows the button is present. After clicking the Add Task Button, a new Text Field shows up in order to enable the user to input the new task name.

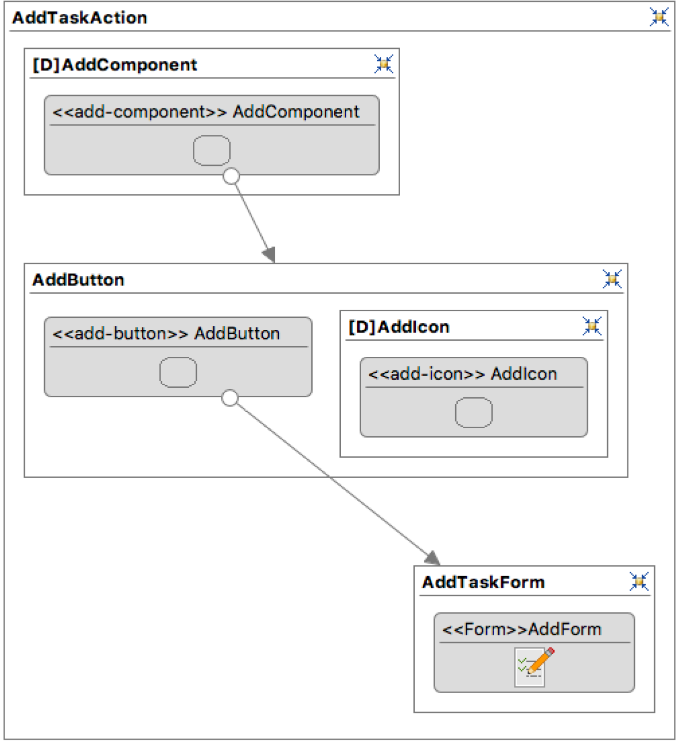


Figure 21: Modeling Add New Task Action

To model this flow, a ViewContainer called "AddTaskAction" contains both the custom button and the input field to enter the goal/task name. As Figure 21 suggests, this element contains three ViewComponents. *AddComponent* is the default ViewContainer, which contains a new custom element with the same name. This custom element will be a component that is in charge of managing both the button and the input field, which is present in a form. Notice that AddForm View Component is of type <<Form>> which is a default extension of IFML (more information in section 7.5.2.).

When *AddButton* is clicked, a new ViewContainer with a form (AddForm) containing an input field is displayed.

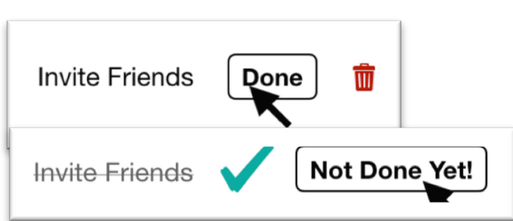


Figure 22: Two Hover States

Next element to model will be the hover states for the tasks. The two possible hover states are shown in Figure 22. Given only one of the two states is present at any given moment, the modeling wraps the two options in a XOR View Container.

As Figure 23 describes, one ViewContainer per set of actions is defined: *DoneActions*, and *NotDoneActions*. For the purpose of testing the Shadow DOM feature, these two ViewContainers define a boundary for a Shadow DOM in order to encapsulate and isolate the contents of each container. NotDoneActions contains two ViewComponents: a DoneButton, and an IconBin, which serves as the delete button. DoneActions ViewContainer only accommodates the NotDoneButton ViewComponent. Furthermore, we can wrap this model in a broader one that represents the entire task, in order to make it part of the task element, as Figure 24 describes. This ViewContainer also incorporates a *Task* ViewContainer, that contains both the Task name and a green check icon that appears when the task is marked as done (see Figure 22).

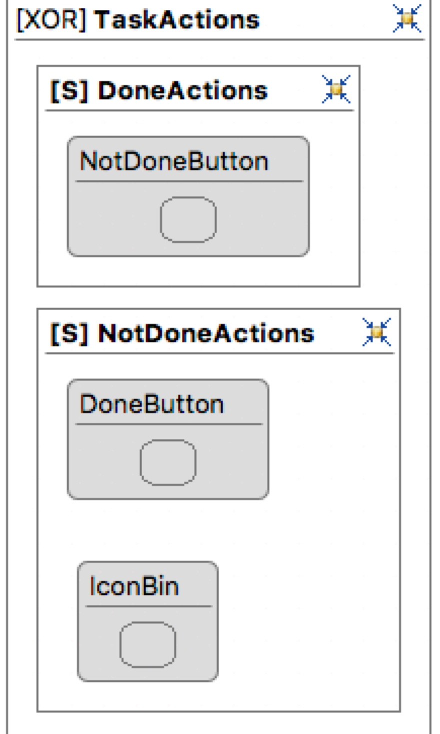


Figure 23: Modeling Task Hover States

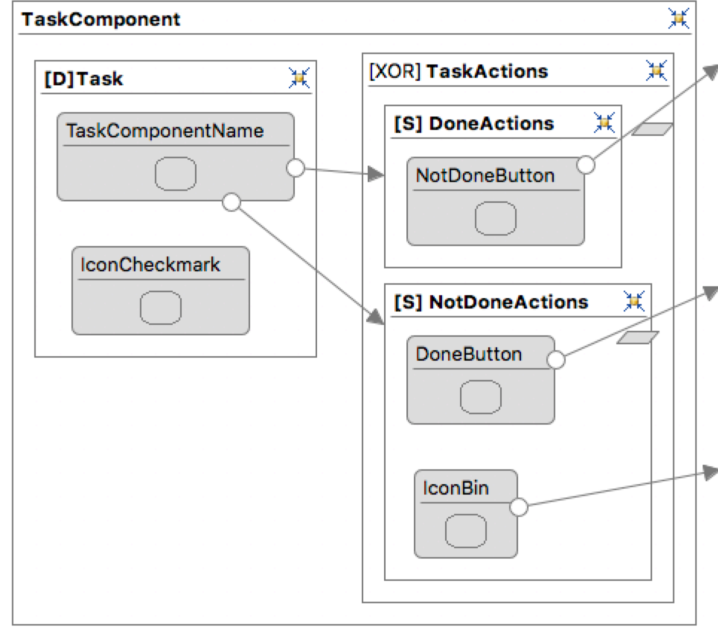


Figure 24: Modeling Task

The modeling of the completion information section, is achieved by wrapping a ViewComponent in a ViewContainer, as Figure 25 describes.

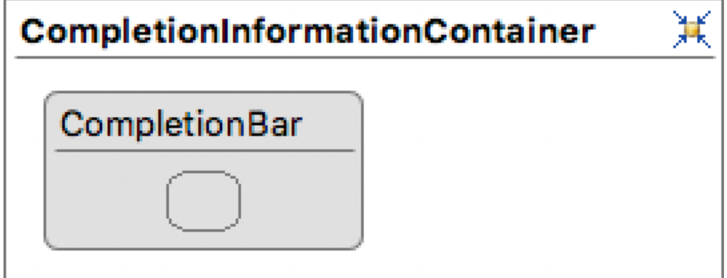


Figure 25 Modeling Completion Ratio

Finally, the model of Goals and the Goals' hover states are very similar to the Tasks and Tasks' hover states respectively. For that reason, the same modeling is reused for both. Figure 26 shows

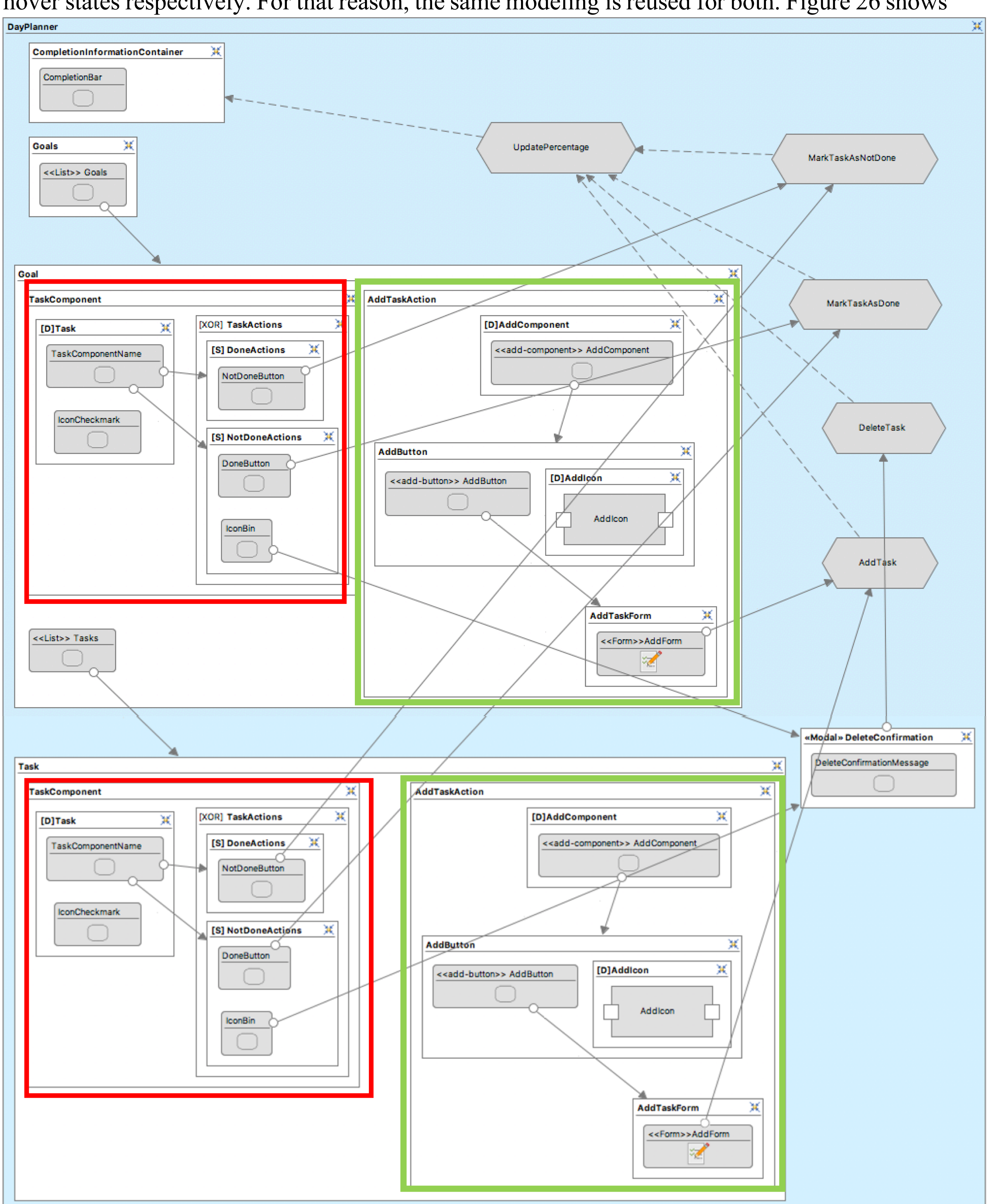


Figure 26: Application in IFML, Iteration 1

an integration of all the previous models into one single diagram. In addition, it includes a representation for the list of goals, and some flows triggered by events such as clicks or mouse over.

Notice that *TaskComponent* (highlighted in red) repeats itself for both goals and tasks. HTML Template tag provides reusability features; thus, this will be part of a refactor that will move these sections to be defined in a IFML module.

A similar situation is presented by the ViewContainer called *AddTaskAction* (highlighted in greed), that even though is present only once in the model, it is reused by the web application for each goal. Thus, is also an improvement that will be implemented through the usage of an IFML module, which will be mapped to a template tag.

9.3.1 Refactoring and Module Definitions

The module described in Figure 26 has room to improve by making use of IFML Modules. According to the proposal made by this author in section 8.2 Extending IFML, they are able to take advantage of Web Components features, since all IFML modules map to HTML Templates.

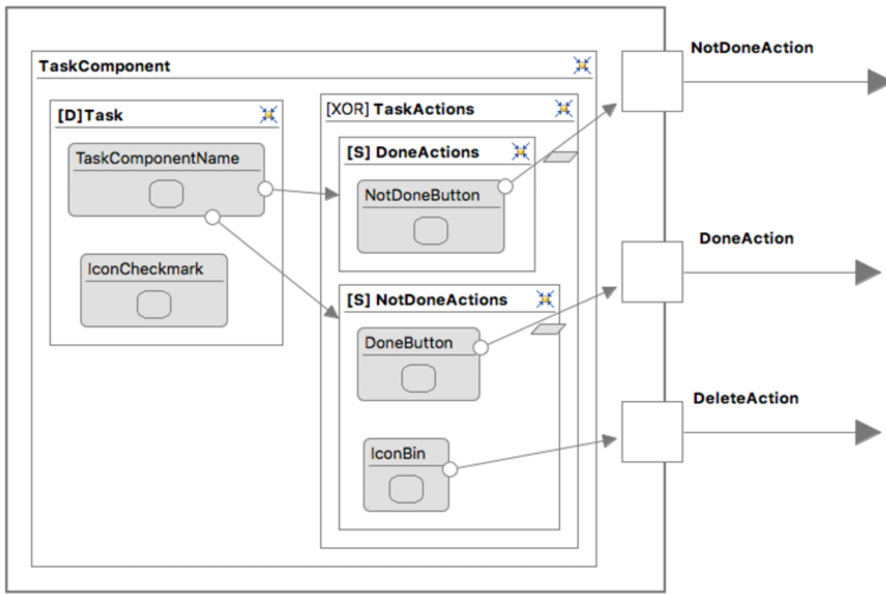


Figure 27: TaskComponent Module Definition

Figure 27 specifies the Module Definition for *TaskComponent*. Notice it also defines one output port per action performed. Figure 28 is the Module Definition that corresponds to *AddTaskAction*.

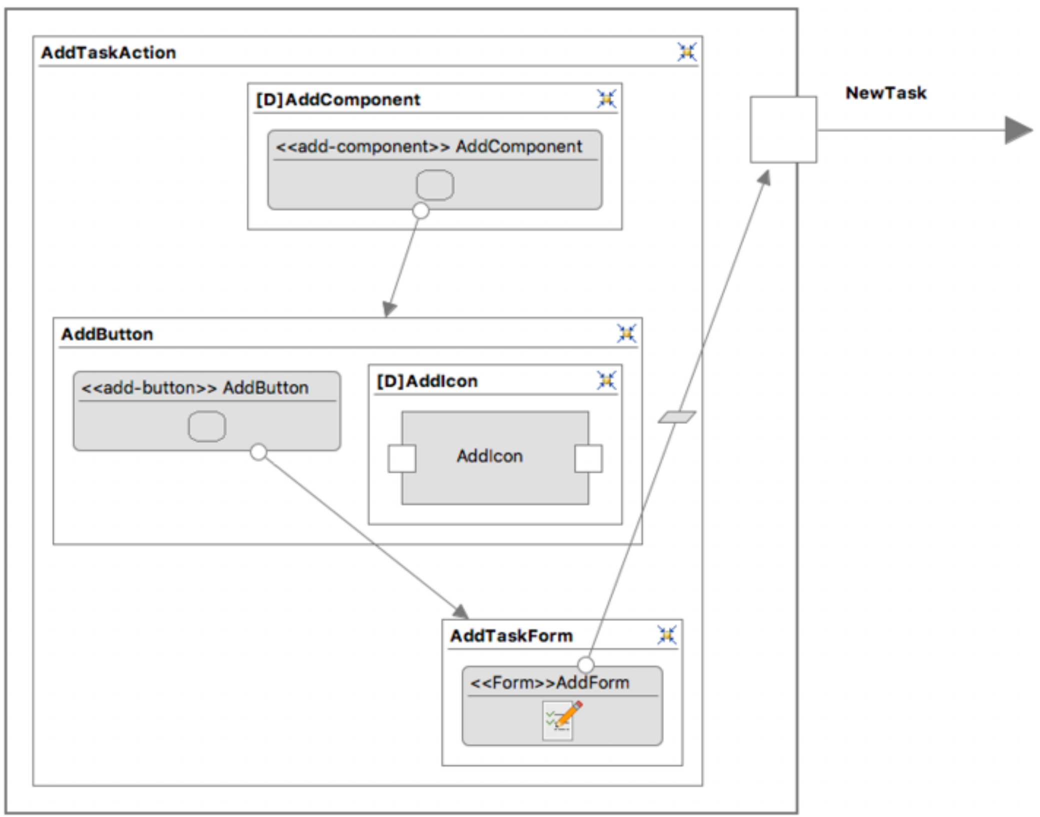


Figure 28: AddTaskAction Module Definition

The final artifact of the modeling process refers to the Module Definitions, which according to this proposal maps to HTML Templates directly. The final site view in IFML is a much more simplified more readable model. (See Figure 29: Application in IFML, Iteration 2)

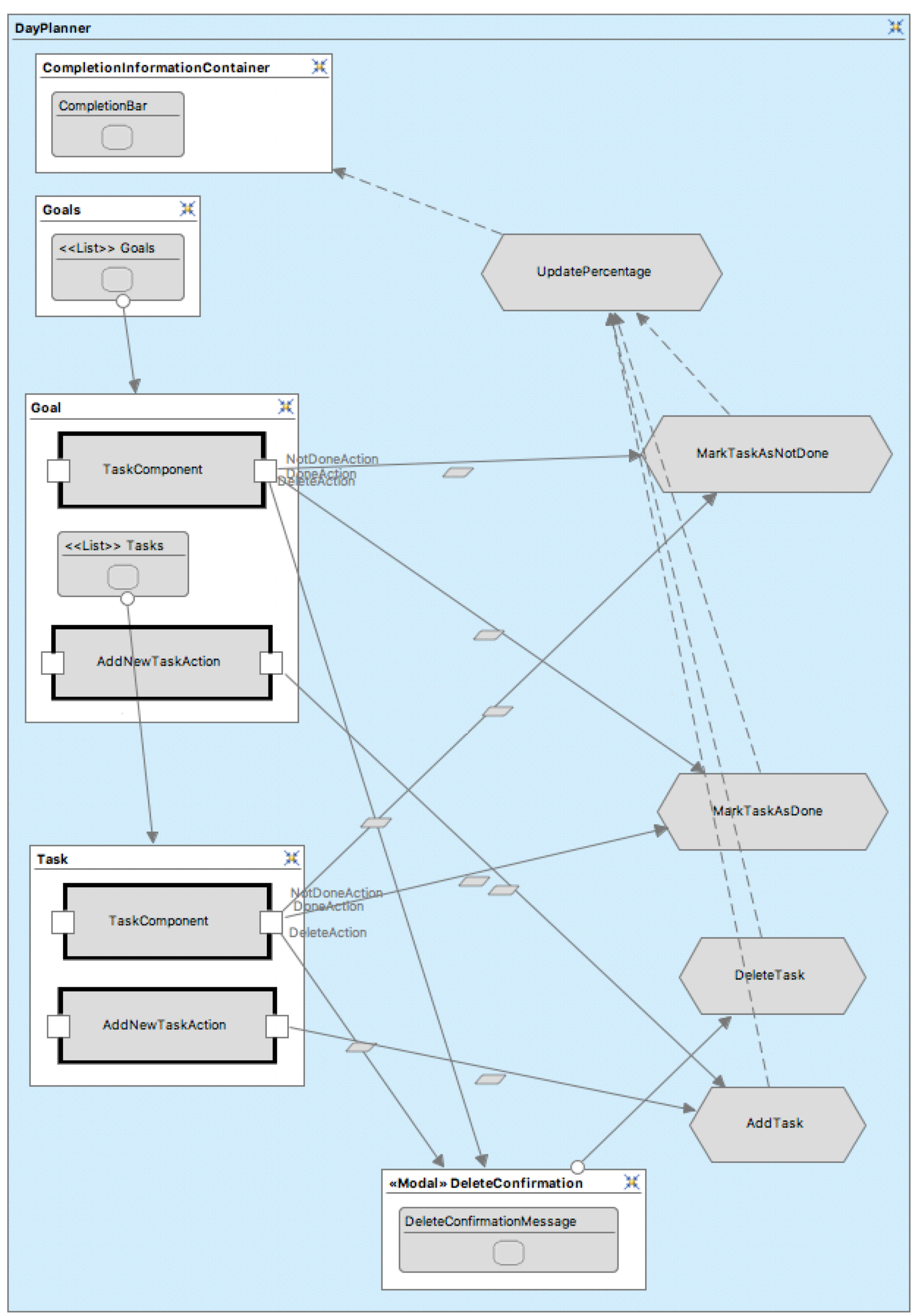


Figure 29: Application in IFML, Iteration 2

9.4 Transformation to code

The process to transform the model to code follows a process similar to the practice in the industry, according to the interviewees. They manifest the commonly use the models as “Sketches” or as “Blueprints”, according to the classification by Fowler (described in section 3.1 Purpose of modeling). For that reason, the role of the model described in Figure 29 is to be a formal guide to the coding process.

This section presents only the most relevant sections of the code transformations. However, the source code is publicly accessible in its entirety. More information in section 9.4.3 Source Code and Live Application.

9.4.1 From Specific to General

In line with Atomic Design principles (section 4.4), and in a similar order that the modeling follows, the first step is coding the smaller atomic elements of the model to later use them to build larger and more complex elements.

The first transformation to code corresponds to the IFML Modules. According to the extension proposal, all IFML Modules should map to HTML Templates. Such is the case for the “AddTaskAction” module, which contains the custom element “add-component” (Figure 30: Creation of AddComponent)

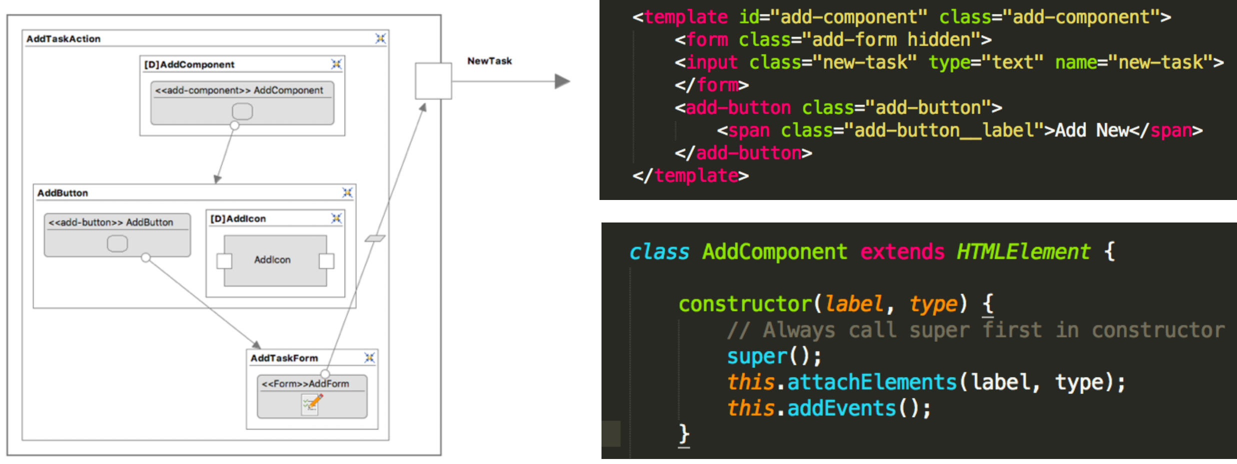


Figure 30: Creation of AddComponent

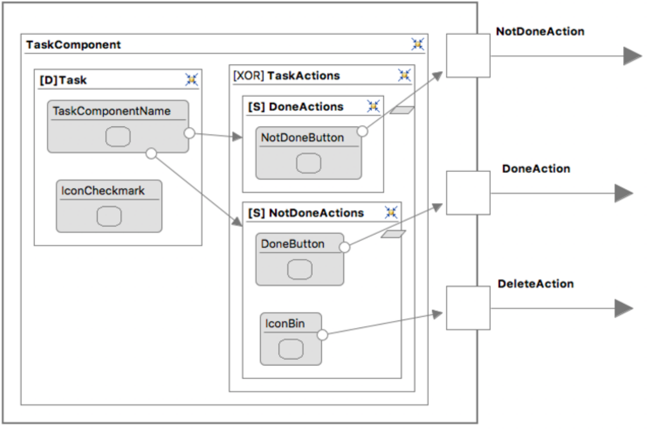
Notice that *AddIcon* is an IFML module, that according to our extension is an HTML Template in terms of Web Components, and extra logic is present in order to add it to the custom <add-button> once a new instance is created, as illustrated in Figure 31.

```
<template id="add-icon">
    <span class="icon-plus"></span>
</template>
```

```
class AddComponent extends HTMLElement {

    constructor(label, type) {
        // Always call super first in constructor
        super();
        this.attachElements(label, type);
        this.addEvents();
    }

    attachElements(label, type) {
        var ADD_COMPONENT_TEMPLATE_ELEM = d.querySelector('#add-component').content.cloneNode(true);
        this.classList.add('add-component');
        if (type == 'goal') {
            this.classList.add('big-font');
        } else {
            this.classList.add('add-padding');
        }
        this.setAttribute('data-type', type);

        this.appendChild(ADD_COMPONENT_TEMPLATE_ELEM);
        var ADD_ICON_TEMPLATE_ELEM = d.querySelector('#add-icon').content.cloneNode(true);;
        this.getElementsByClassName('add-button')[0].appendChild(ADD_ICON_TEMPLATE_ELEM);
        this.getElementsByClassName('add-button__label')[0].innerText = label;
    }
```

Figure 31: Implementing AddIcon Template

Figure 32 shows how the module *TaskComponent* is transformed to code. In the JavaScript code marked in red squares, describes the process of cloning the HTML Template to later add it to the DOM. Since this module is reused for both Goals (top level) and Tasks (Goals' children), the constructor considers certain customization depending on the input parameters.




```html
<template id="task-component" class="task-component">
    <span class="task-component__name"></span>
    <span class="icon-checkmark hidden"></span>
    <div class="task-component-actions done-actions hidden">
        <button class="done-button not-done">Not Done Yet!</button>
        <span class="icon-bin"></span>
    </div>
    <div class="task-component-actions not-done-actions hidden">
        <button class="done-button done">Done</button>
        <span class="icon-bin"></span>
    </div>
    <div class="tasks-placeholder"></div>
    <div id="button-placeholder"></div>
</template>
```

```javascript
class TaskComponent extends HTMLElement {

    constructor (params) {
        super();

        this.type = params.type;
        this.id = params.id;

        this.classList.add('task-component');

        var TASK_COMPONENT_TEMPLATE_ELEM = d.querySelector('#task-component')
            .content.cloneNode(true);

        if (params.type == 'task') {
            this.classList.add('add-padding');
            this.goalId = params.goalId;
        } else {
            TASK_COMPONENT_TEMPLATE_ELEM.getElementById('button-placeholder')
                .appendChild(new AddComponent('Add New Task', 'task'));
        }

        this.appendChild(TASK_COMPONENT_TEMPLATE_ELEM);
        this.setAttribute('data-type', params.type);
        this.setAttribute('id', params.id);
        this.setName(params.text);

        this.addEvents();
    }
```

Figure 32: Creating Task Component

9.4.2 Polyfills

Section 4.2.5 describes the necessity of employing polyfills when working with Web Components, in order to ensure cross-browser compatibility. In this Use Case, the polyfill present is "*webcomponentsjs"*, and it is publicly available in the following GitHub repository:

https://github.com/webcomponents/webcomponentsjs

The polyfill works as described in the documentation, and no extra steps are necessary.

9.4.3 Source Code and Live Application

The final code is publicly available at GitHub, and anyone can access the repository through the following URL:

https://github.com/raguilaru/dayplanner

The live application is also hosted in GitHub. The URL to access the app is:

https://raguilaru.github.io/dayplanner/

9.5 Use Case Results

The present use case presents an application design, then it describes a modeling process using the extensions introduced to IFML, and finally a brief description of how the model translates into code (web application). As a result, this use case proves some benefits, but also sheds some light into relevant concerns and challenges.

9.5.1 Benefits

- Successful Modeling of Reusable Web Components: The extensions suggested for IFML, particularly to describe *HTML Templates* and *Custom Elements*, proves to be sufficient for the scale of the use case.
- IFML extensions readability: IFML, and the extensions are easy to read, therefore easy to follow in development stages. This author considers this has the potential to also become a good mean of communicating the anatomy of Web Components across a development team.
- Clearer Roadmap: The exercise of modeling Web Components provides good clarification and a better roadmap for the development stage. This can also bring other benefits in itself, such as providing more accurate time estimations, and estimating the complexity of any given component.
- Good quality code: Due to the effort and focus in designing components in IFML, many of the problems arise during the designing stage, making the coding stage leaner and free of cumbersome hacks.
- Less time coding: In comparison to other experiences developing reusable components, this author considers that the inclusion of the model accelerates the coding process in a significant amount. Especially when complex Web Components need to interact with each other.

9.5.2 Drawbacks and Challenges

- Tooling: The process of getting a modeling software that supports the minimum required features of IFML is not a simple task. This author tested a couple of online tools, and a plugin for modeling in Eclipse, but they lack the support of many important features. In addition, most of them lack the support from the authors and do not have recent updates or improvements. The only tool available is the proprietary software *WebRatio*. This author considers this the best tool available for modeling IFML. However, it lacks sufficient online resources for beginners, which makes the process of learning the tool very time consuming.
- Not everything in the model: When coding, some elements necessary for the application are not initially present in the original model such as custom elements for the templates. Depending on the usage of the models within an organization, extra effort is necessary to keep the original Web Components models up to date.
- Implementation of Polyfills: The use of polyfills can be detrimental to a browser's performance. However, the most popular browsers do not yet support all Web Component features, which makes the usage of polyfills a necessary evil.

9.5.3 General considerations

When reusable web components are designed, their styling requires a very careful planning. This is because a component can visually vary a lot depending on the properties or parameters of the component, which all need to be considered since the modeling stages. According to the interviews this a common problem when reusing components, and it is also present in the present use case in the *TaskComponent*, where one style or another is applied depending on the type of task (either goal or simple task).

9.6 Limitations of the Use Case

According to the information drawn from the interviews, all participants practice different forms of agile methodologies for web development. To apply any of the extensions proposed in this thesis, transformations should be created in order to create executable models Stephen J. Mellor, as discussed in section 3.2.7. This can consume a high amount of resources, that may or may

not be significant depending on the scale of the project. However, once in place, other projects can reuse the same tools to transform the models into executable code.

This work is limited to modeling elements present in the client side. Web components specifications’ scope is limited to front-end development, and it does not describe any interaction with the server.

10 Discussion

According to the literature survey, there are different approaches for modeling: some practitioners believe in short development cycles, where lightweight models are the answer to quickly move to the development stage. Others support an intensive modeling scheme, where models are executable artifacts themselves, or can be automatically transformed into code, by the implementation of automation tools. However, the approach towards modeling is not a binary choice, but it consists of a wide spectrum, depending on the purpose for modeling. Martin Fowler classifies these purposes into three categories: modeling as sketches, which are often informal; modeling as blueprints, or as formal guidelines for programming; and finally, as executable models, automatically mapping models into executable applications.

Interaction Flow Modeling Language (IFML) is currently the most complete effort in attempting to model visual aspects of web applications. It is based in a previous modeling language called WebML, but includes many features that makes IFML a technology-agnostic language.

IFML's extensibility is a key aspect that allows practitioners to make customizations for specific technologies. There is good evidence in the literature about its applicability to adapt to different scenarios such as its flexibility to model, for instance, mobile applications' user interfaces. This is the main argument that justifies its usage, and extend its existing visual elements to include features that are specific to Web Components standards and specifications.

The interviews with experts in Front-End Engineering suggest us that the most common purpose of modeling in this field is as informal sketches, and also very often as blueprints, but never as executable models. They take advantage of models mainly for describing and communicating certain web components with co-workers, to aid the designing process of relatively complicated components, and as guidelines for the coding process.

The participants report the usage of tools that aid the coding process by generating very specific pieces of the application related to styling. However, the resulting code is not based on models, but on visual guides provided by graphic designers. Based on the interviews, it is fair to claim that modeling as a programming language is not a standard practice in client-side development.

As section 7.6.2 describes in greater detail, IFML and the extensions suggested in the present work, do not attempt to describe graphic design aspects of the application, since they are antagonistic to modeling. This means that is a virtually impossible task to automatically transform all aspects of Web Components into code. However, based on the information provided by interviewees, this author considers this problem can be addressed by applying tools that generate styling code based on graphic design documents rather than models.

The extension proposed in this thesis proves to be effective in a small-scale application like the one present in the section 9 “Use Case: DayPlanner Application”. The two features this author finds more effective are: the extension to map IFML modules into HTML Templates, and the one mapping a custom View Component into HTML Custom Element.

The extension created to isolate code through the usage of Shadow DOM does not contribute in any relevant manner to the Use Case described. This author suspects this is due to the reduced scale of the application. However, there is evidence that the problem Shadow DOM tries to address does not represent a serious problem in larger scale web applications either, since the interviewees consider “Isolation of pieces of code” the least challenging problem of the list when it comes to developing reusable web components.

Although HTML Imports is part of Web Components specification, it is the feature that causes the more technical debate among browsers’ developers and standards authors, since many argue that it clashes with ECMAScript’s approach to modularization and packaging. This author agrees with that claim and considers that it is a compelling reason to leave it out of the proposed extensions to IFML.

All the new extensions are easy to follow as blueprints for programming. Assuming the developer is already familiar with IFML, these extensions do not represent a new paradigm, but an addendum to the standard concepts of the language.

IFML and its extensions prove to be of great benefit in small-scale application scenarios such as the one featuring in the Use Case. Some of the advantages this author identifies are: the

extensions are easily understandable and readable, the model helps clarifying a development roadmap, the code as the result of following the model is of high quality, and development time is short. These results are in line with the results of the interview, in which participants state that reducing development time and delivering high quality code is enough justification for introducing models as part of an agile development process.

The usage of modeling that the Use Case describes, is one where the model designed serves the purpose of a forma guidelines for coding Web Components. It successfully addresses two scenarios practitioners describe in the interviews as challenges: communication across the development team and the high complexity of defining a standard modeling language.

Even though IFML is part of Object Management Group's (OMG) toolset, it lacks the support from software vendors, which makes its use very difficult to promote in comparison to other OMG's standards such as UML, which has almost universal support by IDEs, modeling and diagramming software vendors.

WebRatio is the only IFML modeling tool that is ready for production use. However, this author considers it far from ideal for beginners, since its high complexity and not very comprehensive documentation is a barrier that needs consideration.

10.1 Recommendations for Future Research

The extension of IFML proposal needs further testing in the context of larger web applications and several team members to evaluate maintainability and scalability.

Even though the design of the IFML extensions present in this work aim to support features of standard Web Components, more tests are necessary to test the ability that these extensions have to also cover important features of frameworks based on web components such as React, Angular or Vue.

To achieve Model Driven Development with IFML and the extensions, it is necessary the development of automatic model to code transformation tools. This author believes that such

tools have the potential to dramatically improve the practice of web components development in the context of agile processes.

Future research can focus more on behavioural aspects, that can include latest specifications for ECMAScript and interactions between components. IFML already provides flows, which this author believes can be extended in order to model different type of interactions and flows of information.

The future creation of open tools that support IFML standard features, that also support custom IFML extensions, is crucial for the applicability of the tool proposed.

This author sees an opportunity to research IFML and its extensions in the context of critical systems, in which client-side components can be formally describe to ensure people's lives and security.

10.2 Conclusions

Existing work in modeling web applications - Research Question 1

The first research question aims to collect the existing knowledge and experience in modeling web. As a conclusion, this author finds very valuable work in the efforts for developing tools such as Web Application Extensions for UML, WebML, and Interaction Flow Modeling Language (IFML).

IFML is the most recent effort and the one that covers in a more effective way the needs of the modern web, by being technology-agnostic.

There is literature that proves IFML to be a flexible language for representing user interfaces, since it is capable to adapt to technologies such as Mobile applications, thanks to its extensibility features.

Identification of challenges to model Web - Research Question 2

The second research question attempts to identify the challenges when it comes to modeling front-end Web Components. According to the literature, the most important challenge is: graphic design aspects are antagonistic to modeling, which means that is impossible to model absolutely all aspects of a Web Component in a model. In order to solve it, the interviewees resort to tools that generate styling code, taking graphic design documents as input. However, this is an answer to the practical problem of representing graphical design aspects rather than modeling them.

According to the interview participants, the most difficult challenge for modeling reusable components is to find a proper language that can effectively model components in a way that covers all possible scenarios and reusability cases.

Another challenge, also suggested in the interviews, is justifying the usage of models in the context of an agile process, where a reduction in development speed can justify its inclusion as an integral part of developing Web Components.

How to model Web Components - Research Question 3

Finally, the last research question aims to find a solution for effectively modeling Web Components. Considering the challenges present in the previous research question, this author concludes that the best solution is extending IFML, because it is an already mature language that is technology-agnostic and that encourages extensibility to cover technology specific aspects.

In order to model Shadow DOM, an additional IFML View Container identified as [S], easily identifies containing elements that need to be isolated from the rest.

To represent HTML Templates, IFML’s modules prove to be ideal, because they provide a similar function of defining encapsulated units in order to reuse it when necessary.

New types of IFML View Components effective for mapping HTML Custom Elements.

According to evidence found in the interviews and in the Use Case, impact of modeling a Shadow DOM in a web application might be low.

Even though HTML Import is part of Web Components, its modeling is not relevant, due to conflicts with new ECMAScript specifications in terms of handling modules and packaging functionality.

Modeling Web Components by implementing IFML and the extensions proposed in this work reduces the development time and promotes high quality code.

Although their implementation is not ideal, polyfills are a reasonable solution for cross-browser compatibility with the latest standards.

Even though Web Components specifications are a standard proposition, the reality is that most solutions for component-based front end follow alternative solutions such as third-party frameworks such as React or Vue.

11 References

Acerbis, R. *et al.* (2007) "Developing eBusiness solutions with a model driven approach: the case of acer EMEA," in *International Conference on Web Engineering*, pp. 539–544.

Ambler, S. W. (2004) *The object primer: Agile model-driven development with UML 2.0*. Cambridge University Press.

Ambler, S. W. (2014) *Be Realistic About the UML*. Available at: http://agilemodeling.com/essays/realisticUML.htm (Accessed: January 20, 2018).

Beck, K. *et al.* (2001) "Manifesto for agile software development."

Bell, A. E. (2005) "UML fever: diagnosis and recovery," *Queue*. ACM, 3(2), pp. 48–56.

Beydeda, S. *et al.* (2005) *Model-driven software development*. Springer.

Bidelman, E. (2013) *HTML Imports*. Available at: https://www.html5rocks.com/en/tutorials/webcomponents/imports/ (Accessed: May 20, 2018).

Brambilla, M. (2017) *Model-Driven Software Engineering in Practice : Second Edition.* 2nd ed. Edited by J. Cabot, M. Wimmer, and L. Baresi.

Brambilla, M. and Fraternali, P. (2014) *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann.

Ceri, S., Fraternali, P. and Bongio, A. (2000) "Web Modeling Language (WebML): a modeling language for designing Web sites," *Computer Networks*. Elsevier, 33(1–6), pp. 137–157.

Codesido, I. (2009) *What is front-end development?* Available at: https://www.theguardian.com/help/insideguardian/2009/sep/28/blogpost (Accessed: April 18, 2018).

Connor, J. O. (2012) *Pro HTML5 accessibility*. Apress.

ECMA (2018) *ECMAScript® 2018 Language Specification*.

Fowler, M. (2003) "A Brief Guide to the Standard Object Modeling Language." Qinghua University Press.

Frost, B. (2016) *Atomic Design*. Brad Frost.

Glassdoor (2018a) *Front End Web Developer Salaries*. Available at: https://www.glassdoor.com/Salaries/front-end-web-developer-salary-SRCH_KO0,23.htm (Accessed: July 16, 2018).

Glassdoor (2018b) *Web Developer Salaries*. Available at: https://www.glassdoor.com/Salaries/web-developer-salary-SRCH_KO0,13.htm (Accessed: July 16, 2018).

Glazkov, D. (2011) *What the Heck is Shadow DOM?* Available at: https://glazkov.com/2011/01/14/what-the-heck-is-shadow-dom/.

Hennicker, R. and Koch, N. (2001) "Modeling the User Interface of Web Applications with UML.," *pUML*, 7, pp. 158–172.

House, C. (2015) *HTML5 Web Components: Solution to the Div Soup?*, *Pluralsight*. Available at: https://www.pluralsight.com/blog/software-development/html5-web-components-overview (Accessed: July 15, 2018).

IFML Specification (2015). Available at: http://www.omg.org/spec/IFML/1.0/ (Accessed: February 21, 2018).

ITU (2017) *ICT Facts and Figures*, *International Telecommunication Union*. Available at: https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2017.pdf.

Jobs, S. (2010) *Thoughts on Flash*.

van Kesteren, A. (2014) *Mozilla and Web Components: Update*. Available at: https://hacks.mozilla.org/2014/12/mozilla-and-web-components/ (Accessed: June 14, 2018).

Kleppe, A. G. *et al.* (2003) "The model driven architecture: practice and promise." Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Leenheer, N. (2015) *HTML5 - How well does your browser support html5?*, *Html5Test.Com*. Available at: https://html5test.com/results/desktop.html (Accessed: April 27, 2018).

Marco Brambilla, Mauri, A. and Umuhoza, E. (2014) "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End," in *International Conference on Mobile Web and Information Systems*.

Mavridis, T. and Symeonidis, A. L. (2015) "Identifying valid search engine ranking factors in a Web 2.0 and Web 3.0 context for building efficient SEO mechanisms," *Engineering Applications of Artificial Intelligence*. Elsevier, 41, pp. 75–91.

Mavrommatis, N. P. P. and Monrose, M. A. R. F. (2008) "All your iframes point to us."

Mellor, S. J. (2004) "Agile mda," *MDA Journal, www. bptrends. com June*. Citeseer.

Mellor, S. J., Clark, A. N. and Futagami, T. (2003) "Model-driven development - Guest editor's introduction," *IEEE Software*, 20(5), pp. 14–18. doi: 10.1109/MS.2003.1231145.

Mills, C. (2018) *Web Components*. Available at: https://developer.mozilla.org/en-US/docs/Web/Web_Components (Accessed: April 27, 2018).

ModuleCounts (2018) *Module Counts*, *modulecounts.com*. Available at: http://www.modulecounts.com (Accessed: July 16, 2018).

Snyder, A. (1986) "Encapsulation and inheritance in object-oriented programming languages," in *ACM Sigplan Notices*, pp. 38–45.

The MDA Foundation Model, an ORMSC DRAFT (2010). Available at: http://www.omg.org/cgi-bin/doc?ormsc/10-09-06 (Accessed: February 15, 2018).

Torres, A. *et al.* (2018) *Using shadow DOM*, *2 April*. Available at: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

(Accessed: May 1, 2018).

W3C (2017) *HTML 5.2 W3C Recommendation*. Available at: https://www.w3.org/TR/html52/introduction.html#background (Accessed: July 16, 2018).

W3C (2018a) *Custom Elements, Working Draft*. Available at: https://www.w3.org/TR/custom-elements/ (Accessed: April 21, 2018).

W3C (2018b) *HTML Imports, Editor's Draft*. Available at: https://w3c.github.io/webcomponents/spec/imports/ (Accessed: May 17, 2018).

W3C (2018c) *Shadow DOM*, *1 March*. Available at: https://www.w3.org/TR/shadow-dom/ (Accessed: May 1, 2018).

WebComponents.org (2018) *Polyfills*. Available at: https://www.webcomponents.org/polyfills (Accessed: June 14, 2018).

WebKit (2018) *WebKit Feature Status*, *2018*. Available at: https://webkit.org/status/#feature-html-imports (Accessed: June 14, 2018).

WebRatio (2014) *Custom enterprise applications in a third of the time*. Available at: http://www.webratio.com/casestudies_pdf/WhitePaper_WebRatio_EN.pdf.

West, M. (2016) *How to Create Custom HTML Elements - Treehouse Blog*, *May 23rd*. Available at: http://blog.teamtreehouse.com/create-custom-html-elements-2 (Accessed: May 15, 2018).