DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automatically extracting view models from component-based web applications

Andreas Tielitz

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automatically extracting view models from component-based web applications

Automatisches Extrahieren von View-Modellen aus komponentenbasierten Web-Anwendungen

Author:	Andreas Tielitz
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	M. Sc. Adrian Hernandez-Mendez
Submission Date:	15.12.2016

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, 15.12.2016 Andreas Tielitz

Acknowledgements

I want to thank Prof. Dr. Florian Matthes for giving me the opportunity to write the thesis at the Software Engineering for Business Information Systems chair. I am deeply grateful to M. Sc. Adrian Hernandez-Mendez for his advice and encouragement during the development of this thesis. His valuable input and guidance helped me numerous times and made this thesis possible in the first place.
I would also like to thank my dear friend Lili Hadsell and Johann Arendt for their valuable contributions and ongoing support.

Abstract

Reverse engineering of applications has become an integral part of today‘s development. Most tools focus on addressing the needs of retrospective model extraction of business and presentation logic, while neglecting the user interface as a whole. Developing a reverse engineering process targeted at the presentation layer faces different criteria and requirements. The platform dependence of the layer requires a narrower specialisation and definition of the supported scope.

The thesis explores challenges and requirements of a view model and how modern web user interfaces can be represented. A tool supported reverse engineering process was developed alongside to provide an example implementation. The library React and the frameworks AngularJS and Polymer were selected as the initially supported choices for the extraction tool. The selection was made based on the popularity and support of a component-based architecture. The suitability of the extraction tool was assessed through evaluation of real-world projects.

Contents

1. Introduction

In today's world, the development of web applications is faster paced than ever. New frameworks and technologies emerge continuously, bringing in new concepts and new approaches to solving common problems. In addition, new presentation devices are being constantly developed, requiring new and innovative user interfaces to be developed for software systems.

A modern web-application can be divided into two parts: The user interface and a logical computation part. The user interface is highly dependent on the environment in which the software is running. An interface for a desktop computer looks different than a mobile device or a watch, for example. The latter computation part, on the other hand, does not necessarily need to change to the same extent, as the provided information usually remains the same. This creates a challenge for the developers. Logical constructs and their architecture are usually well documented in class, sequence or architecture diagrams and the reverse engineering is widely supported by numerous modelling tools. The part of the software component which remains constant across multiple environments is the most supported and understood, while the user interface, in general, is neglected. Understanding the structure and composition of user interfaces and their contained elements can provide valuable insight into the system as a whole. It provides insight into the structure and the contained information of each view. Reverse engineering of the interface and extracting the underlying model helps developers in understanding a system as a guide for adapting them to new environments or as a point of reference for software evolution. Instead of starting from scratch, the essential elements are extracted and displayed in a structured way. Not only does it make the development easier, it also reduces the needed cost and time.

1.1. Problem Description

Component-based web-applications lack the support of reverse engineering tools aimed at extracting the view model. Common modelling tools provide possibilities to extract classes and methods, but hardly any can effectively interpret the hierarchy of components in the HTML document or their dependencies between each other. Automatically extracting information and presenting it to a developer assists the development and evolution of newer systems. The developers should be assisted by an extraction tool which is capable of visualising the structure of a user interface in a view model without any additional interactions. The tool should be capable of analysing a whole software project and creating a tailored view model for a web-application. Three of the most common JavaScript frameworks and libraries should be supported: React, AngularJS and Polymer.

1.2. Research Questions

The ultimate goal of this research is to provide a tool which is capable of automatically extracting a view model for a project developed in one of the supported JavaScript frameworks. In order to do so, it is necessary to first determine what a view model looks like. Given a definition, certain steps must be considered before a model can be constructed from source code. Afterwards, a tool which utilises the research methods and procedures should be developed and evaluated.

1. What is a view model of a web component?
2. How can a model be extracted from existing web components code snippets?
3. How do existing technologies support the model extraction process?

1.3. Research Method

The research for the extraction tool is based on the design science research methodology described by Alan, Hevner, March, et al. [Ala+04]. By structuring the framework into the pillars *Environment*, *IS Research* and *Knowledge Base* as shown in Figure 1.1, a clear separation of the inputs and interactions provide valuable insight into the research approach.

The environment is composed of the people, organisations and technologies which define the problem at hand. Their goals, tasks and problems outline needs that must be met by the conducted research. Given that correlation, the research becomes relevant, if the research is targeted at one or more needs.

The research itself is conducted in two processes. The behavioural science focuses on developing theories and justifying their results, while the design science is targeted at creating artefacts and evaluating their capabilities to meet business needs. Both processes are relevant for the research conducted at hand. First, the theoretical foundation must be developed and their applicability and correctness justified. Only then can a specific tool, or artefact, be built which is based on theoretical foundations.

The knowledge base provides the fundamentals for conducting research. Existing research of methods and processes provide a framework of useful additions upon which one's own research can be built upon or enriched. The methodologies of the knowledge base also provide boundary conditions for justification and evaluation of the conducted research.

The outcome of the research loops back to the environment and knowledge base. The business needs are met by the research and new needs or problems can arise from the altered landscape. The research and its results are incorporated into the knowledge base and can be reused and built upon by future research.

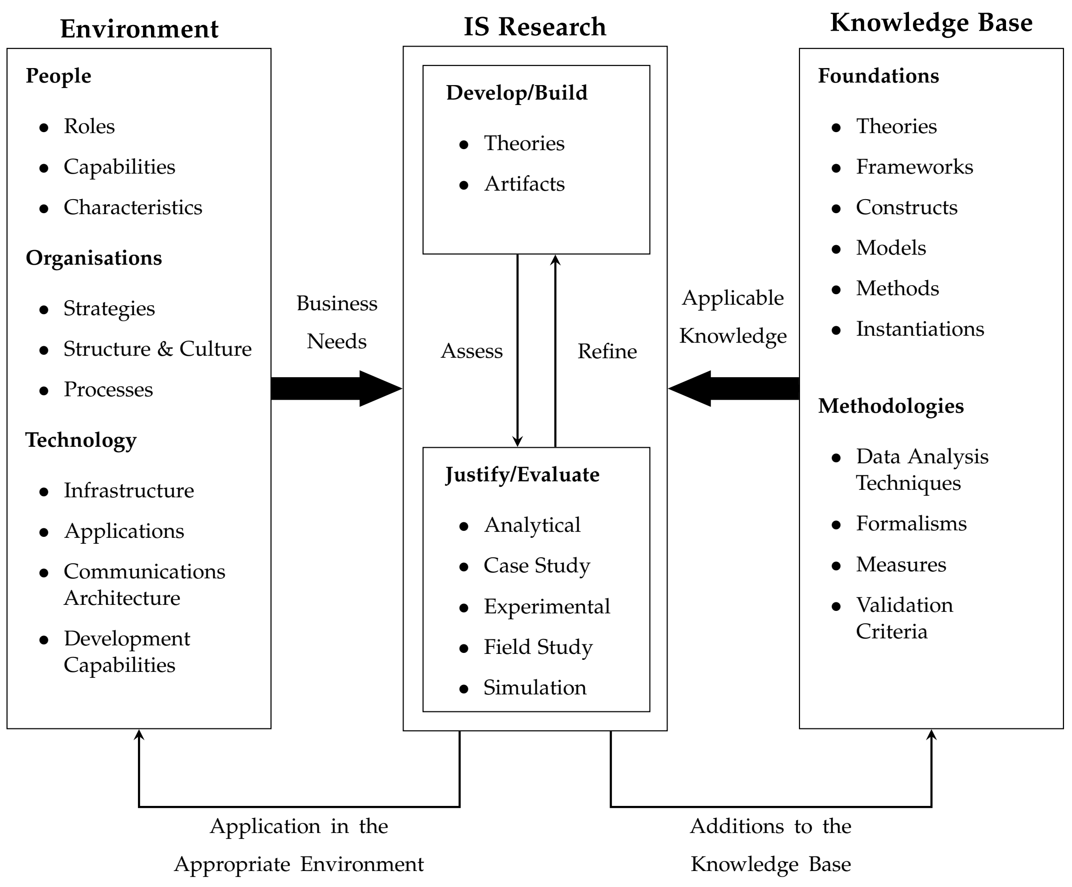


Figure 1.1.: Research framework based on Design-Science paradigm [Ala+04]

2. Conceptual Design

2.1. View Model

The structure, layout and contained elements of a user interface are part of the view. The abstraction of a view is described by the view model. Ideally, such a model can be used to describe any arbitrary view in any environment. A draft to achieve vendor-neutral and standardised representation of view models, called User Interface Modeling Language (UIML, [OAS08]), was designed by the Organization for the Advancement of Structured Information Standards (*OASIS*). The elements of the model in UIML are declared using XML. Figure 2.1 shows a visualisation of the general structure of an example UIML document. Structure, style, content and behaviour compose each interface definition. An interface can be presented in multiple ways depending on the execution environment. Each interface, is connected to elements that provide logic and connections to data sources.

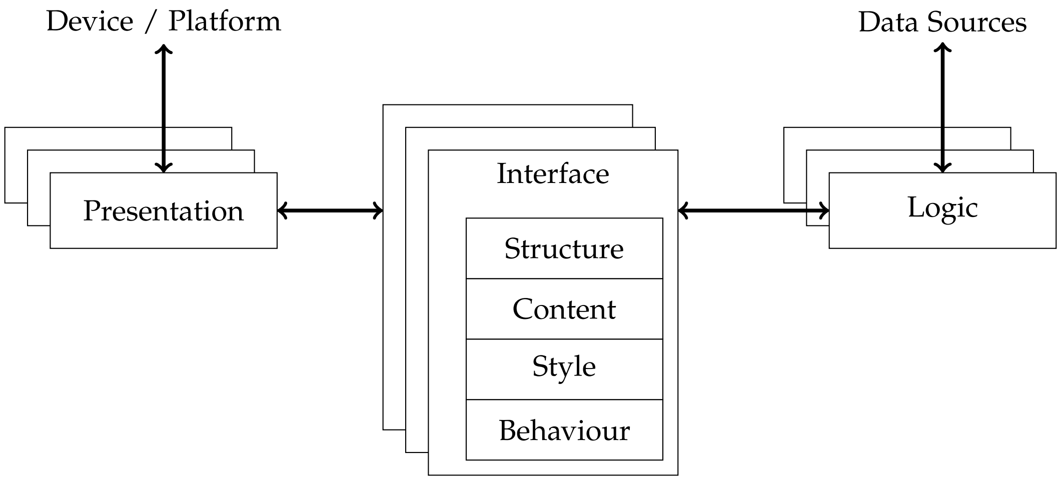


Figure 2.1.: Separation of concerns in UIML [OAS08]

One of the goals of *UIML* is to provide a generic approach to defining user interfaces. Some of the supported functionality is unnecessary, overly complicated or not compatible with elements of a modern JavaScript component-based web application. The focus of UIML lies on supporting and describing a wide variatey of interfaces. Concepts like reusable elements do not exist in the specification. Reused elements are redefined for each occurence instead of referenced, for example. Due to those limitations, a more streamlined and simplified adoption of the standard was required. As a representation, the simpler JSON format was chosen instead of XML. The derived view model was tailored to the task of representing existing component-based views in JavaScript applications. One of the things omitted from the model was the support for representations of non-browser based visualisations. The derived view model on which the extraction process is based on is shown in Figure 2.2. The generic term *interface* was replaced by *component* and the contained elements in the structure, content, style and behaviour were simplified.

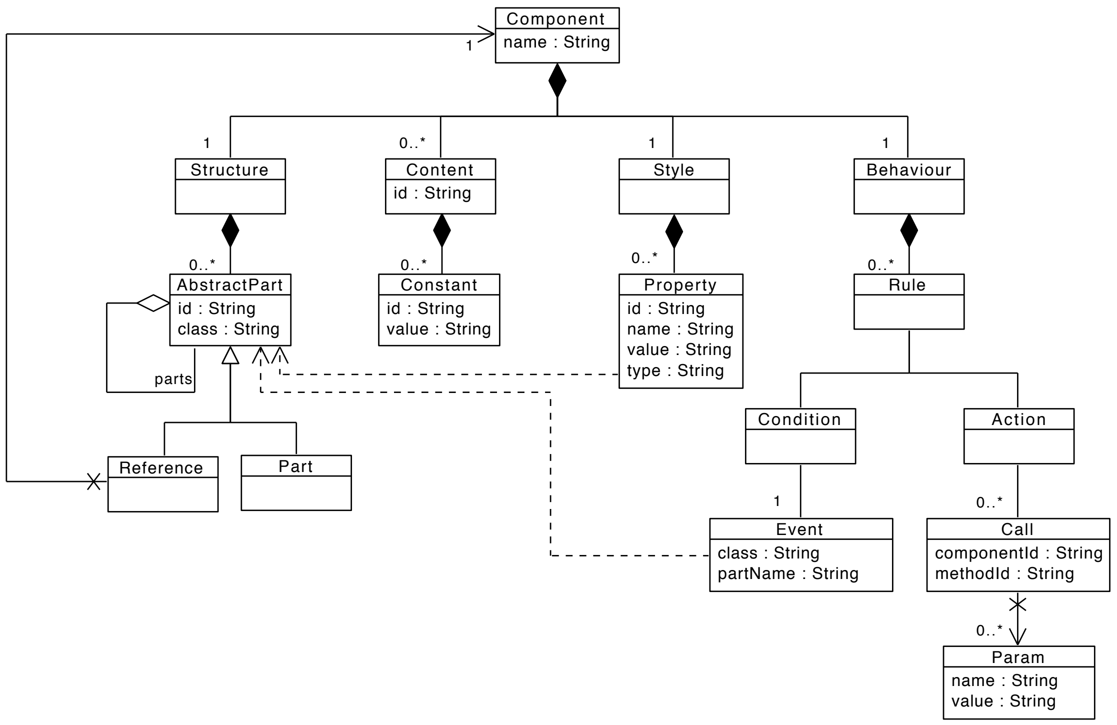


Figure 2.2.: View model adaption of UIML

2.1.1. Structure

Elements of a structure represent elements on the page. An element can either be used as a content element to display information, act as an HTML element or reference to another component. In order to support the multi-level hierarchy of HTML elements, a self-referencing relation is used. Each `AbstractPart` contains a list of child parts (Figure 2.3).

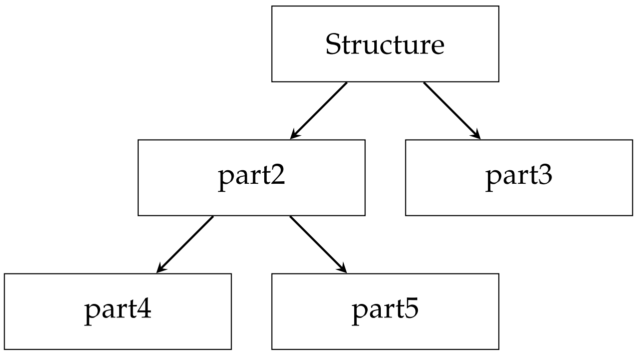


Figure 2.3.: Hierarchy of structural elements in UIML

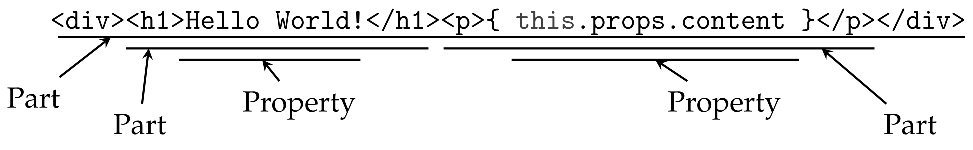
The official UIML standard allows for nesting of components without restrictions. A child component could be declared inside a part of the structure, which contains its own structure, content, behaviour and style. In the proposed simplified view model each component has its own independent model and is only referenced with a unique component type and id. The possibility and support of nesting components is omitted.

2.1.2. Content

Information presented in the component is modelled as a content element. Each entry is stored in its own *Constant* entry. UIML designed the *Content* element to be used to specify information in multiple languages and thus providing internationalisation capabilities. The *Constants* are referenced by *Properties*, which define in which *Part* the information is located.

2.1.3. Style

Connections between parts and content elements are defined with the *Property* element of the style. A property can be a key-value object or a reference to a *Constant*. UIML, in theory, allows for even more use cases which exceed the required functionality for modelling web applications.



```
<div><h1>Hello World!</h1><p>{ this.props.content }</p></div>
```

Figure 2.4.: Example of a property element in the view model

Figure 2.4 shows an example of how parts of a structure and style properties are combined. The structure itself contains a single direct child part `<div>` which in turn contains the parts `<h1>` and `<p>`. The text contents of the `<h1>` tag are modelled as a property referencing the `<h1>` part. The variable output in the `<p>` tag is stored as a property with the variable name as its value.

2.1.4. Behaviour

The events during the life cycle of a component are described in the behaviour. Aside from custom behaviour, like a mouse click or submitted form, the behaviour also describes the specific life cycle events and transitions of a component. An event contains two parts: A trigger (or condition) and an action to be executed. Both are modelled as a rule in the view model.

A JavaScript web component passes through different stages of its life cycle. Each stage serves a different purpose, for example indicating if the component was created or destroyed. These stages are part of the Behaviour model. User defined actions at any of these life cycle stages are automatically part of the same event if executed in conjunction with a life cycle event. These actions are triggered by a component-wide event and are not limited to a specific part of the structure.

User triggered events in the interface, on the other hand, are often times restricted to a certain part. For example, a button can define an on click event which triggers a form submit. These events are restricted to a specific part in the structure and would not fire if the event is triggered on another target.

2.2. Component Life Cycle

2.2.1. WebComponents

The WebComponents standard[1] attempts to define common tasks and interactions of web components. Other frameworks and libraries, as React, AngularJS or Polymer, partially implement the functionality described in documents provided by WebComponents. It can be used as a base reference to compare the different tools.

In the WebComponents documentation, the term element is used to describe individual parts of a web application. For consistency with other chapters, the term *element* is replaced with *component* in the thesis. According to the documentation for custom component reactions[2], a component provides five distinct callback methods which a developer can take advantage of. The transitions between those callbacks are shown in Figure 2.5.

[1]http://webcomponents.org/

[2]http://w3c.github.io/webcomponents/spec/custom/#custom-element-reactions

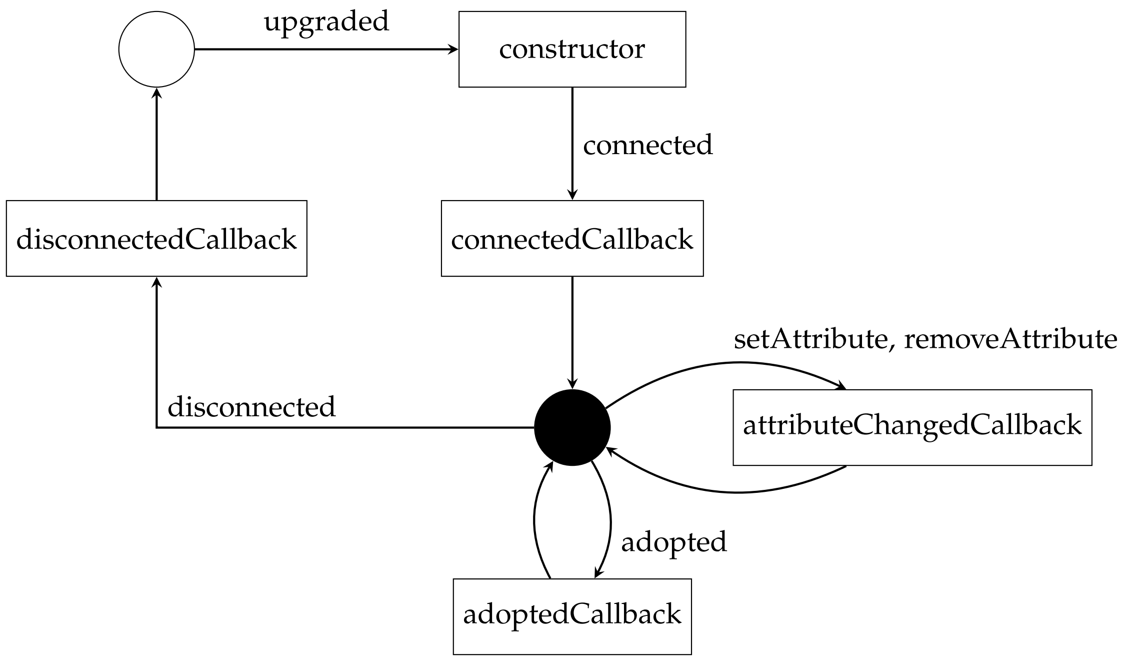


Figure 2.5.: Component life cycle events as defined by WebComponents

Constructor

The constructor should be used to handle the initial set-up of the component. This involves setting the initial state, default attribute values and event listeners. At this point, the component has not yet been attached to the document object model (DOM). Additionally, this callback is only called once in the lifetime of a specific component.

connectedCallback

This callback is executed whenever the component passes through the insertion steps outlined in the DOM Living Standard[3]. These steps describe the necessary actions that need to be performed when an HTML element is inserted as a child inside the DOM. The callback is executed once the component becomes connected. It is possible for this method to be executed more than once in case the component is inserted multiple times.

attributeChangedCallback

Each alteration to an attribute triggers the attributeChangedCallback. Based on the documentation of WebComponents[4], the callback is triggered whenever an attributes is changed, appended, removed or replaced. Each component should implement the method `observedAttributes` which returns a string array with desired observed

[3]https://dom.spec.whatwg.org/
[4]http://w3c.github.io/webcomponents/spec/custom/#custom-element-reactions

attribute names. The attributeChangedCallback will be called for each change to any of the observed attributes. Changes to the attribute must be made through the methods `setAttribute` and `removeAttribute` in order to trigger the attribute changed event.

adoptedCallback

Whenever the component is moved to another document, the adopted callback is triggered. The parent of the component is replaced with a new element and ownership is passed on to the new document as described in the Mozilla MDN JavaScript documentation[5].

disconnectedCallback

The disconnectedCallback is executed whenever the component is removed from the document. Afterwards, the component will not be considered connected anymore, meaning that it does not have a DOM element as a parent.

2.2.2. React

Component life cycles for React are listed and described in the official documentation[6]. Changes in the UI or state trigger an automatic rerender of the affected components. A React component implements the following seven life cycle callbacks[7]. An overview of the transitions between the life cycle events is shown in Figure 2.6.

[5]https://developer.mozilla.org/en-US/docs/Web/API/Document/adoptNode
[6]https://facebook.github.io/react/docs/component-specs.html#lifecycle-methods
[7]https://facebook.github.io/react/docs/component-specs.html#lifecycle-methods

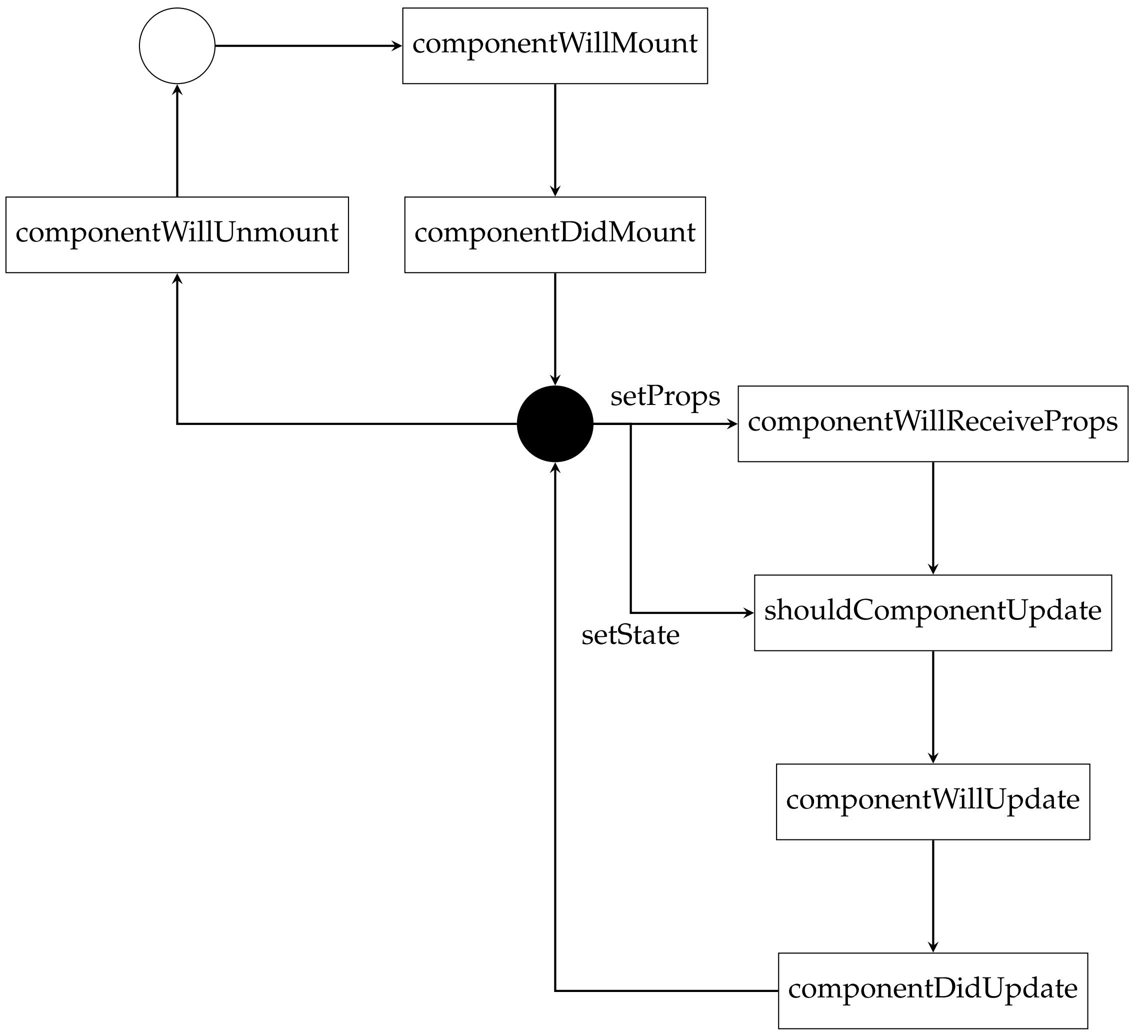


Figure 2.6.: Life cycle events defined in a React component

componentWillMount

This callback is executed after the component is created, but before the actual rendering occurs. The documentation states that it is possible to change the initial state for the rendering process within this method.

componentDidMount

Once the creation of the DOM tree has succeeded, this callback is executed. It is possible to access any child elements within the component since React renders each child before the corresponding parent.

componentWillReceiveProps

This callback is executed whenever a property of the component is about to change. It provides the possibility to change the change of the component before the UI is rerendered.

shouldComponentUpdate

React per default monitors the initial state of a component and rerenders parts of the UI on any change. Within this callback, it is possible to return a boolean value indicating whether the component should update or not.

componentWillUpdate

Upon updating the component due to a change in state, the componentWillUpdate method is called before the rerendering is executed.

componentDidUpdate

This method is called after the component rerendered part of its UI due to a change in state.

componentWillUnmount

This callback provides the last possibility to execute any action before the imminent detachment of the component from the DOM.

2.2.3. AngularJS

In the beginning, an architecture composed of modules, controllers, services and directories was used in AngularJS to encapsulate the functionality. With the release of Version 1.5, the component element was introduced. Components in AngularJS share similar traits with a directive, but provide better-suited functionality for a component based web-application[8]. Each component provides a series of life cycle hooks as defined in the official documentation[9]. The transitions are shown in Figure 2.7.

[8]https://docs.angularjs.org/guide/component
[9]https://docs.angularjs.org/guide/component#component-based-application-architecture

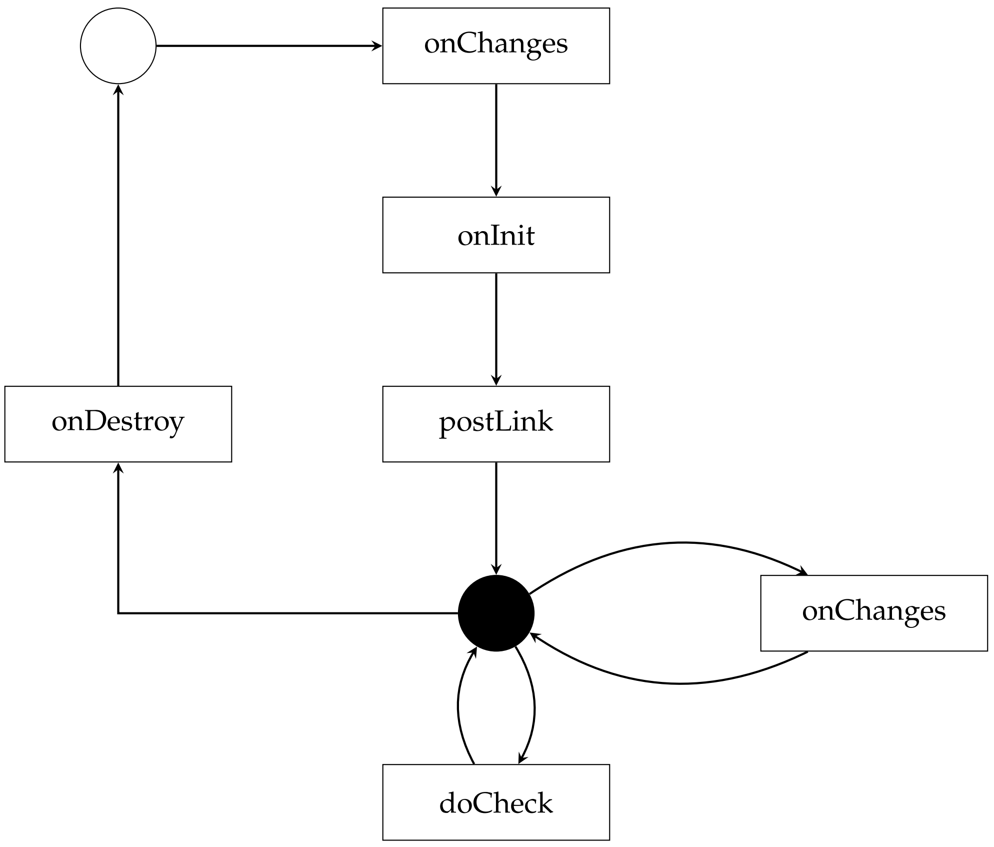


Figure 2.7.: Life cycle events defined in an AngularJS component

onInit

It is executed once per component upon creation of the component. It is meant to be used for initial setup and registration of further event handlers.

onChanges

This callback is executed whenever a one-way bound variable is updated. This typically occurs at two different stages of the life cycle. Initially, upon creation of the component when initial values for the variables are passed to the component, the method is executed even before the `doInit` callback. The second, and repeating, usage of the method is during the lifespan of the component. Whenever a binding is updated, the `onChanges` callback is called with a reference to the variable, and the earlier and newer value.

doCheck

`doCheck` is called whenever the internal change detection cycle of AngularJS is executed. It provides the possibility to implement custom change detection procedures.

onDestroy

This method is called when the component becomes detached from the DOM and the corresponding scope is destroyed.

postLink

This method is called after the initial setup has been completed and the DOM is constructed. Access to children elements is still restricted, due to asynchronous template loading of child elements in AngularJS.

2.2.4. Polymer

Polymer is a framework that tries to incorporate most of the Web Components standard while providing improvements wherever possible. It is also aimed at seamlessly integrating into the browser and other elements. The life cycle methods for a Polymer component[10] are illustrated in Figure 2.8.

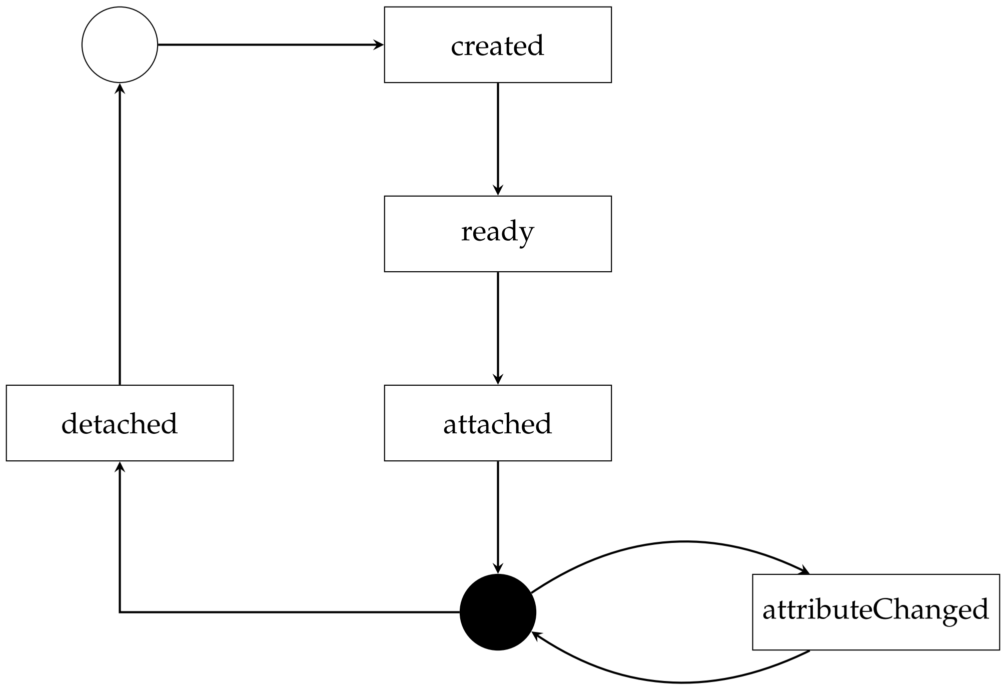


Figure 2.8.: Life cycle events defined in a Polymer component

created

This callback is invoked once when the initial set-up of the component is finished. The local DOM is not yet initialised at that point.

[10]https://www.polymer-project.org/1.0/docs/devguide/registering-elements#lifecycle-callbacks

ready

This callback is invoked when the set-up of the local child DOM is finished. The component itself is not yet attached to the page DOM.

attached

Whenever the component is attached to the page DOM, this callback is invoked.

detached

Counterpart callback which is invoked for whenever the component is detached from the DOM.

attributeChanged

This callback is invoked whenever the value of a non-declared property is changed. A declared property is defined as a property which is declared in the properties section of the component definition. Polymer automatically creates instances of those properties with default values and types[11].

2.2.5. Comparison

While each of the frameworks define different names for their life cycle callbacks, they all provide a similar functionality. Notable differences can be observed when looking at whether the callback is executed pre or post event. As an example, React provides callbacks for the pre and post connected event (`componentWillMount` and `componendDidMount`) while WebComponents only declares a post-event callback (`connectedCallback`). A comparison of each framework and callback is shown in Table 2.1. WebComponents, while being the framework of reference, offers the least amount of callbacks in comparison. It is also the only framework that explicitly defines a dedicated callback for adopting a component into a new document (`adoptedCallback`). The other frameworks offer that functionality via detaching the component and reinitialising it in the new document.

AngularJS, in particular, differs based on how the callbacks `doCheck` and `onChanges` are defined. No other framework provides the possibility to add custom change detection routines as it is possible in `doCheck`. Furthermore, `onChanges` is only invoked for one-way bound variables, rather than all component properties.

[11]https://www.polymer-project.org/1.0/docs/devguide/properties#attribute-deserialization

Table 2.1.: Side-by-side comparison of framework specific life cycle hooks

WebComponents	**React**	**AngularJS**	**Polymer**
constructor	-	onChanges	created
-	componentWillMount	onInit	ready
connectedCallback	componentDidMount	postLink	attached
-	componentWillReceiveProps	-	-
-	shouldComponentUpdate	-	-
-	componentWillUpdate	-	-
-	-	doCheck	-
attributeChangedCallback	componentDidUpdate	onChanges*	attributeChanged
-	componentWillUnmount	-	-
disconnectedCallback	-	onDestroy	detached
adoptedCallback	-	-	-

2.3. Refactoring Process

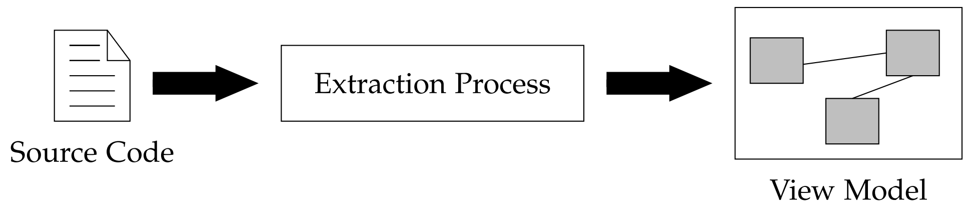


Figure 2.9.: Abstract view of extraction process

The extraction process can be described as a transformation function that translates JavaScript code into a view model as shown in Figure 2.9. As part of the function, various steps change, analyse or filter the in- and output information. Chikofsky and Cross [CC90] describe a basic architecture, provided by the Software Productivity Consortium, for a platform independent reverse engineering process. They suggest splitting the transformation function into three distinctive steps; Semantic analyser, information base and view composer. Each step performs independent transformations or extractions on the in- and output.

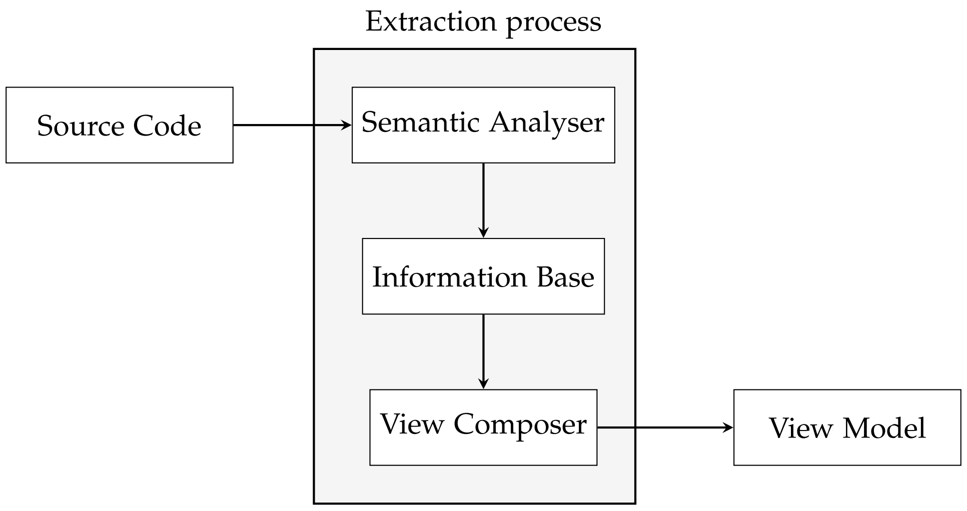


Figure 2.10.: Proposed process to extract the view model from JavaScript source code

The base architecture in Figure 2.10 describes a generic approach to handling data extraction and model generation. Applying the same approach to JavaScript and the specific framework required the selection of the right tools to perform the necessary actions in each step. The selected tools are shown in Figure 2.11.

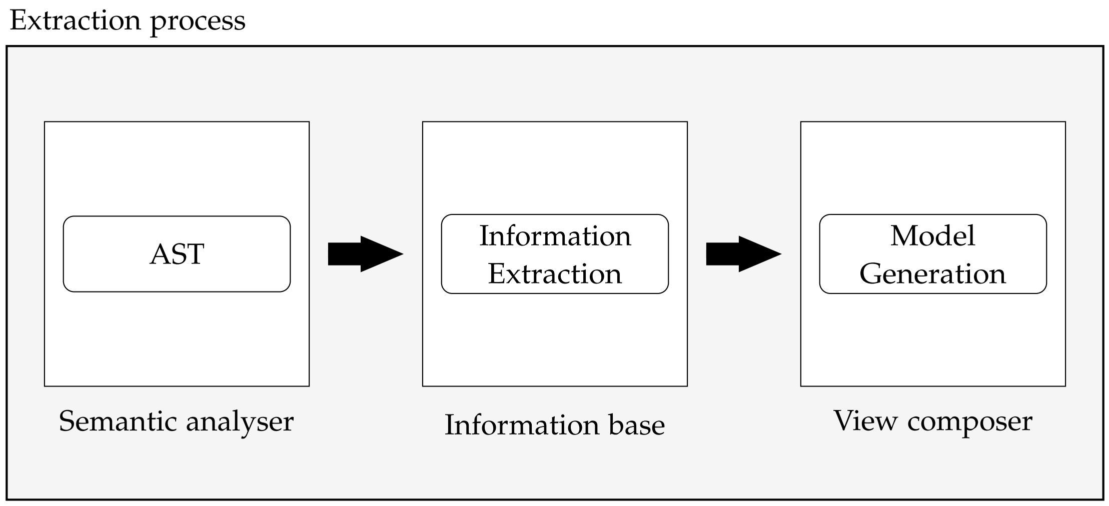


Figure 2.11.: Implementation specific tools for each extraction process step

2.3.1. Semantic Analyser

The Semantic analyser step focuses on transforming the input into well-structured information. Conventional source code usually contains unnecessary information which is not beneficial or relevant to reverse engineering. Fragments, like comments or white space, do not provide any insight and can be discarded. To effectively work with the given input, a syntactical analyser parses the source code into a syntax tree. This approach is commonly used in compilers for optimisation and analysis [Aho+86] or as the basis for code completion and error detection in integrated development environments [WM15].

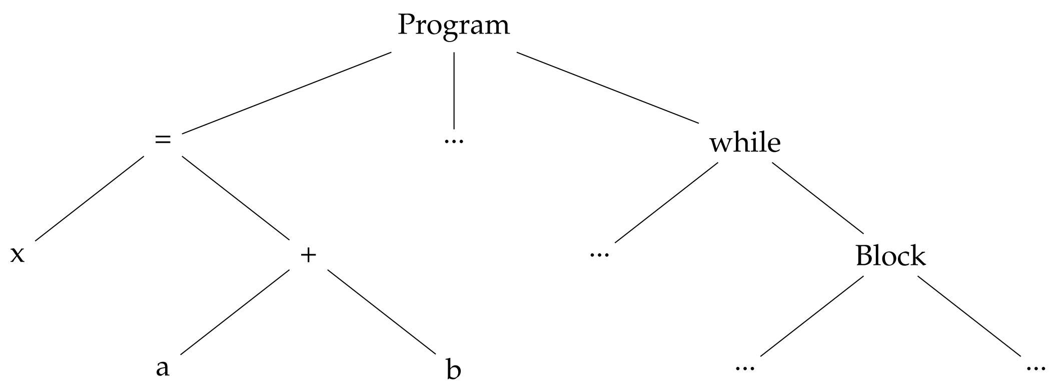


Figure 2.12.: Example of an abstract syntax tree

As Joel Jones [Jon03] states, if the transition from a given source code to another form of representation exceeds a certain amount of complexity, an intermediate step through an implementation language should be considered. Thus, the initial source code is first transformed to an AST and after that the view model can be generated. An abstract syntax tree resembles the example in Figure 2.12. The crucial structural elements are stored as part of the tree nodes, due to the tight relationship between the AST and the source language. An AST and corresponding tools are language specific [Koc16]. The specification for the JavaScript AST is based on the SpiderMonkey engine.

SpiderMonkey

SpiderMonkey was the first JavaScript engine originating from the works of Brendan Eich at Netscape and later on Dave Mandelin at Mozilla [Eic11]. Only after the engine was exposed through an API, the internal workings were then documented in the Mozilla Developer Network (MDN) [Her15]. The developer community took it upon themselves to continuously update the underlying definition of the abstract syntax tree following the latest developments and standards of JavaScript. The specification is hosted on GitHub [12] and an example tree is shown in Figure 2.13.

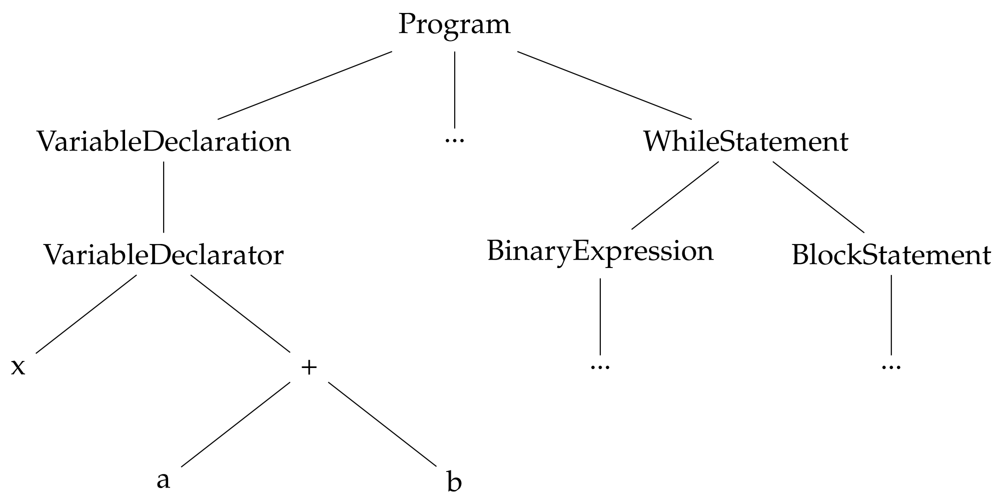


Figure 2.13.: Example of an ESTree abstract syntax tree

Syntactical Process

The Semantic Analyser step receives the unmodified source code as input (Figure 2.14). Before the syntax tree can be generated, a transformation of the code with a JavaScript compiler unifies the code which results in a unified syntax tree structure. The compiler does not alter the behaviour or remove crucial information, but rather adds additional and occasionally redundant information. As an example, JavaScript allows the assignment of unnamed or named functions to variables. The JavaScript compiler annotates all unnamed functions and transforms them into named functions based on the name of the assigned variable.

[12]https://github.com/estree/estree

Afterwards, the unified code is translated into a complete abstract syntax tree. The resulting tree contains all syntactical important elements. The post-processing provides the possibility to clear the tree of unnecessary information, while still remaining a valid AST. The output is referred to as *Relevant AST*.

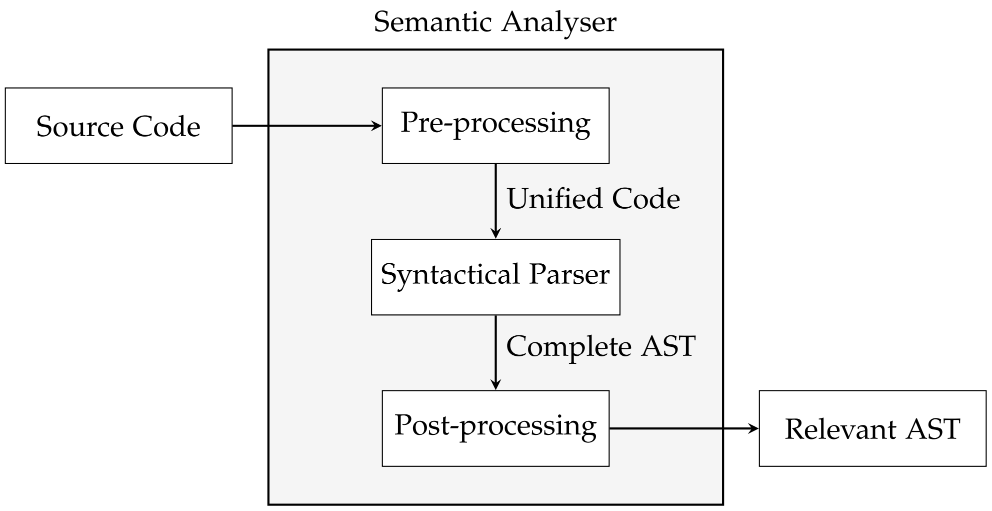


Figure 2.14.: Workflow of the semantic analyser with in- and outputs

2.3.2. Information Extraction

The information extraction takes the *Relevant AST* from the semantic analyser step as input and tries to gather and extract as much information as possible. The AST contains all relevant information as a tree structure and the retrieval of the correct leaves is achieved through a rule-based system. The gathered information is stored and passed on to the next step in the extraction process.

Rule-based System

The rule-based system is loosely based on the chain-of-responsibility pattern [Gam+94]. Each rule, or processing object, is specialised in independently retrieving a certain fragment of information from the tree. A dispatcher holds a list of all rules and runs them sequentially on the *Relevant AST* as shown in Figure 2.15. The results from each execution are stored in the information base under a unique identifier. After every rule has been executed, the resulting information base is passed to the next step.

Each rule in itself can work independently from any other rule. In general, it will query the abstract syntax tree for a certain information set. The resulting tree is a smaller tree which is contained in the original *Relevant AST* and is in itself a valid AST. The desired information can be retrieved by selecting a leaf, performing computations or flattening the tree.

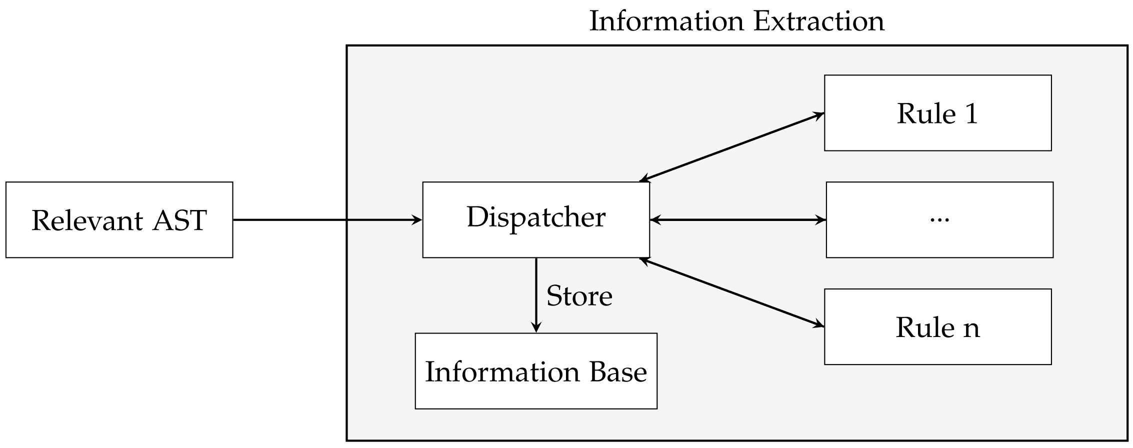


Figure 2.15.: Rule-based system for information extraction

2.3.3. Model Generation

The model generation receives the information base from the earlier step as its input. Based on the available information, the component model, as described in section 2.1, is created and populated. The process generates a view model and a dependency graph between the components (Figure 2.16).

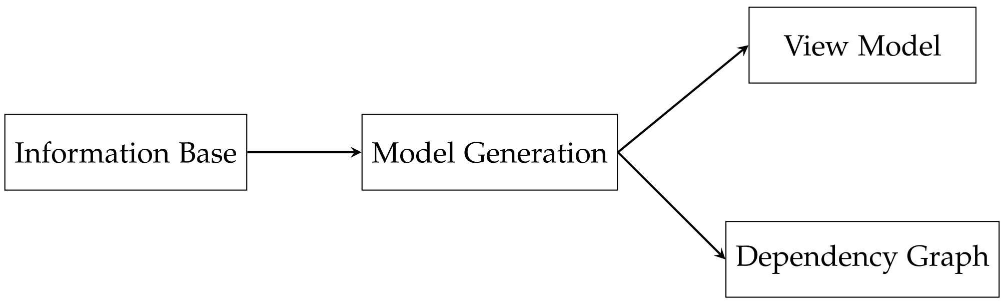


Figure 2.16.: Model generation with in- and outputs

3. Design and Implementation

3.1. Columbus

Columbus is a web-based JavaScript application which implements the view model extraction process described in section 2.3. It fetches the contents of a GitHub repository and shows the files and generated output from the applied extraction process. The information generated from each extraction step is displayed alongside the view model (Figure 3.1).

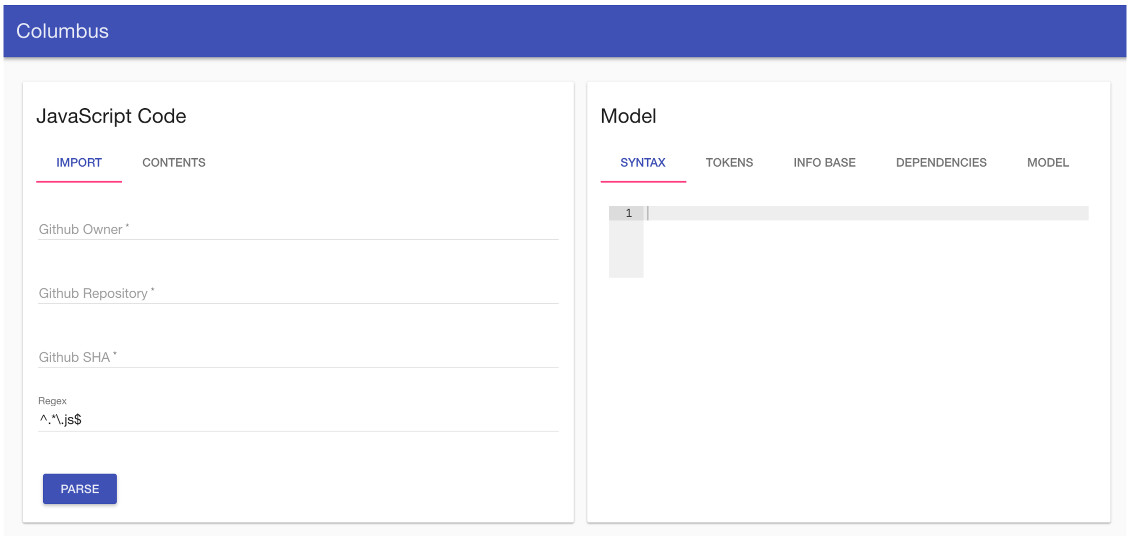


Figure 3.1.: User interface of Columbus

3.1.1. General Requirements and Limitations

The initial source code is fetched via the GitHub API from a public repository. The user interface provides inputs for a GitHub repository, commit hash and regular expression to restrict the fetched files. The repository is fetched recursively and only files which match the regular expression are passed onto the extraction process. Per default the tool fetches the current working state of the repository, but an optional *SHA* value of a specific commit can be supplied to select a specific version.

The JavaScript source code is limited to the format and extent defined in the ECMAScript 5 specification [ISO16262]. Each JavaScript file should contain exactly one component definition and the whole project must use the same framework. Dependencies to other components must be declared with the ECMAScript 6 `import`[1] statement. The definitions of the component must be done in a single file and directly without intermediate variables. Columbus cannot determine the value of complex variables, let alone function assignments. The extraction process is static, which means that any run time modifications cannot be identified or processed. Each framework and library imposes different limitations upon the allowed input. Specific limitations are listed in the subsequent chapters.

3.1.2. Architecture

Columbus is developed as an AngularJS 1.5 web-application. The architecture centres around the `AppCtrl` which directs the program flow during the whole extraction process. The general application architecture is shown in Figure 3.2. The diagram depicts the relevant structure and parent classes which are relevant for the framework specific extraction processes. The `SharedModelExtractorChain` and the `AbstractModelGenerator` must be extended for framework specific implementation logic. The green child classes of `Ast` provide access to common operations on framework specific syntax trees.

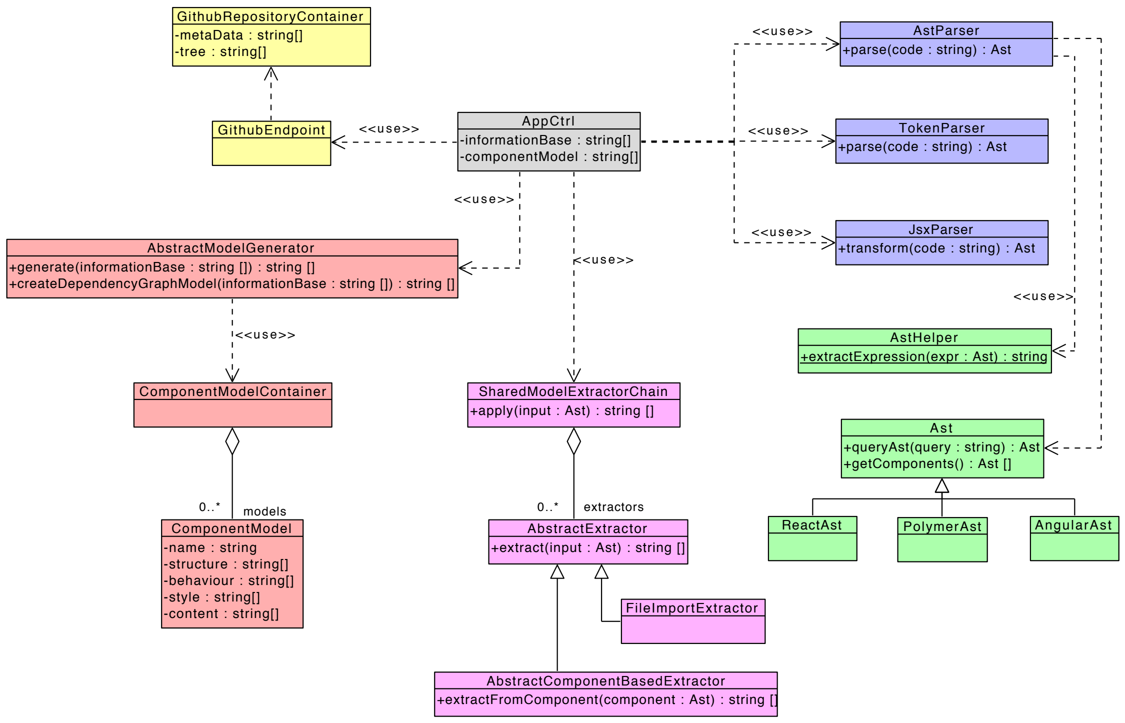

GithubRepositoryContainer
-metaData : string[]
-tree : string[]
GithubEndpoint
<<use>>
AppCtrl
-informationBase : string[]
-componentModel : string[]
<<use>>
AstParser
+parse(code : string) : Ast
<<use>>
TokenParser
+parse(code : string) : Ast
<<use>>
JsxParser
+transform(code : string) : Ast
<<use>>
AbstractModelGenerator
+generate(informationBase : string []) : string []
+createDependencyGraphModel(informationBase : string []) : string []
<<use>>
<<use>>
AstHelper
+extractExpression(expr : Ast) : string
<<use>>
ComponentModelContainer
SharedModelExtractorChain
+apply(input : Ast) : string []
Ast
+queryAst(query : string) : Ast
+getComponents() : Ast []
0..*
models
ComponentModel
-name : string
-structure : string[]
-behaviour : string[]
-style : string[]
-content : string[]
0..*
extractors
AbstractExtractor
+extract(input : Ast) : string []
ReactAst
PolymerAst
AngularAst
FileImportExtractor
AbstractComponentBasedExtractor
+extractFromComponent(component : Ast) : string []

Figure 3.2.: Architecture of Columbus

[1] http://www.ecma-international.org/ecma-262/6.0/#sec-imports

Code Retrieval (Yellow)

Columbus is capable of fetching the source code of GitHub repositories. The controller provides inputs for a repository owner, name and commit hash. The `GithubEndpoint` retrieves the contents of the repository recursively and stores each file and its contents as a `GithubRepositoryContainer`. Additionally, a regular expression can be supplied to the controller to restrict the fetched files. The flow of interactions is visualised in Figure 3.3. The initial fetch of the repository contents only includes a list of files without their content. It is necessary to iterate over all files and fetch the content individually.

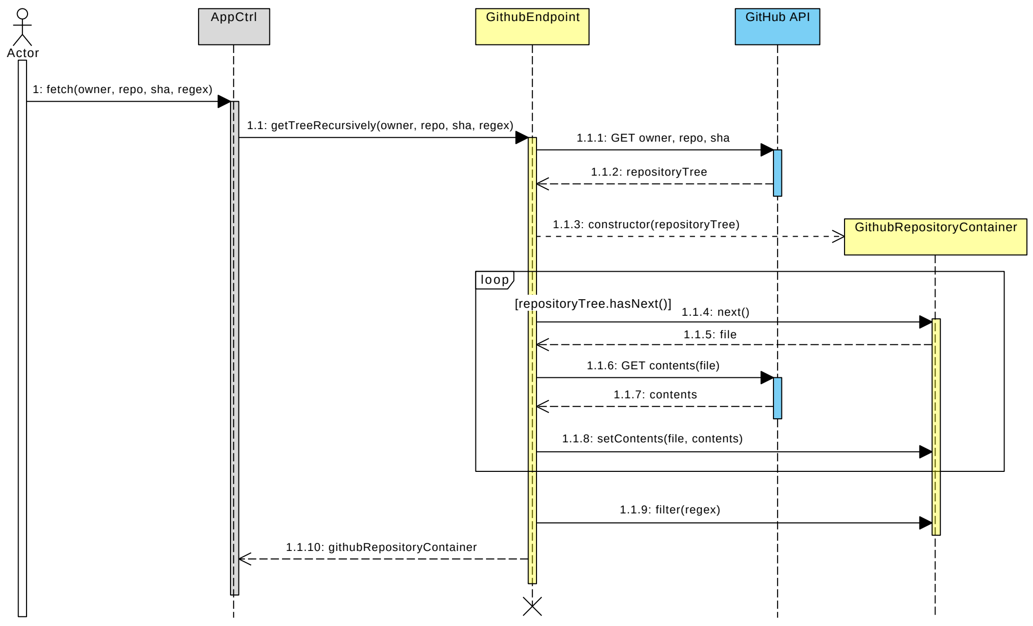

Actor
AppCtrl
GithubEndpoint
GitHub API
1: fetch(owner, repo, sha, regex)
1.1: getTreeRecursively(owner, repo, sha, regex)
1.1.1: GET owner, repo, sha
1.1.2: repositoryTree
1.1.3: constructor(repositoryTree)
GithubRepositoryContainer
loop
[repositoryTree.hasNext()]
1.1.4: next()
1.1.5: file
1.1.6: GET contents(file)
1.1.7: contents
1.1.8: setContents(file, contents)
1.1.9: filter(regex)
1.1.10: githubRepositoryContainer

Figure 3.3.: Sequence diagram of GitHub file retrieval

Semantic Analysers (Blue)

The blue classes act as façades for third party libraries for syntactical parsing.

The `AstParser` forwards the code to the library *Esprima 2.7.2*[2] which implements the ESTree standard[3]. It combines the syntactical parser and post-processing of the semantic analyser step, thus immediately generating the *Relevant AST*. The parser is also responsible for detecting the used framework in the input code. It tries to extract a

[2]http://esprima.org/
[3]https://github.com/estree/estree

component definition for every available supported framework until a valid match can be found. It returns a framework specific implementation of the `Ast` class.

The `TokenParser` performs a lexical analysis of the given source code with the same `Esprima` library. The generated tokens are displayed in the controller but are currently not used for the extraction process.

Pre-processing of the code is handled by the `JsxParser` or `BabelParser` which implement the Babel 6 compiler[4]. Both perform the same basic compilation of the source code. The `JsxParser` additionally converts the JSX tags inside Reacts components to valid JavaScript.

AST Models (Green)

The green classes are used as containers to be passed around inside the application. Each contains utility methods for different frameworks. `ReactAst`, `PolymerAst` and `AngularAst` contain shortcut methods to return the name of all contained components and component specific syntax trees. These methods are used primarily during the information extraction process. Additionally, the `Ast` class contains methods to query itself and return matching subtrees with the help of the ESQuery[5] library.

The `AstHelper` provides utility methods to work with the ESTree syntax tree. It provides methods to flatten a tree or recursively extract a value based on the trees node type. It can also be used to detect which framework is used in the source code.

Information Extractors (Purple)

The information extractor uses a handler class `SharedModelExtractorChain` which holds a reference to a list of registered extraction rules. The handler calls each rule sequentially and passes the `Ast` as an argument.

The rules can extend two different parent extractors: Child classes of `AbstractExtractor` receive the whole complete syntax tree from the semantic analyser as an input and can store their information directly under their unique key. Classes of type `AbstractComponentBasedExtractor` are called for each component and receive only the sub-tree of that component. Their returned values are stored under the unique key and grouped by component name. Figure 3.4 shows an example sequence diagram with React specific implementations of the operations performed during the extraction process.

[4]https://babeljs.io/

[5]https://github.com/estools/esquery

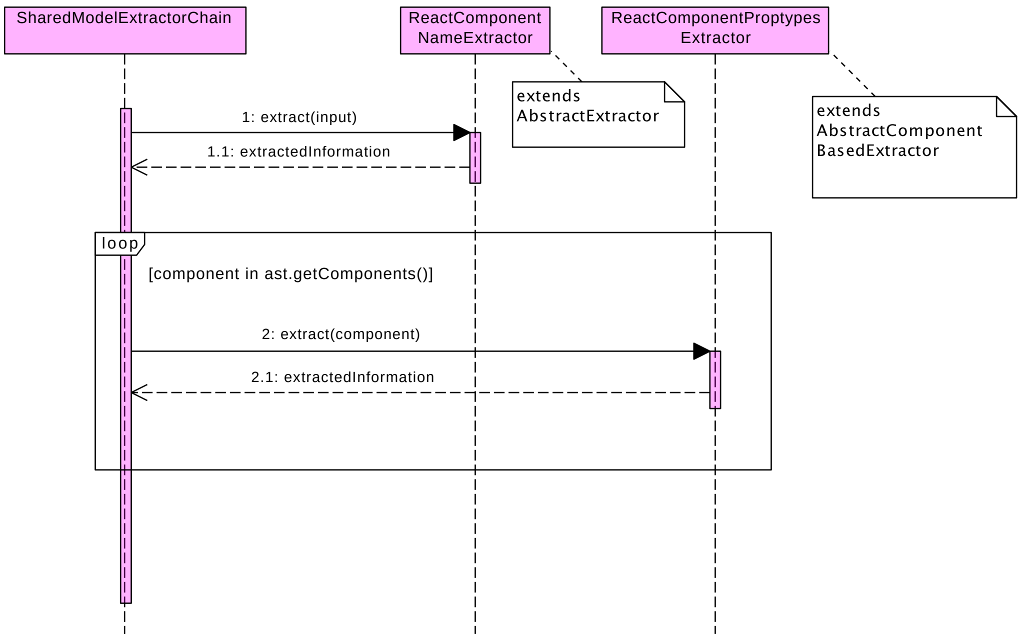

SharedModelExtractorChain
ReactComponent NameExtractor
ReactComponentProptypes Extractor
extends AbstractExtractor
extends AbstractComponent BasedExtractor
1: extract(input)
1.1: extractedInformation
loop
[component in ast.getComponents()]
2: extract(component)
2.1: extractedInformation

Figure 3.4.: Sequence diagram of the information extraction rule processing

Model Generators (Red)

The model generation selectively retrieves entries from the information base. Each iteration takes one or more information entries, compares and merges their content, and stores the information in the view model.

For each component identified during the extraction process, a corresponding blank component model is created at the start. The models are stored inside a `ComponentModelContainer` as shown in Figure 3.5. The component specific life cycle event rules are added to each blank component model at the beginning of the generation process. If a component defines a life cycle event and the entry already exists in the information base, the information is merged into the model and the new action is added to the existing event.

Once every blank model is set up, the model generator iterates through entries of the information base. The view model of each component is constructed piece by piece from the aggregated information.

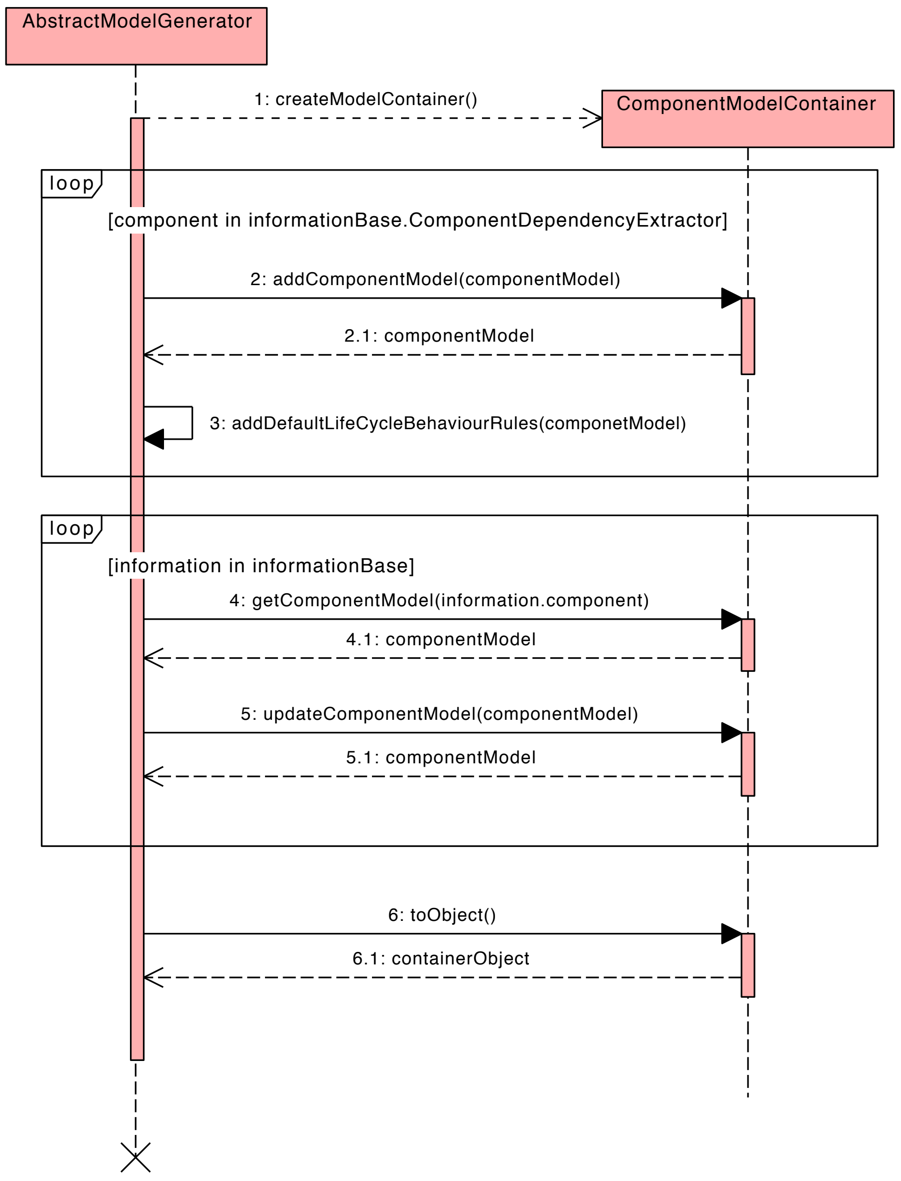

AbstractModelGenerator
ComponentModelContainer
1: createModelContainer()
loop
[component in informationBase.ComponentDependencyExtractor]
2: addComponentModel(componentModel)
2.1: componentModel
3: addDefaultLifeCycleBehaviourRules(componetModel)
loop
[information in informationBase]
4: getComponentModel(information.component)
4.1: componentModel
5: updateComponentModel(componentModel)
5.1: componentModel
6: toObject()
6.1: containerObject

Figure 3.5.: Sequence diagram of the model generation

Overview / Big Picture

Figure 3.6 shows an overview of interactions between the different parts of the extraction process. The `AppCtrl` is responsible for delegating the program flow through the whole process. The output of each iteration is stored in the controller and displayed in the user interface.

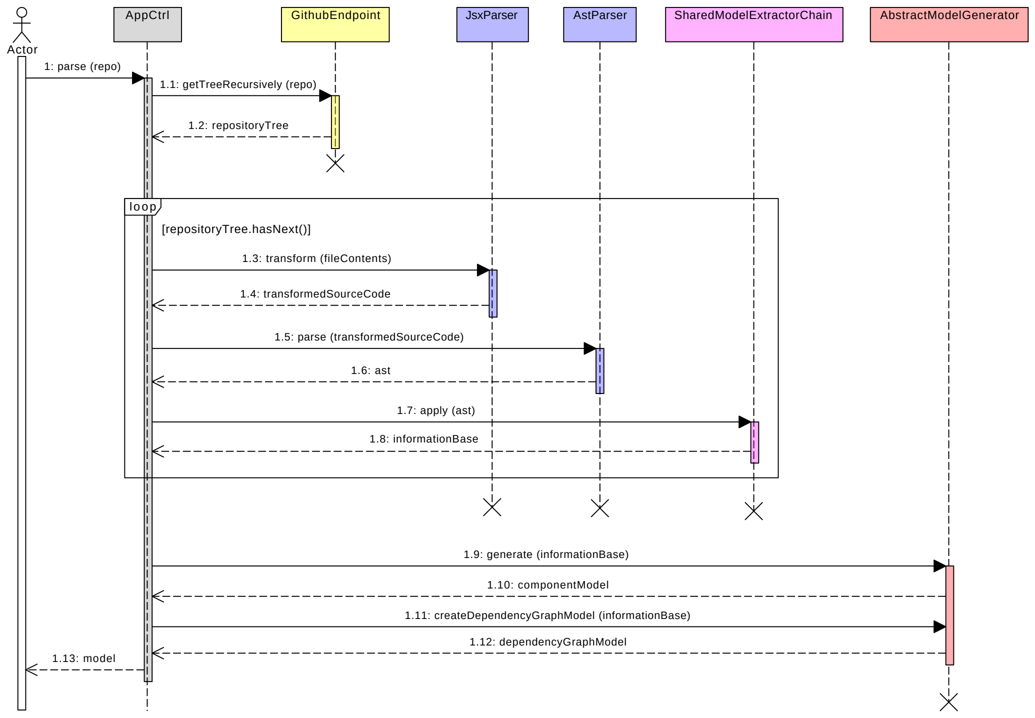
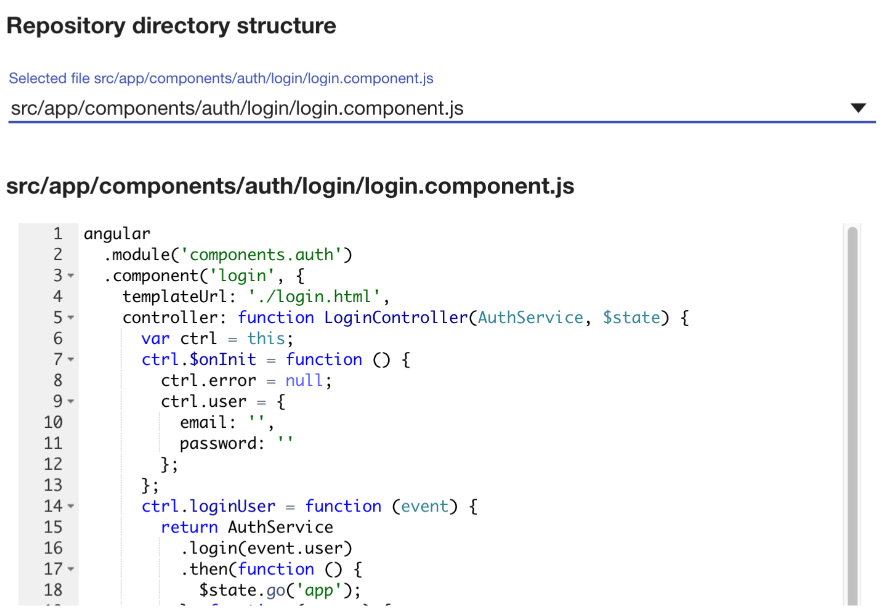

Actor
AppCtrl
GithubEndpoint
JsxParser
AstParser
SharedModelExtractorChain
AbstractModelGenerator
1: parse (repo)
1.1: getTreeRecursively (repo)
1.2: repositoryTree
loop
[repositoryTree.hasNext()]
1.3: transform (fileContents)
1.4: transformedSourceCode
1.5: parse (transformedSourceCode)
1.6: ast
1.7: apply (ast)
1.8: informationBase
1.9: generate (informationBase)
1.10: componentModel
1.11: createDependencyGraphModel (informationBase)
1.12: dependencyGraphModel
1.13: model

Figure 3.6.: Sequence diagram of the extraction process in Columbus

3.1.3. User Interface

Repository File Explorer

The files of the fetched and filtered GitHub repository are shown as a drop down menu inside Columbus. The currently selected file contents are displayed in an editor beneath the file selector as shown in Figure 3.7.



Repository directory structure

Selected file src/app/components/auth/login/login.component.js

src/app/components/auth/login/login.component.js

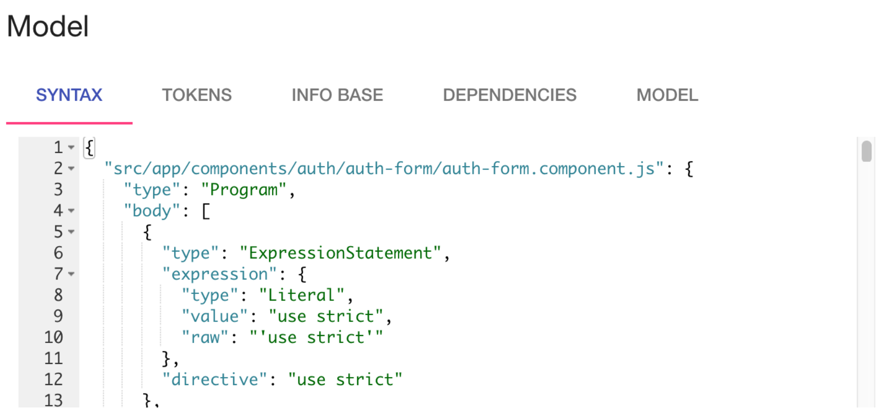
src/app/components/auth/login/login.component.js

```
angular
  .module('components.auth')
  .component('login', {
    templateUrl: './login.html',
    controller: function LoginController(AuthService, $state) {
      var ctrl = this;
      ctrl.$onInit = function () {
        ctrl.error = null;
        ctrl.user = {
          email: '',
          password: ''
        };
      };
      ctrl.loginUser = function (event) {
        return AuthService
          .login(event.user)
          .then(function () {
            $state.go('app');
```

Figure 3.7.: Columbus UI: Repository directory structure

Model Tabs

The output of each extraction step is displayed on the right side of Columbus (Figure 3.8). Each tab contains output which is passed along to the next step of the extraction process.



Model

```
{
  "src/app/components/auth/auth-form/auth-form.component.js": {
    "type": "Program",
    "body": [
      {
        "type": "ExpressionStatement",
        "expression": {
          "type": "Literal",
          "value": "use strict",
          "raw": "'use strict'"
        },
        "directive": "use strict"
```

Figure 3.8.: Columbus UI: Output tabs

The individual contents of each tabs are:

Syntax
Displays the *Relevant AST* of the semantic analyser step in JSON format.

Tokens
: Esprima provides the possibility to tokenise the source code and perform a lexical analysis. This information is currently not used in any further processing step, but is included in the output section for future use.

Info Base
: Displays the contents of the information base. Each information is grouped by the unique extraction rule identifier, the component and the file in which it originated from.

Dependencies
: Generated output of the dependencies between components which were declared with the import statements in each file. The output displays which components and their dependencies are declared in each file.

Model
: Generated output of the view model.

3.1.4. Produced Output

Columbus generates two output models from the extraction process.

Dependencies

The dependencies are extracted from the `import` statements. The JSON model displays the declared components and dependencies per parsed file. An example of the output is shown in Figure 3.9.

```
{
  "components/hello-world/hello-world.js": {
    "components": ["HelloWorld"],
    "dependencies": [
      "../second-component/second-component"
    ]
  },
  "components/second-component/second-component.js": {
    "components": ["SecondComponent"],
    "dependencies": [
      "list-item"
    ]
  }
}
```

Figure 3.9.: Example of extracted dependencies

View Model

The view model is constructed in JSON. Each view model has the component name as a parent and contains the four entities: structure, behaviour, content and style. The model generation step adds information to the model based on the contents of the information base. Multiple component definitions are stored as siblings and only references in the structure can link to other components.

```
{
  "component-name": {
    "HelloWorld": {
      "structure": {
        "parts": []
      },
      "behaviour": {
        "rules": []
      },
      "content": {},
      "style": {
        "properties": []
      }
    }
  }
}
```

Figure 3.10.: Example of a blank view model

3.2. React

React is a component-based declarative JavaScript framework for building web user interfaces. It is often used in conjunction with a preprocessor called JSX, which allows developers to write HTML like constructs inside JavaScript code.

3.2.1. Limitations

```
import React from 'react';
import SecondComponent from 'second-component';

var ComponentName = React.createClass({
  propTypes: {
    // ...
  },
  componentDidMount: function () { ... },
  ...
  render: function () {
    return (
      <div>
        <SecondComponent />
        ...
      </div>
    );
  }
});
```

Figure 3.11.: Example of a supported React component definition

Dependencies between components must be declared on top of each file using the ECMAScript 6 `import` syntax. The imported variable names can be used inside the render statement to refer to other components. All properties of a component must be declared inside the `propTypes` key of the definition. An optional static default value can be supplied inside the `getDefaultProps` method. The template must use JSX to define the structure, listeners and dependencies in the render function. Life cycle methods can be declared for each component and will be extracted by Columbus.

3.2.2. Semantic Analyser

Before a React component can be translated into an AST, the JSX tags must be converted into valid JavaScript code first. The Babel transformer[6] is used to convert the JSX syntax

[6]https://babeljs.io/

to valid ECMAScript 5. An example of a JSX snippet is shown in Figure 3.12. Each HTML tag is transformed into a method call to `React.createElement`. The contents of the tag is appended in the arguments at position three or higher.

```
return (
  <div id="content">
    <h1>Lorem ipsum</h1>
    <p>Neque porro quisquam</p>
  </div>
);
```

```
return React.createElement(
  "div",
  { id: "content" },
  React.createElement(
    "h1",
    null,
    "Lorem ipsum"
  ),
  React.createElement(
    "p",
    null,
    "Neque porro quisquam"
  )
);
```

Figure 3.12.: Example of a JSX to JavaScript transformation

The semantic analyser for React requires a preprocessing step in which the JSX code is transformed and valid JavaScript code is generated (Figure 3.13).

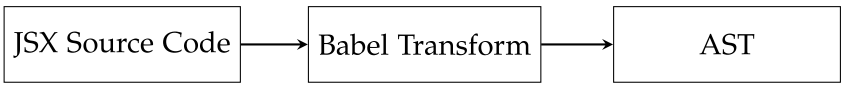


Figure 3.13.: Syntactical analyser steps for React

Post-processor

React requires the use of a post-processor to alter the generated AST.

During the transformation with the JavaScript compiler, the original import statements at the top of the file are altered. The original import expressions are converted into ECMAScript 5 compliant statements, which in turn alters the used variable names in the remainder of the file. An example for the HelloReact project is shown in Figure 3.14. The same replacement occurs for each usage of the imported variable in the component. This meant that each time the SecondComponent is referenced, the replaced variable is used instead. This leads to the deviating class name in the reference entity unless the variable names are transformed back into their original name.

```
// Original source code (ECMAScript 6)
import SecondComponent from 'second-component';

// Transformed code (ECMAScript 5)
var _secondComponent = require('second-component');
var _secondComponent2 = _interopRequireDefault(_secondComponent);
function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj :
    { default: obj }; }
```

Figure 3.14.: Transformation of import statements

The post-processor selects all variable assignments of `interopRequireDefault` with the query shown in Figure 3.15. Each returned entry contains the new variable name, for example `_secondComponent2` and the original name `_secondComponent` with a lower-case first character. For each entry, all occurrences of the new variable are replaced with the original name, which in this case would be `SecondComponent`.

```
[body]>[declarations]>[init.callee.name="_interopRequireDefault"]
```

Figure 3.15.: AST query for selecting variable assignments of `interopRequireDefault`

3.2.3. Information Extraction

All rules, except the first rule to extract all component names, are executed on a per component basis. This means that the initial input syntax tree is a sub-tree of the original AST and contains only elements inside the component declaration.

Extracting Component Declarations

This rule provides a list of all component declarations occurring inside the source code. It searches for variable declarations which are initialised by calling `React.createClass`. The rule returns the name of each variable and stores it inside the information base.

```
[body] > [type="VariableDeclaration"] > [init.type="CallExpression"]
```

Figure 3.16.: AST query for selecting variable declarations

The query shown in Figure 3.16 returns a set of syntax sub-trees, each having a variable declaration as the root node. The returned trees are already filtered to include only call expressions. Out of the returned sub-trees, only `React.createClass` method calls should stay in the final component set. For each entry, the name of the variable in the assignment is stored in the information base. The whole process is illustrated in Figure 3.17.

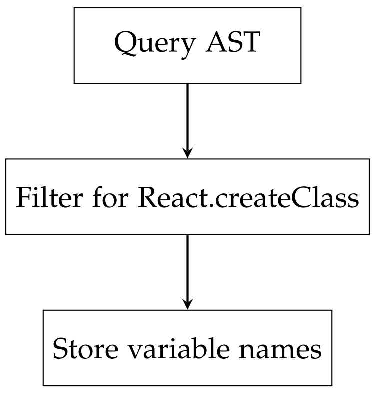


Figure 3.17.: Component extraction rule execution

Figure 3.18 illustrates the filtering steps based on JavaScript code examples.

```
// Filtered out by query
var notAComponent2 = 5;

// Filtered out by post query filter
var notAComponent1 = myFunction();

// Returns "myComponent"
var myComponent = React.createClass({
  // ...
});
```

Figure 3.18.: Component extraction illustration based on code snippets

Extracting component proptype declarations

Each React component can contain a property called `propTypes`. It allows specifying variables names and types for type validation. For example, it is possible to declare a property called `name` and declare it as a required string property. Definitions inside `propTypes` provide the most reliable source of names and validated types of component variables.

It is possible to extract the relevant information solely by executing the query shown in Figure 3.19. The resulting set contains an entry for each declaration inside `propTypes` and the relevant information can directly be added to the information base.

```
[key.name="propTypes"] > [properties] > [type]
```

Figure 3.19.: AST query for selecting propTypes

It is only possible to extract simple member expressions like `React.PropTypes.string` or `React.PropTypes.func.isRequired`. It is not possible to correctly interpret call expressions with parameters or definitions based on the return value of a function. See Figure 3.20 for an example of declarations.

```
propTypes: {
  // Can be extracted with name and type
  optionalBool: React.PropTypes.bool,
  requiredAny: React.PropTypes.any.isRequired,

  // Extracting the type is limited to "React.PropTypes.arrayOf()"
  optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

  // Extraction is limited to the name and no type
  customProp: function(props, propName, componentName) { ... }
}
```

Figure 3.20.: Example of possible ways to define propTypes in React

Extracting component default property values

React provides the possibility to define default values inside a `getDefaultProps` function. The function returns an object with the name of the property as key and the default value as the object value.

The contents of that array can be extracted with the query shown in Figure 3.21. This query, similar to the earlier propTypes extractor rule, already returns the specific set of properties. The relevant information can be immediately extracted and stored in the information base.

```
[type="FunctionExpression"][id.name="getDefaultProps"] [type="
    ReturnStatement"] [properties] [type="Property"]
```

Figure 3.21.: AST query for extracting default values inside the getDefaultProps function

This rule is only capable of extracting assignments of constant value, variables, or functions without parameters. Additionally, any conditions altering the contents of the return statement cannot be detected correctly. The query to the syntax tree combine all occurring return statements, which can lead to inconsistencies. For example, it would be possible to declare a variable `foo` as value `true` in one return statement, while returning `false` in another. The rule would insert the variable `foo` twice to the information base, once with value `true` and once with `false`.

Extracting functions of a React component

This rule returns a list of all functions with corresponding parameter names declared inside a component. The query in Figure 3.22 selects all function declarations which are declared in the component. Each returned sub-tree contains the name of the function, as well as a list of the parameter names.

```
[arguments] > [properties] > [value.type="FunctionExpression"]
```

Figure 3.22.: AST query for extracting function declarations

Extracting inter-component dependencies inside the render function

It is possible to reference components inside JSX tags by including a custom HTML tag which starts with an upper-case letter. During the transformation process the definition of the component will be converted into the first parameters of a `React.createElement` function call. While common HTML tags are added as a literal, the component is an identifier, thus allowing one to distinguish between them.

```
render: function() {
  return (
      <div>
        <AnotherComponent />
      </div>
    );
}
```

```
render: function render() {
  return React.createElement(
    'div', // Literal
    null,
    React.createElement(
      // Identifier
      AnotherComponent,
      null
    )
  );
}
```

Figure 3.23.: JSX transformation of an inter-component dependency

As per the definition of `React.createElement`[7], a dependency to another component always has to be the first parameter. This behaviour can be incorporated into the query as shown in Figure 3.24. Using the selector `:first-child` guarantees that only

[7]https://facebook.github.io/react/docs/react-api.html#createelement

occurrences with an identifier as the first parameter are returned. For each returned set, only the name of the dependency is stored in the information base. The view model does not yet support defining which variables are assigned to another component. Any additional parameters for component declarations in JSX are therefore ignored during the extraction.

```
[value.id.name="render"] [type="ReturnStatement"] [arguments] [type="
    Identifier"]:first-child
```

Figure 3.24.: AST query for extracting component references inside the render function

A benefit of this rule is to provide an overview of the dependencies between components. With the information retrieved, it is possible to create a basic dependency graph. The graph contains all declared components and links towards all child components. An example of how such a graph could look like is shown in Figure 3.25.

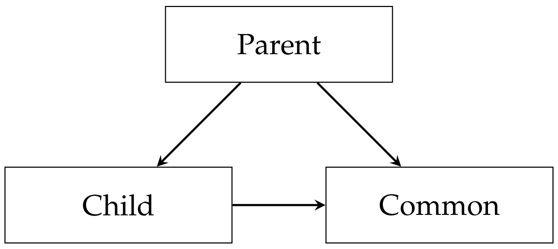


Figure 3.25.: Example of a dependency graph between components

Extracting propTypes usage inside render function

In React it is possible to use a variable in a function, without explicitly declaring the variable and corresponding type. It is therefore not enough to only extract the declared propTypes of a component, but also check for any usage of the member expression `this.props` inside functions. The query in Figure 3.26 filters all member expressions inside the render function. The returned set contains the variable names which are stored in the information base. It is possible for this rule to extract duplicates depending on the usage of the variables. The model generation is responsible for combining the information in the view model.

```
[key.name="render"] [type="MemberExpression"][object.property.name="props"
    ]
```

Figure 3.26.: AST query for extracting propTypes usage inside the render function

Extracting possible return types for functions

Determining the return type of a function provides insight into the usage and behaviour of a function. Since JavaScript does not use a declarative way to declare the returned type, it is necessary to analyse the variable types used in the return statements. Additionally, JavaScript does not impose a limit to one type, but rather allows for multiple different ones. To correctly find the set of return types per function, each return statement must be analysed. As a first step, a list of all function sub-trees is retrieved through the query shown in Figure 3.27.

```
[type="FunctionExpression"]
```

Figure 3.27.: AST query for extracting all function declarations

For each function, a list of all return statements must be selected with Figure 3.28. This query is executed for each of the earlier function subtrees.

```
[type="ReturnStatement"]
```

Figure 3.28.: AST query for extracting render statements inside a specific function subtree

Based on the kind of expression inside the return statements, different information can or cannot be extracted. While it is possible to extract the type and value of a hardcoded string, determining the correct type of a variable or even function is not possible. For variables and function expressions, only the name is stored inside the information base. The possible extraction types are shown in Figure 3.29.

```
fnc1: function () {
  return 'Dummy'; // Extracting type 'Literal' and value 'Dummy'
},
fnc2: function () {
  var foo = 5;
  return foo; // Extracting type 'Identifier' and variable name 'foo'
},
fnc3: function () {
  return fnc4(); // Extracting type 'CallExpression'
},
```

Figure 3.29.: Example of supported return types for functions

Extracting HTML structure inside render function

As part of the component model, the structure contains a hierarchy of parts. The HTML code provides insight into the structure of a component. Each tag can be seen as a web component which contains additional tags, information or dependencies to other JavaScript components. The extraction process targets the converted JSX source code, which uses `React.createElement` method calls. The JSX transformation process places an element which can contain child elements at the beginning of the parameter list (Figure 3.30). Elements like `div`, `p` or `i` are passed as first parameter, with corresponding attributes as second attribute. All elements placed third and onwards are child elements of the first placed element.

```
render: function() {
    return (
        <div id="header">
          <h1>Text</h1>
          <p>
            Hello
            <i>World</i>
          </p>
        </div>
      );
  }
```

```
render: function render() {
    return React.createElement(
      "div",
      { id: "header" },
      React.createElement(
        "h1",
        null,
        "Text"
      ),
      React.createElement(
        "p",
        null,
        "Hello ",
        React.createElement(
          "i",
          null,
          "World"
        )
      )
    );
  }
```

Figure 3.30.: JSX transformation of JSX code

This behaviour can be used to define a recursive extraction process which is visualised in Figure 3.31. The query to the syntax tree returns a set of calls to `React.createElement` . For each call, the argument at the first position corresponds to the current parent element and all subsequent elements will be treated as child elements. Basic elements like literals, member or call expressions can immediately be resolved and appended. Calls to `React.createElement` must be processed recursively, since they might contain additional elements. This process repeats until no further element remains.

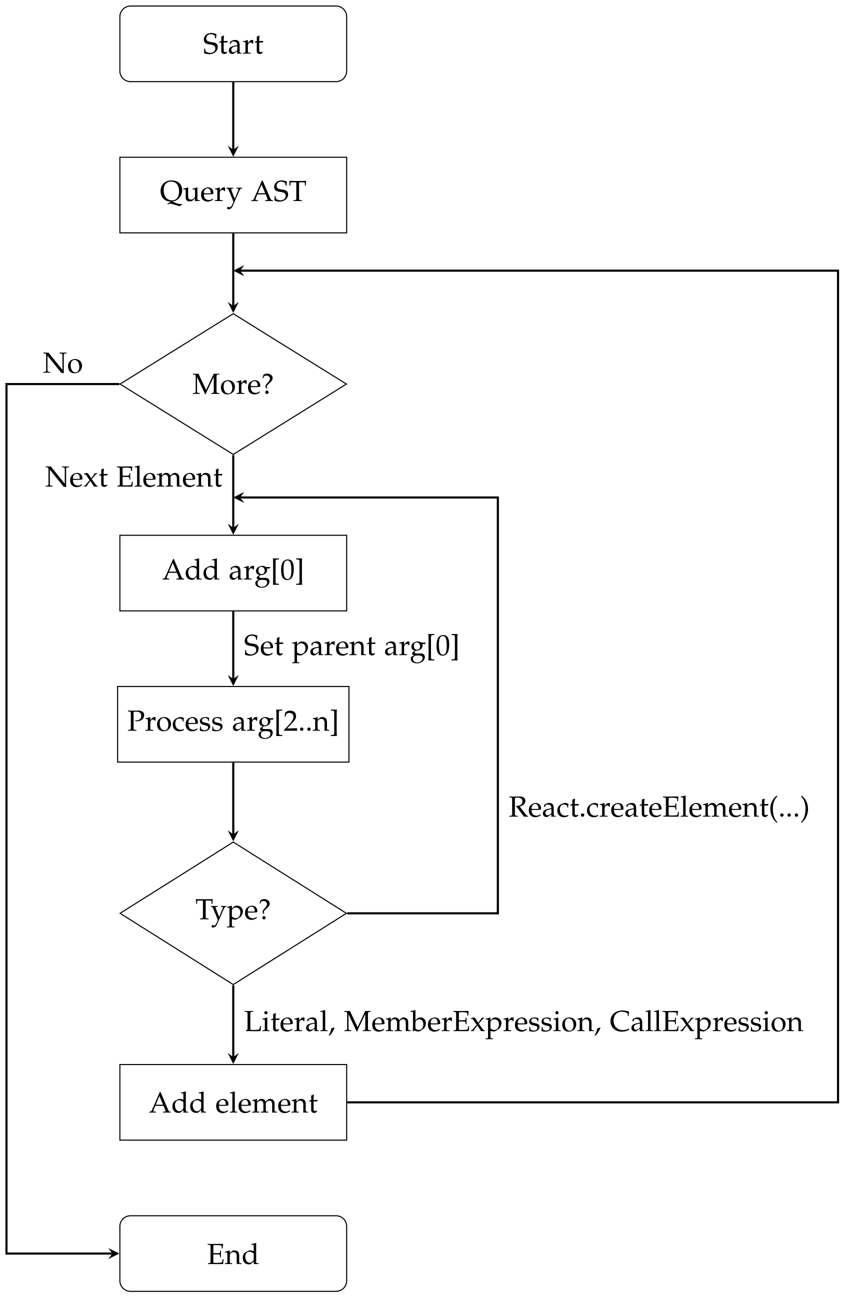


Figure 3.31.: Extraction process of the template inside the React render function

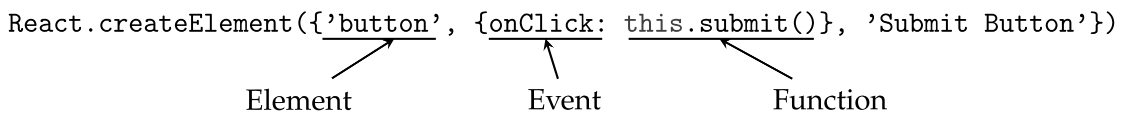
Extracting declaration of events and associated behaviour

This rule extracts runtime behavioural function calls which are triggered by events starting with "on...". Possible variations are among others `onClick`, `onKeyPress` or `onBlur`. A full comprehensive list is provided in the official React documentation for the event system [8].

```
[type="FunctionExpression"][id.name="render"] [type="CallExpression"]'
```

Figure 3.32.: AST query for extracting call expressions inside the render function

The initial query in Figure 3.32 selects all call expressions within within the render function. In order to only select elements which have event listeners attached to them, each `React.createElement` call is filtered based on the second argument, which contains all properties of the current element. The object at the second position must contain at least one entry with a name starting with "on...". The matching entries are stored inside the information base. The first argument in `React.createElement` is the name of the object on which the event is attached to. An example is shown in Figure 3.33. The button has an on click listener attached to itself, which calls the method `this.submit()`.



```
React.createElement({'button', {onClick: this.submit()}, 'Submit Button'})
```

Figure 3.33.: Breakdown of parameter associations for `React.createElement` function call

3.2.4. Model Generation

Based on the information aggregated in the information extraction process the model can be constructed. Each process takes one or more information entries, compares and merges their content, and stores the information as part of an entity in the view model. The individual information fragments are stored in the information base under the unique key of the extractor rule as shown in Figure 3.34.

[8] https://facebook.github.io/react/docs/events.html

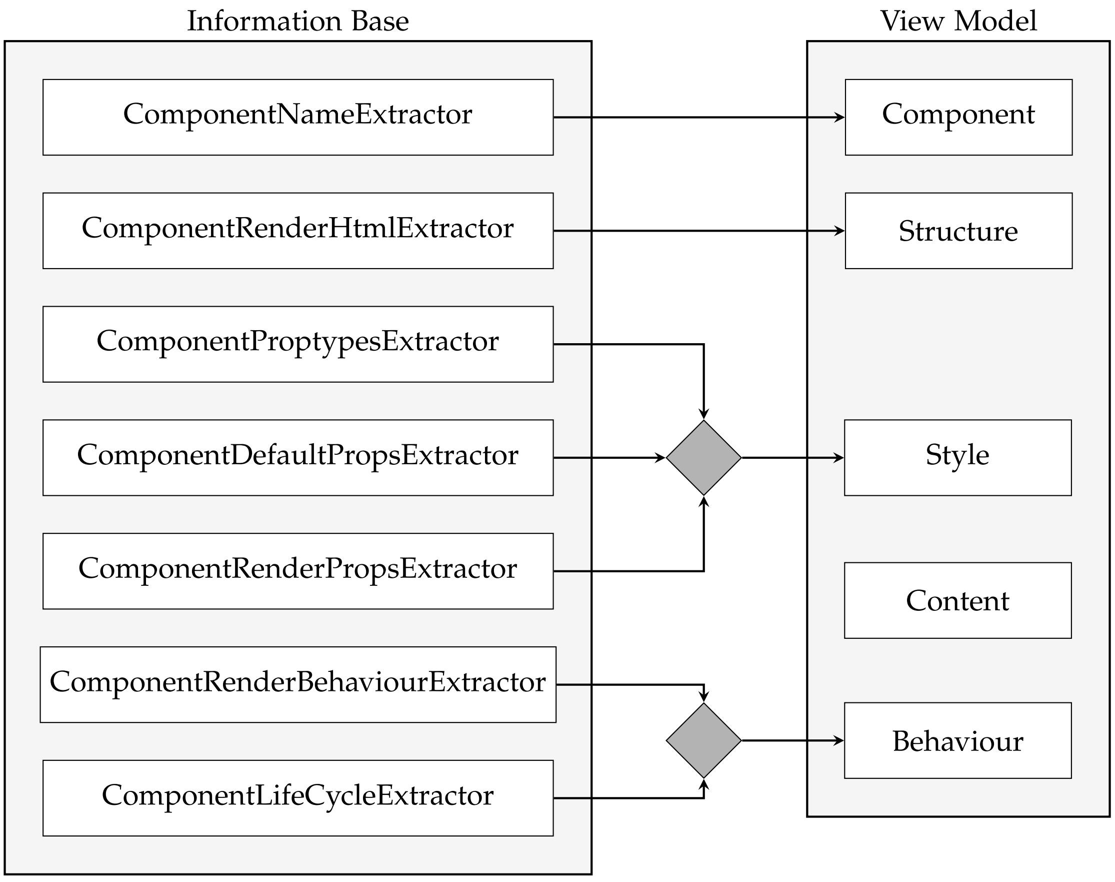

Information Base
View Model
ComponentNameExtractor
ComponentRenderHtmlExtractor
ComponentProptypesExtractor
ComponentDefaultPropsExtractor
ComponentRenderPropsExtractor
ComponentRenderBehaviourExtractor
ComponentLifeCycleExtractor
Component
Structure
Style
Content
Behaviour

Figure 3.34.: Model generation for React components

Populating the structure

The extraction step converts the JSX code into a JSON representation and stores it inside the information base. In order to include the structure in the model, the extracted information must be converted into entities. The parent structure entity contains a single part entity, which in turn can contain multiple child parts. The filling of the model can be achieved through recursively stepping through the entries in the information base.

HTML elements are added as the concrete entity Part and can contain additional child parts. References to other components are stored as a Reference entity. An example of the two step transformation process from JSX to view model is shown in Figure 3.35.

JSX Code

```
<div>
  <p>Text</p>
  <HelloWorld />
</div>
```

Information Base

```
{
  "type": "
      HtmlExpression",
  "value": "div",
  "children": [
    {
      "type": "
          HtmlExpression
          ",
      "value": "p",
      "children": [
        ...
}
```

Component Model

```
"structure": {
  "parts": {
    "_entity": "part",
    "className": "div",
    "parts": [
      {
        "_entity": "part"
            ,
        "className": "p",
        "parts": [
          ...
        ]
      },
      {
        "_entity": "
            reference",
        "className": "
            HelloWorld"
      }
    ]
  }
}
```

Figure 3.35.: Transformation steps from JSX code to a view model

Aside from the parts, the generation process additionally pre-populates behaviour rules and style properties if a corresponding entry in the information base is present. Each CallExpression which is evaluated when rendering the component corresponds to a behaviour call that is triggered on the life cycle event `componentWillMount`.

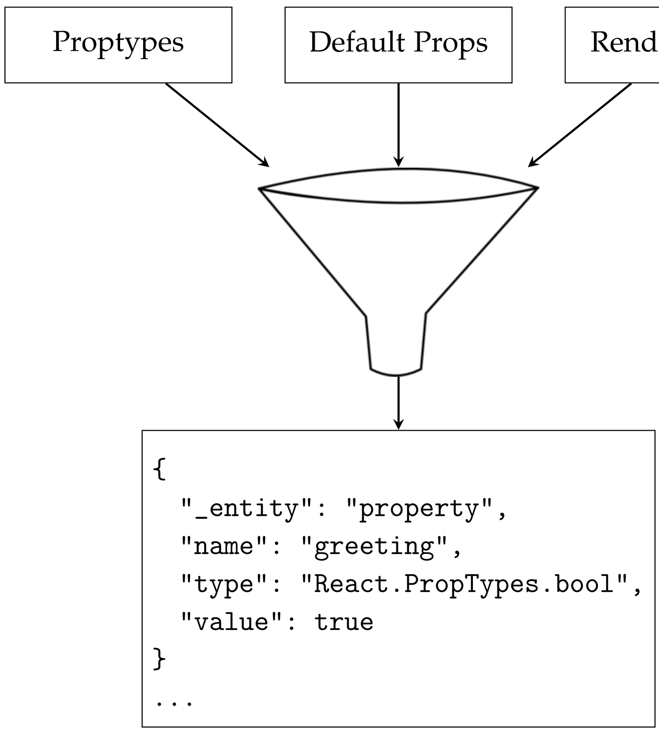
Regular text, like the content of an HTML element or labels of buttons, and input elements are stored as property and referenced via a unique id, which is added to each part of the structure entity.

Adding additional behaviour calls

Besides methods being executed upon rendering of the component, certain methods are executed on runtime events like `onClick` or `onSubmit`. These methods are added to the behaviour along with the triggering event and target part.

Merging and adding variable declarations

JavaScript allows the usage of variables without a previous declaration. This proves to be a challenge, since possible variable usages can be spread among several methods inside the component. The information extraction uses three rules to detect a majority of declarations. Individual entries can either be unique, duplicates or prove to be conflicting with each other. The output of the process contains the merged information from all input sources (Figure 3.36).



```
{
  "_entity": "property",
  "name": "greeting",
  "type": "React.PropTypes.bool",
  "value": true
}
...
```

Figure 3.36.: Combine and merge information base entries to create unambiguous variable definitions

3.2.5. HelloReact Development Project

The HelloReact[9] example project was used during the creation of Columbus and implements examples of the supported functionality. The repository contains two React components and an HTML file.

- index.html
- components/hello-react/hello-react.js
- components/second-component/second-component.js

[9]https://github.com/tielitz/columbus-react-example

Dependencies

The HelloReact component requires the SecondComponent and both have a dependency to the React library (Figure 3.37). Both components are defined in different directories, which means that the import statement contains path expressions to navigate the folder structure. Columbus cannot correctly interpret the directory structure and ignores the path expressions altogether.

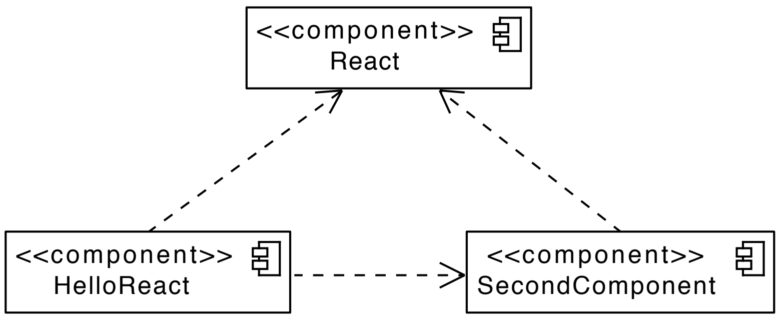


Figure 3.37.: Extracted dependencies of the HelloReact project

View Model of the HelloReact Component

The generated view model of the HelloReact component is shown in Figure 3.38. The JSON model has been converted into an object diagram.

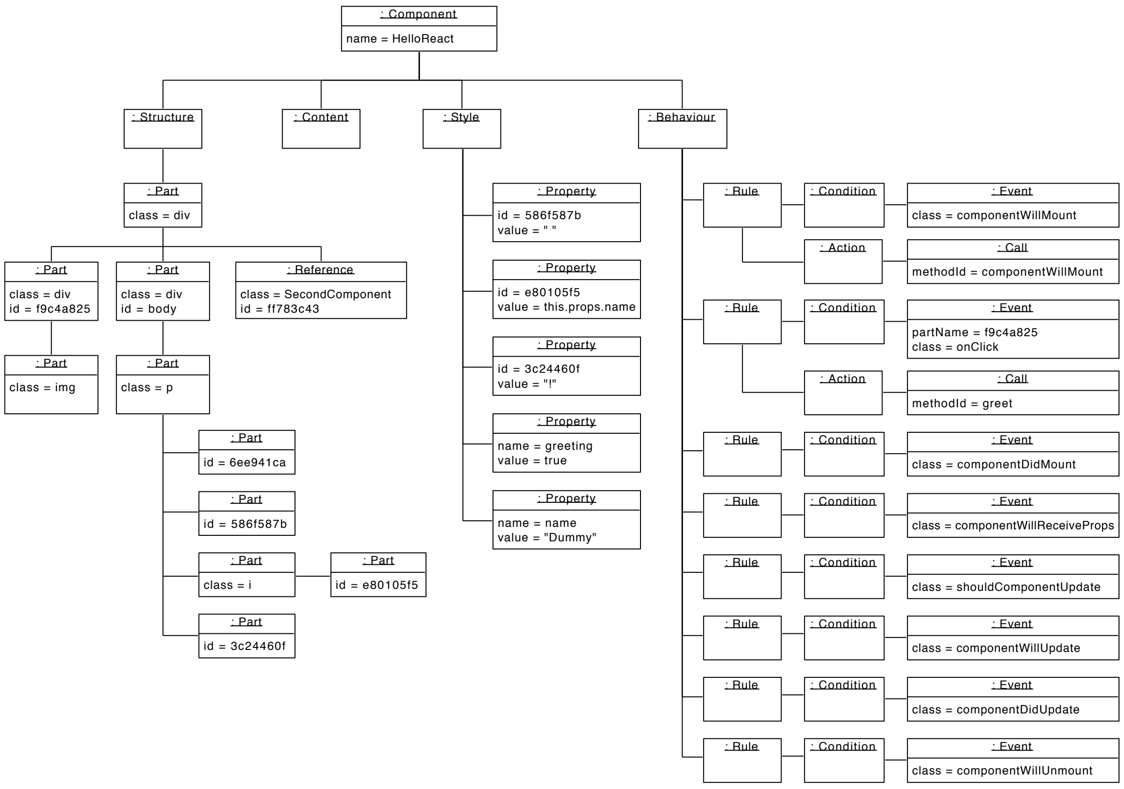


Figure 3.38.: Extracted view model of the HelloReact component

Due to the availability of a JSX compliant JavaScript compiler, the template can be converted into pure JavaScript. This transformation allows Columbus to extract the HTML structure and library specific expressions.

A limitation can be observed in the render function. Columbus is capable of extracting basic JavaScript expressions, but fails at processing statements like *if-else*, *loops*, or any runtime modifications of the HTML structure, like `React.cloneElement`. The part with the id *6ee941ca* contained an in-line *if-else* output of a variable and function call. This demonstrates another limitation of the view model since it does not support runtime alterations of the structure. The properties as such can only hold either a hardcoded string or reference to a variable.

The properties contained all text fragments used in the template, as well as all declared component properties. Columbus managed to extract the name, type and value of the properties, regardless if they were used inside the render statement or not. During the extraction process, all occurring text fragments were treated as properties, since it is not possible to determine correct internationalisation. Therefore, the content of the model is left empty.

The behaviour contained an entry of every life cycle event which can occur in the React component, as well as all user defined events. Life cycle events without an implementation only include an event entry without a corresponding action call. In the case of `componentWillMount`, a callback function was provided in the application and therefore the action entity was present. The user defined `onClick` event was included in the model as well.

View Model of the SecondComponent Component

The generated view model of the SecondComponent component is shown in Figure 3.39. The JSON model has been converted into an object diagram.

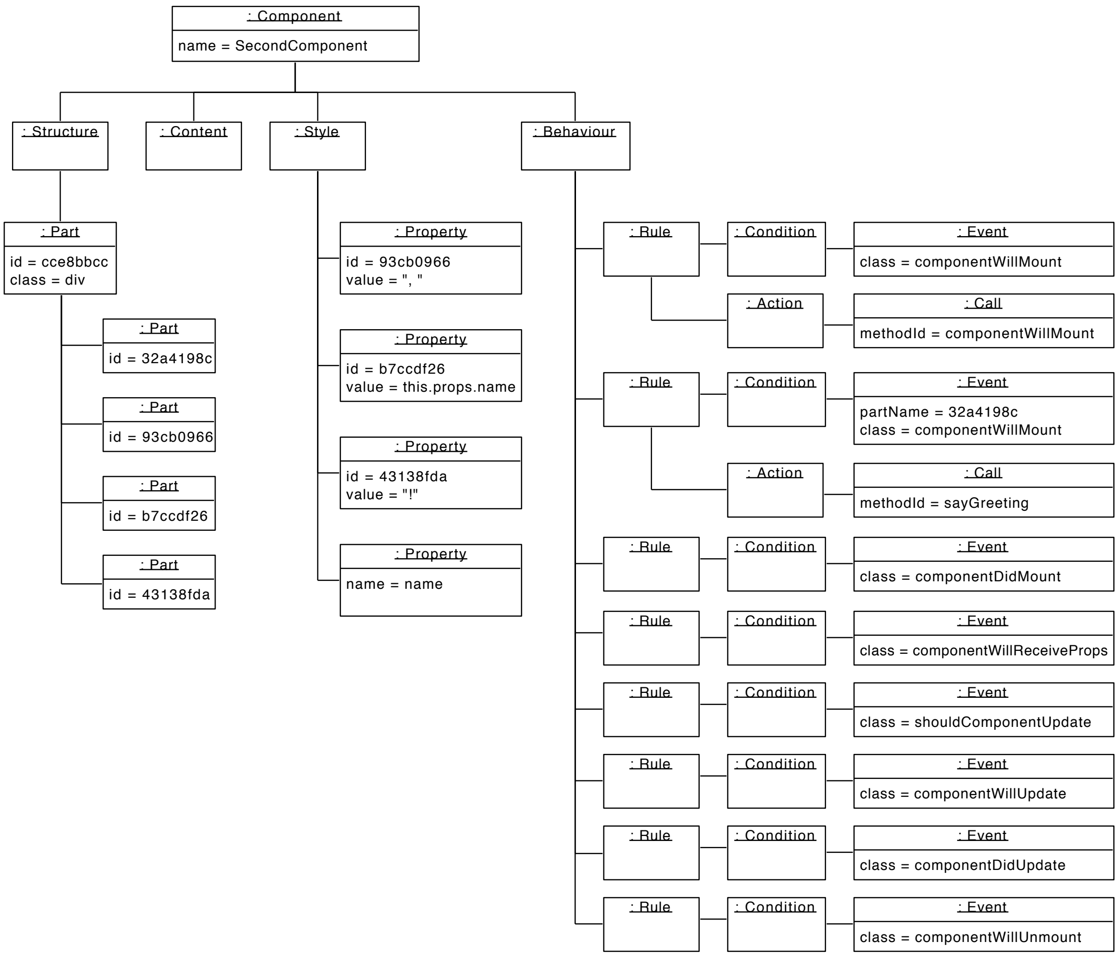


Figure 3.39.: Extracted view model of the React SecondComponent component

Columbus managed the extraction of all relevant information of the SecondComponent. The part with the id *32a4198c* was not referenced by a property, but a behaviour entity instead. The JavaScript source code defined an output of the `sayGreeting()` method in the `div` tag. The render function is evaluated upon the event `componentWillMount` and at the same time the output of the `sayGreeting()` method is determined as well. Because the rule is bound to a specific part, the corresponding event is contained in the model twice.

Results

The expected view model was compared to the model generated by Columbus. The left side of the bars in Figure 3.40 shows the expected number of entities for each type. To correctly model the HelloReact component, seventeen parts, sixteen rules, ten properties and four references were necessary. Out of those numbers, Columbus managed to extract nearly every entity correctly, as shown on the right hand side of each bar.

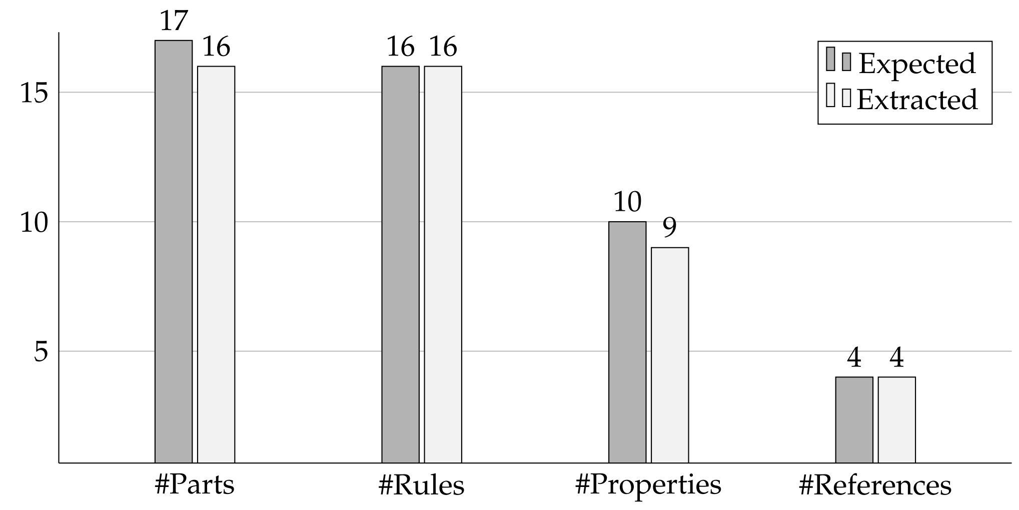


Figure 3.40.: Evaluation of extracted entities of the HelloReact project

3.3. Polymer

A Polymers component is declared in the `dom-module` tag. Each tag can contain a `template` and `script` definition as shown in Figure 3.41. Dependencies between components are declared at the beginning of the file with a `link` tag. The `script` tag contains the JavaScript component definition.

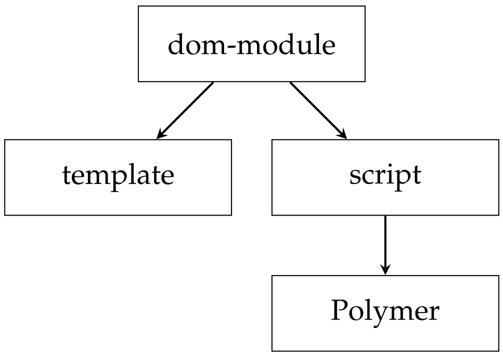


Figure 3.41.: Typical Polymer file content structure

3.3.1. Limitations

A typical Polymer component contains a template and script tag as shown in Figure 3.42.

```
<link rel="import" href="another_component.html">
<dom-module id="component-name">
  <template>
    <div><!-- ... --></div>
  </template>
  <script>
    Polymer({
      is: 'component-name',
      properties: {
        // ...
      },
      created: function () {
        // ...
      }
    });
  </script>
</dom-module>
```

Figure 3.42.: Example of a supported Polymer component definition

In order to extract the view model from Polymer components, the input files must be stripped of all non JavaScript content. This means that only the contents of the `script` tag should be present. The `link` import statements must be transformed into ECMAScript 6 `import` statements. The example from above would end up looking as shown in Figure 3.43.

```
import AnotherComponent from "another_component";

Polymer({
  is: 'component-name',
  properties: {
    // ...
  },
  created: function () {
    // ...
  }
});
```

Figure 3.43.: Refactored Polymer component for view model extraction with Columbus

Columbus is capable of extracting various elements of the component definition. All properties of a component must be declared inside the `properties` key. Each entry should contain the variable name as the key and an object with the variable definition containing a type hint and a possible default value. Any event listeners, aside from life cycle methods, should be listed in the `listeners` key. Alternatively, event listeners can by added by calling `addEventListener` as well. Life cycle methods can be declared in the usual fashion and will be extracted by Columbus.

3.3.2. Semantic Analyser

Polymer's component definitions inside the `script` tag must contain pure JavaScript code and therefore do not require a template transformation. The transformation with a compiler is applied nonetheless in order to add additional annotations to the code (Figure 3.44).

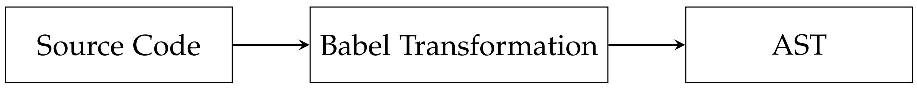


Figure 3.44.: Syntactical analyser steps for Polymer

3.3.3. Information Extraction

Extracting component declarations

Each polymer component is declared by calling the function `Polymer` and passing the definition as the first parameter. The name of a component is defined inside the object parameter under the key `is`. The query to retrieve each definition is shown in Figure 3.45.

```
[body] [type=CallExpression][callee.name="Polymer"]
```

Figure 3.45.: AST query for selecting Polymer() calls

Afterwards, the name is fetched by querying each component sub-tree with the expression shown in Figure 3.46.

```
[type="Property"][key.type="Identifier"][key.name="is"]
```

Figure 3.46.: AST query for selecting names of Polymer components

Extracting component properties

Declaring a property inside the `properties` key of a Polymer component provides several benefits, among others defining default values or event callbacks on value changes[10]. Each entry can be retrieved with the query shown in Figure 3.47. The relevant information is stored inside the sub-tree as object property and can be retrieved either by iterating through the object or querying for the value.

```
[type="Property"][key.type="Identifier"][key.name="properties"]>[
    properties]>[type="Property"]
```

Figure 3.47.: AST query for selecting the properties of a Polymer component

Extracting function declarations

The declared functions can be retrieved by the query shown in Figure 3.48. The parameters of each function are contained in the sub-tree as well and are mapped to the corresponding entry in the information base.

```
[arguments]>[properties]>[type="Property"][value.type="FunctionExpression"
    ]
```

Figure 3.48.: AST query for selecting all functions of a Polymer component

Extracting registered listeners

Listeners inside a component can be declared inside the `listeners` property of the component definition. Each definition consists of an element, a triggering event and a

[10]https://www.polymer-project.org/1.0/docs/devguide/properties

function to execute. Fetching all listeners can be achieved with the query Figure 3.49. The individual parts of the declaration are highlighted in Figure 3.50. It is possible to omit the element definition; in that case the event is registered for the whole component.

```
[type="Property"][key.type="Identifier"][key.name="listeners"]>[properties
    ]>[type="Property"]
```

Figure 3.49.: AST query for selecting all listeners of a Polymer component

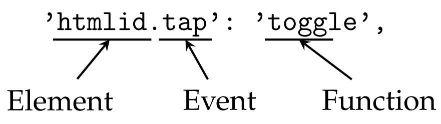


Figure 3.50.: Breakdown of listeners declaration for a Polymer component

3.3.4. Model Generation

The Polymer view model generation is based upon five information extraction rules as shown in Figure 3.51. Due to the inability to process the template the structure remains empty.

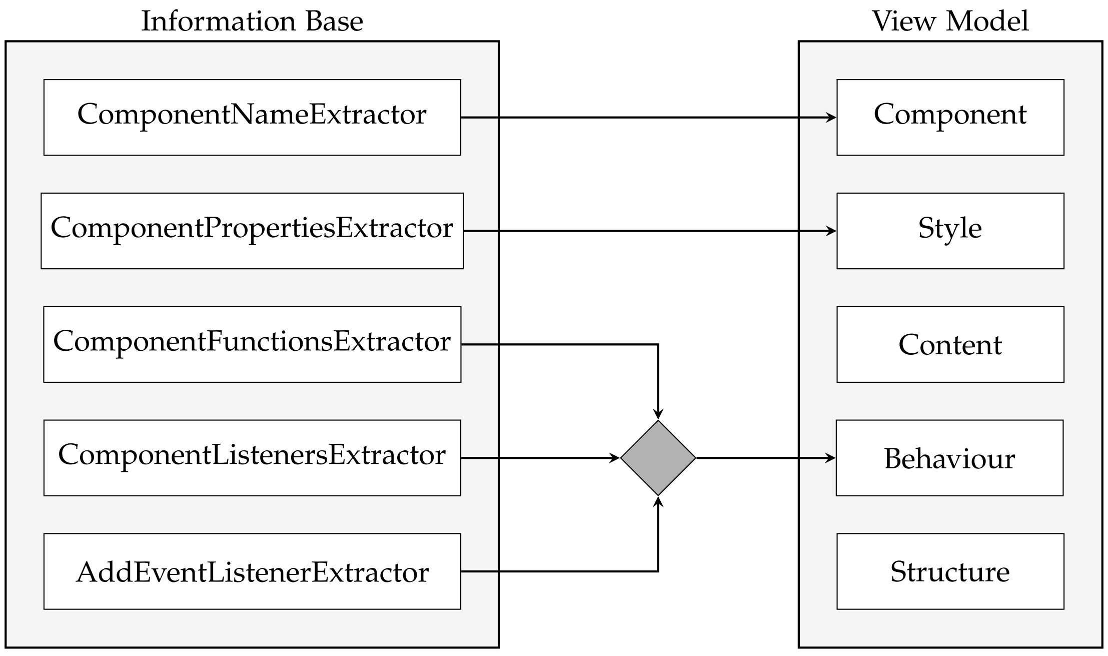

Information Base
ComponentNameExtractor
ComponentPropertiesExtractor
ComponentFunctionsExtractor
ComponentListenersExtractor
AddEventListenerExtractor
View Model
Component
Style
Content
Behaviour
Structure

Figure 3.51.: Model generation for Polymer

Adding component properties

The properties of a component are defined under the `properties` key in the component and contain a key-value pair for each declaration. The name, type and default value for each entry can be extracted with a single extraction rule. The information can be directly added to the view model as shown in Figure 3.52.

Source

```
properties: {
  dummy: {
    type: String,
  },
  pressed: {
    type: Boolean,
    value: false,
    notify: true,
  }
}
```

Information Base

```
{
  "name": "dummy",
  "type": "String"
},
{
  "name": "pressed",
  "type": "Boolean",
  "value": false
}
```

View Model

```
"style": {
  "properties": [
    {
      "_entity": "
         property",
      "name": "dummy",
      "type": "String"
    },
    {
      "_entity": "
         property",
      "name": "pressed",
      "type": "Boolean",
      "value": false
    }
  ]
}
```

Figure 3.52.: Extracting properties of Polymer components

Merging behaviour rules

The behaviour is populated by three entries in the information base: ComponentFunctionsExtractor, ComponentListenersExtractor and AddEventListenerExtractor. Each rule contributes to the same section of the view model, but relies on different source information from the component.

A list of all declared functions in the model are contained in the function extractor. This information set is filtered and all non life cycle methods are ignored. The remaining functions reflect which life cycle methods are declared in the component.

Custom user defined events are extracted by the listeners and addEventListeners extractors. Both rules target different a unique sets of events. The listeners are focused on user triggered events from the user interface like a tap or click on a specific element. The events listened to by the addEventListener are general events which are broadcast inside the components.

3.3.5. HelloPolymer Development Project

The HelloPolymer[11] example project was used during the creation of Columbus and implements examples of the supported functionality. The repository contains three Polymer components:

- x-app.js
- hello-world.js
- child-node.js

Each file containes only the JavaScript part of a Polymer component. The template was removed from the files because Columbus does not support non-JavaScript code.

Dependencies

The dependencies between the components were correctly extracted from the project (Figure 3.53).

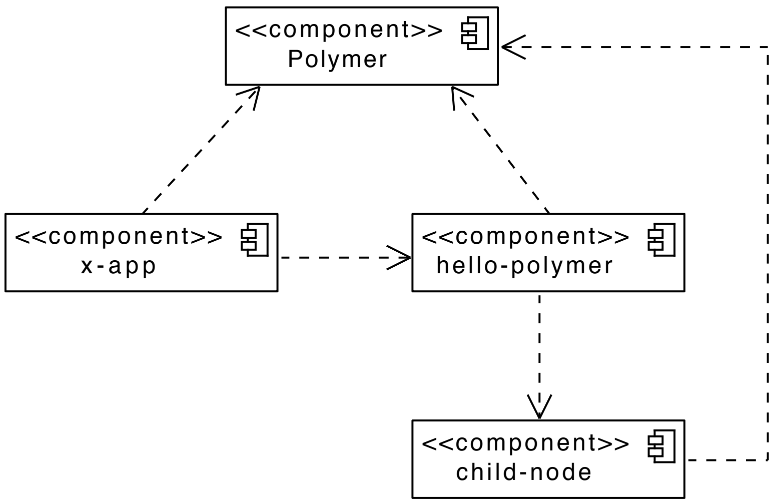


Figure 3.53.: Extracted dependencies of the HelloPolymer project

[11]https://github.com/tielitz/columbus-polymer-example

View Model xApp

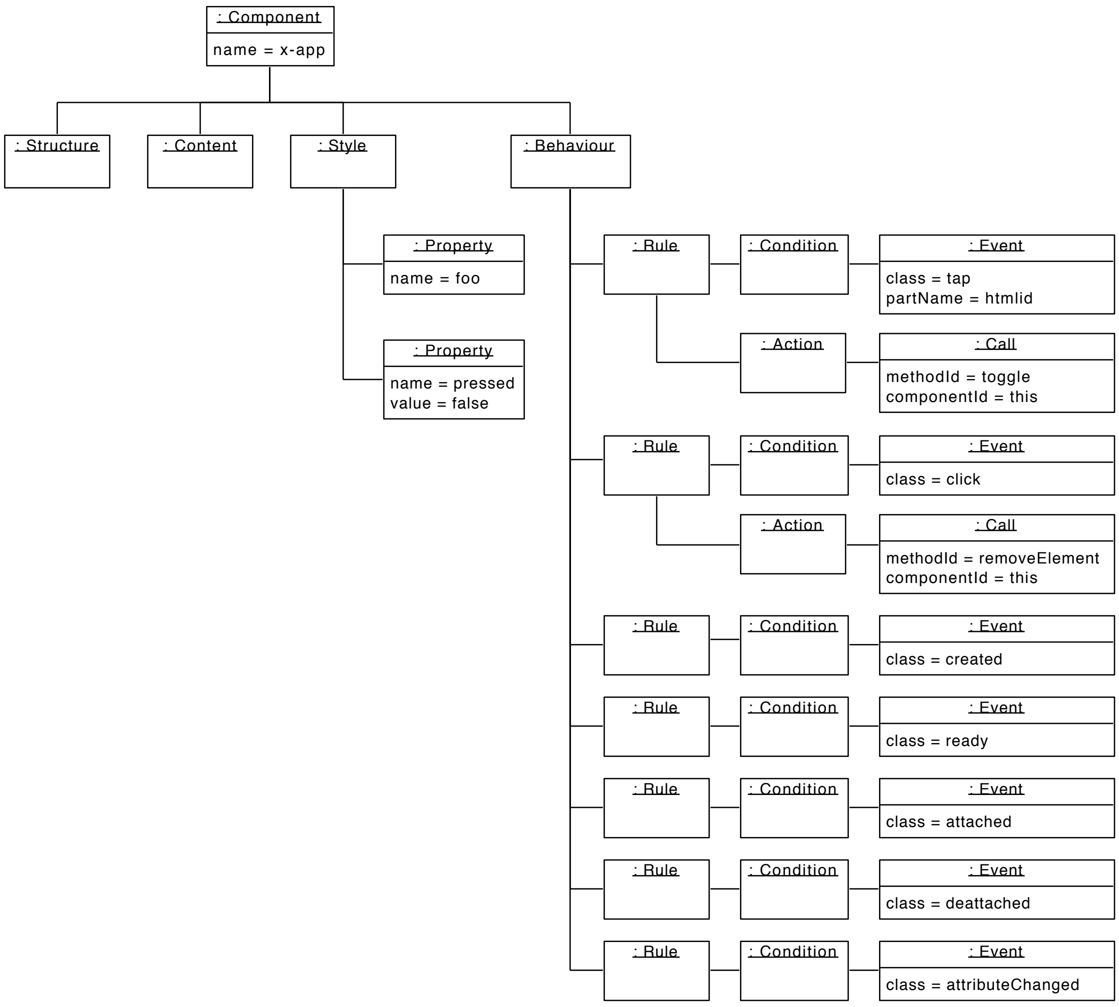


Figure 3.54.: Extracted view model of the Polymer xApp component

Columbus was able to extract the behaviour rules and part of the properties of the component, as shown in Figure 3.54. Extraction of the template is not possible for Polymer in general. React provided the benefit of an already existing JavaScript compiler, which translated the template into JavScript. At the time of development, no such compiler existed for Polymer. Without the information from the template, parts and references were missing from the model altogether. Additionally, properties and events defined in the template were also missing from the model.

```
...
// x-app.js
listeners: {
  'htmlid.tap': 'toggle',
  'click': 'removeElement'
},
...
// Alternative: this.addEventListener('eventName', this.methodToExecute);
```

Figure 3.55.: Excerpt of the file x-app.jsx

Figure 3.55 shows an example of a declared listener which is used in the xApp component. `htmlid` refers to the HTML tag id which identifies the targeted element and `tap` defines the event type. The value of the entry defines the executed method. If no dot is present in the key definition, the event is attached to the whole component itself. Columbus was able to extract both entries as shown in Figure 3.54. Columbus was also successful in extracting custom event listeners defined with `addEventListener`.

Results

The expected view model was compared to the model generated by Columbus. The left side of the bars in Figure 3.56 shows the expected number of entities for each type. Columbus was able to extract all entities from the code and add them to the view model.

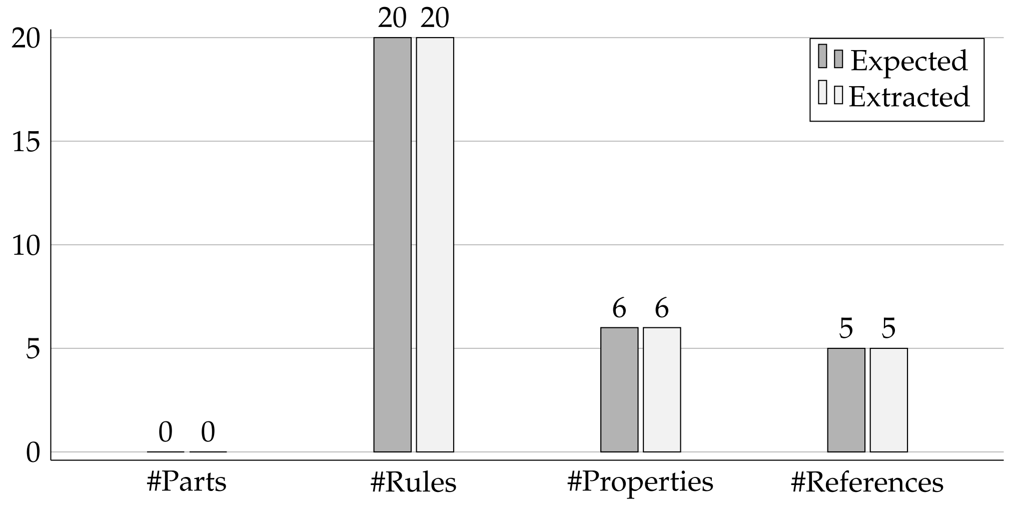


Figure 3.56.: Evaluation of extracted entities of the HelloPolymer project

3.4. AngularJS

AngularJS is a JavaScript framework which implements a model-view-viewmodel kind of architecture. It relies on data binding to keep model and view in sync. The version 1.5 of the framework introduced a component like construct, which originated from the directive.

3.4.1. Limitations

An example of a supported AngularJS component is shown in Figure 3.57.

```
import angular from 'angular';
import SecondComponent from 'second_component';

angular.module('myModule').component('componentName', {
  bindings: {
    ...
  },
  require: {
    secondComponentCtrl: "^secondComponent"
  },
  controller: function () {
    this.$onInit = function () {
      // ...
    };
    ...
  },
  template: '<div><second-component />...</div>'
});
```

Figure 3.57.: Example of a supported AngularJS component definition

In order for Columbus to be able to extract the view model from AngularJS components, the complete component definition must be placed in a single file. Each property of the definition must also be defined inline and without the use of variables to define the controller or template. Properties must be declared in the `bindings` section, while dependencies to other components must be declared in the `require` section. The registration in the AngularJS module system must always specify the structure starting with `angular`, followed by the module and then the component name.

3.4.2. Semantic Analyser

The contents of the files contain pure JavaScript. The goal of the source code transformation is purely the addition of annotation and unification of the structure. The process is identical to Polymer's and shown in Figure 3.58.

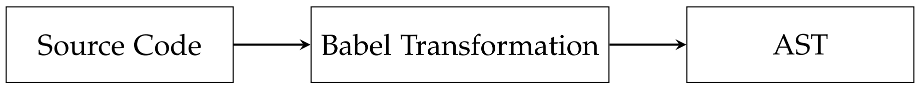


Figure 3.58.: Syntactical analyser steps for AngularJS

3.4.3. Information Extraction

Extracting component declarations

AngularJS declares properties by calling the function `component` on a module. The first parameter defines the name and the second parameter the contents of the component. Retrieving all declared components can be achieved with the query shown in Figure 3.59. The first argument of the call expression in the component sub-tree is the name of the component.

```
[type="ExpressionStatement"] [callee.property.name="component"]
```

Figure 3.59.: AST query for selecting AngularJS component definitions

Extracting component bindings

Variables which are accessible to outside components are declared inside the `bindings` key of the component definition object. Depending on the configured binding it is either declared as an in- or output variable. A list of all bindings can be retrieved with the query shown in Figure 3.60.

```
[type="Property"][key.name="bindings"] [properties]
```

Figure 3.60.: AST query for selecting bindings of an AngularJS component

Extracting component dependencies

An AngularJS component can define a dependency to another component inside the `require` property. Each entry defines the local variable name and the name of the referenced component. The entries can be retrieved with the query shown in Figure 3.61.

```
[type="Property"][key.name="require"] [properties]
```

Figure 3.61.: AST query for selecting component dependencies

Extracting properties from an AngularJS component controller

The properties are declared inside the controller of the AngularJS component. The controller function can be retrieved with the query shown in Figure 3.62.

```
[type="Property"][key.name="controller"][value.type="FunctionExpression"
    ]>[body]
```

Figure 3.62.: AST query for selecting the controller function of an AngularJS component

Variables which are used in conjunction with `this` or any other variable in the controller function are treated as a component property. The query in Figure 3.63 selects all variable definitions.

```
[id.name="controller"][body]>[body]>[type="ExpressionStatement"]
```

Figure 3.63.: AST query for selecting the properties of a Polymer component

Extracting function declarations from an AngularJS component controller

```
[type="Property"][key.name="controller"][value.type="FunctionExpression"
    ]>[body]
```

Figure 3.64.: AST query for selecting the controller function of an AngularJS component

Extracting the functions from the remaining sub-tree can be achieved through the query shown in Figure 3.65. The name and parameters of the function are returned and can be added to the information base.

```
[id.name="controller"][body]>[body]>[type="ExpressionStatement"][
    expression.right.type="FunctionExpression"]
```

Figure 3.65.: AST query for selecting all functions of an AngularJS component

3.4.4. Model Generation

The AngularJS view model generation is based upon five information extraction rules as shown in Figure 3.66.

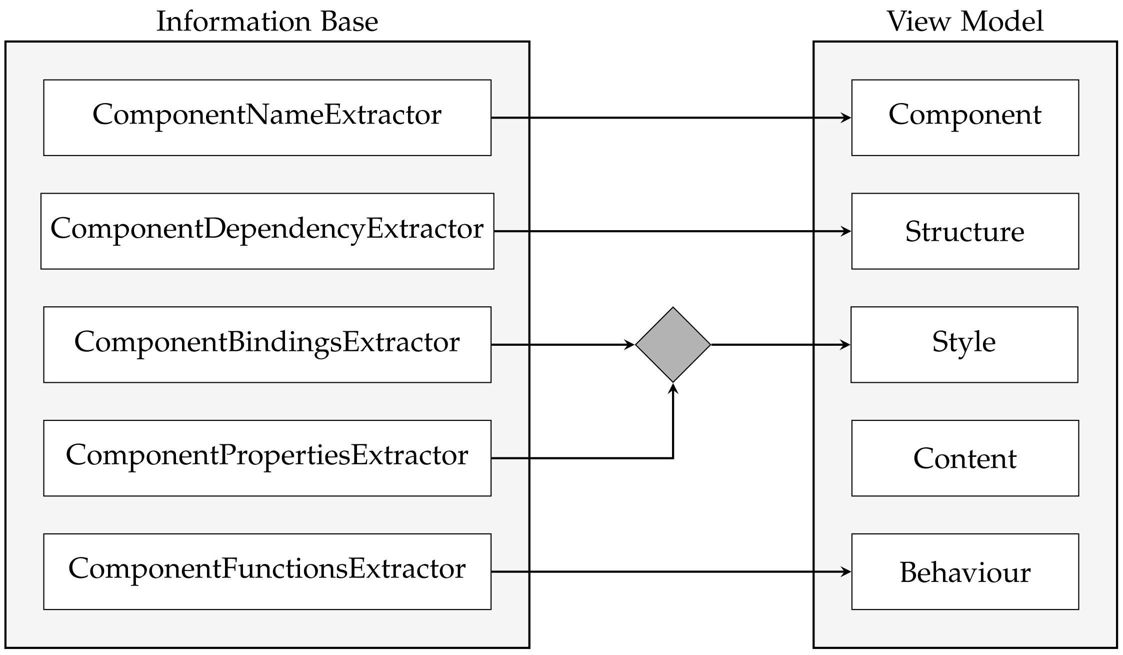

Information Base
View Model
ComponentNameExtractor
ComponentDependencyExtractor
ComponentBindingsExtractor
ComponentPropertiesExtractor
ComponentFunctionsExtractor
Component
Structure
Style
Content
Behaviour

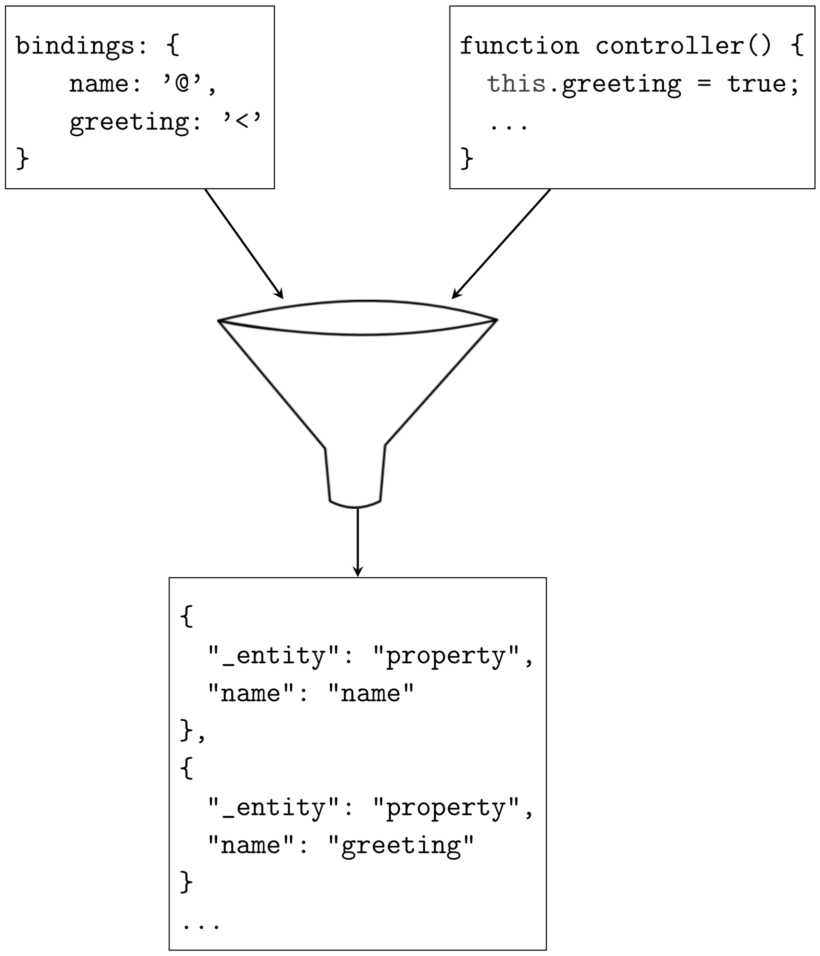
Figure 3.66.: Model generation for AngularJS

Populating the structure

Dependencies to other components can be declared in the `require` part of an AngularJS component. This information is reflected in the view model by adding each entry as a reference in the structure. It is not possible to analyse the template and determine the actual place where the dependency is used. Therefore, each required entry is added as a direct child to the structure entity.

Combining component properties

The properties of the view model are composed of the defined bindings of the component and the properties of the controller function (Figure 3.67). It is possible that both extraction rules contain duplicate entries which are merged and filtered out before they are added to the view model. The declared ways of binding and the assigned values of the properties are ignored during the view model generation process. The binding direction cannot be reflected in the current version of the view model.



```
bindings: {
    name: '@',
    greeting: '<'
}
```

```
function controller() {
  this.greeting = true;
  ...
}
```

```
{
  "_entity": "property",
  "name": "name"
},
{
  "_entity": "property",
  "name": "greeting"
}
...
```

Figure 3.67.: Combine and merge information base entries to create unambiguous variable definitions

Adding behaviour functions

The only behaviour rules which can be extracted from an AngularJS component are the life cycle calls. AngularJS relies heavily on defining custom events in the template which cannot be processed. In order to determine which callbacks are defined, the extraction rule extracts all declared functions inside the controller and the view model generator filters the information set based on the function name. If the function name matches a valid life cycle event, the action is added to the view model.

3.4.5. HelloAngular Development Project

The HelloAngular[12] example project was used during the creation of Columbus and gives an example of the supported functionality. The repository containes three AngularJS

[12]https://github.com/tielitz/columbus-angular-example

components.

- app-class.js
- hello-world.js
- hero-detail.js

Each file contains the complete definition for an AngularJS component.

Dependencies

The dependencies between the components can be extracted from the project (Figure 3.68).

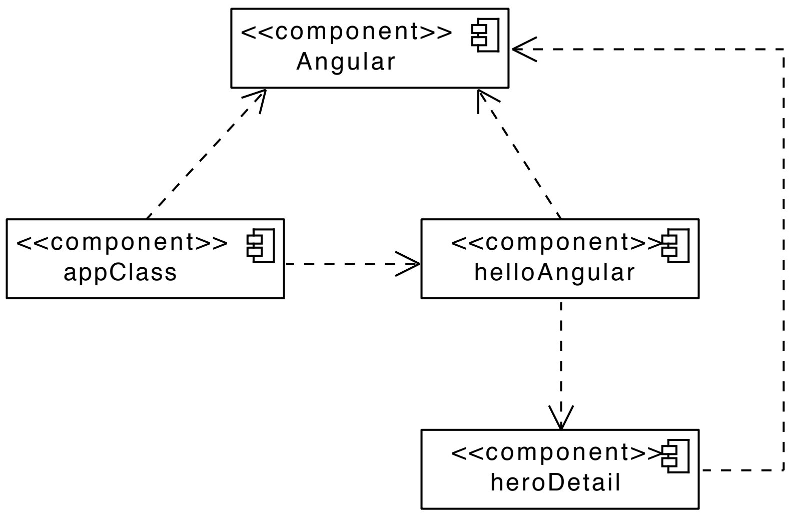


Figure 3.68.: Extracted dependencies of the HelloAngular project

View Model AppClass Component

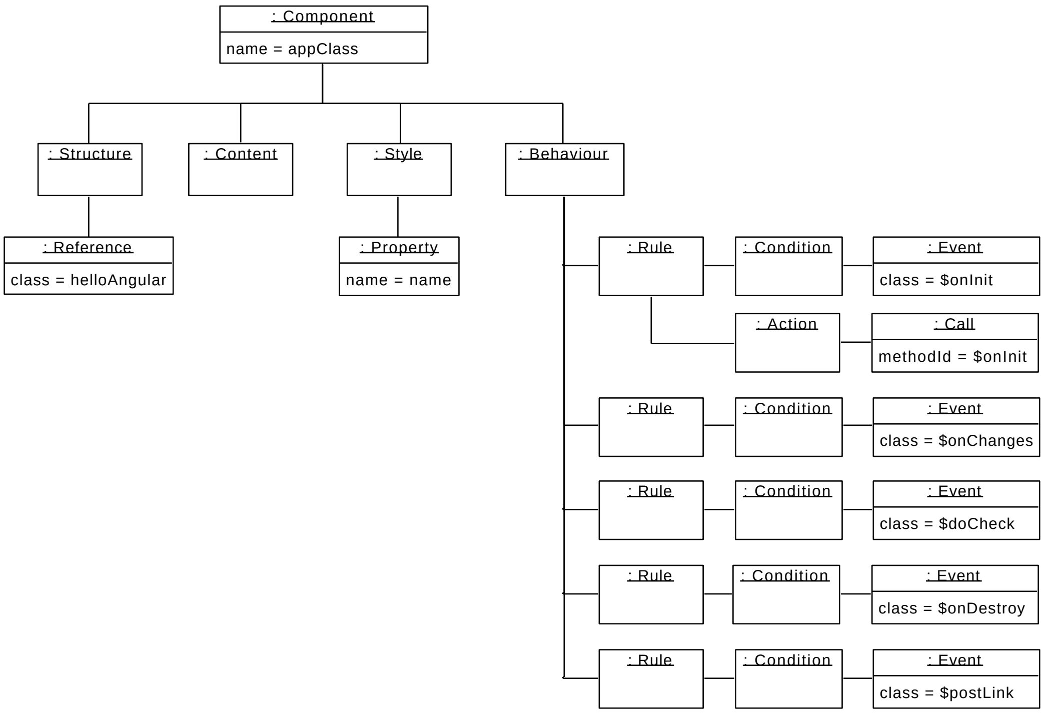


Figure 3.69.: Exemplary view model of the AppClass AngularJS component

View model extraction was limited to properties and behaviour rules. The proprietary AngularJS template syntax could not be processed by Columbus. If components declared dependencies in the `bindings` section of the component definition, they were added as references to the structure without any regards for the actual position or nesting inside the DOM. Since the template could not be parsed with Columbus, the inline and file URL template elements of the component were left out.

Columbus expected the component definition to be contained in one file, with each element defined in line. The extraction process would not be able to resolve references to variables or functions defined in another place or file.

AngularJS uses modules to group components together, which among others act as namespaces. That information was disregarded by Columbus since the view model does not provide any means to represent it. The view model, and therefore Columbus, expects each component name to be unique across the application.

The properties of the component were determined by the variables configured in the `bindings` section of the component definition and by variables declared in the controller function. The bindings define which properties are accessible to other components and also provide a declaration for the in- and outputs of each one. That information could not be fully incorporated into the view model. All properties were grouped together and added as a property to the model. Even bindings which had functions assigned to them were treated as properties and falsely added as properties. The same things happened to the declared variables inside the controller function. Each declaration was added eagerly to the model by Columbus, without an understanding of the used context. Often times a construct as shown in Figure 3.70 can be encountered. The variable `ctrl` is assigned the `this` context and all further declarations are assigned to that variable. Columbus was not able to determine which variable had assigned the controller context, but instead ignored the property of the object expression altogether. This resulted in expressions like `ctrl.foo` and `parent.bar` to be added as a component property of the current component to the view model. The same problem can occur when processing function declarations.

```
...
'controller': function () {
  var ctrl = this;
  ctrl.variable1 = 'Controller Variable';
  parent.variable2 = 'Parent Variable'; // falsely interpreted as ctrl
      variable
}
...
```

Figure 3.70.: Example of a common AngularJS practice to assign the current object context to a variable

The life cycle events were extracted and added to the model correctly. Aside from the declaration in the controller, no additional behaviour functions could be extracted. The lack of the template severely limits the possibilities of any event extractions. The appClass component defines a dependency in the `require` key, which results in a single reference entity in the structure. Columbus cannot verify the usage in the template, which makes this an assumption and is not guaranteed to work every time.

Results

The expected view model was compared to the model generated by Columbus. The left side of the bars in Figure 3.71 shows the expected number of entities for each type. Columbus was able to extract all entities from the code and add them to the view model.

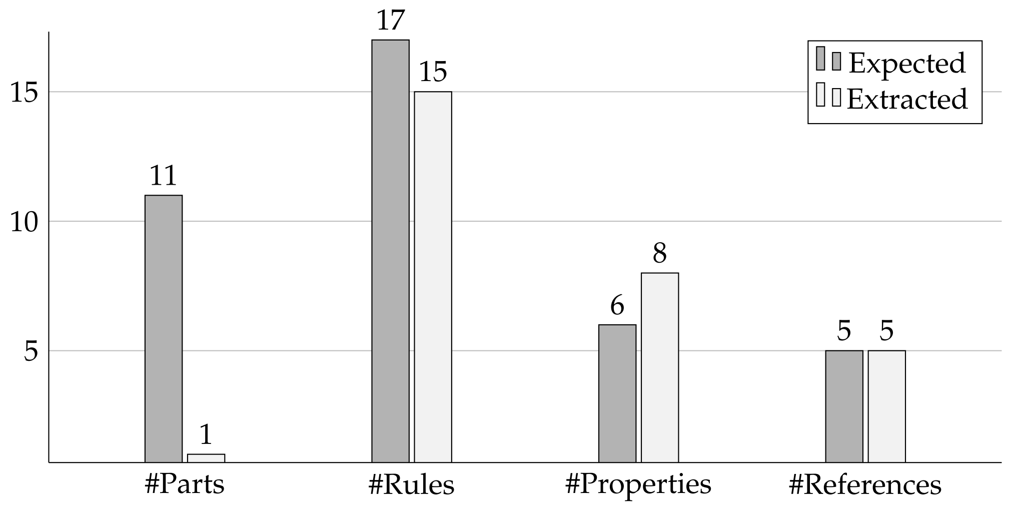


Figure 3.71.: Evaluation of extracted entities of the HelloPolymer project

4. Evaluation of Columbus

The goal of the evaluation is to assess the capabilities of Columbus to extract the view model from real-world projects. For each framework, multiple projects were selected for extraction and the resulting view models were verified and validated. Projects were selected from those hosted on GitHub which fulfilled most of the constraints set for each framework.

In total four projects were evaluated: TodoMVC , TUMitfahrer-WebApp, Contacts-Manager, Official Polymer Shop Example, and Angular 1.5 Component Architecture App. The TodoMVC was evaluated in depth, meaning that for each implementation and component a complete view model was created by hand and one generated by Columbus. Both view models were compared and deviations, and their causes, are stated in the evaluation. The following three projects were too comprehensive to model and analyse by hand, therefore only interesting facts, discovered problems and unintended behaviours were made note of.

4.1. TodoMVC

The TodoMVC[1] is a project which has been implemented in multiple JavaScript frameworks. It was originally designed to provide reference implementation in a variety of different frameworks. The architecture and provided functionality of TodoMVC are similar in every framework, which makes it a suitable project for the evaluation of Columbus. The individual evaluation of the implementations can therefore be compared with each other to assess the viability of the extraction process for the supported frameworks and libraries by Columbus.

TodoMVC provides an implementation for React[2] and Polymer[3]. An AngularJS component-based application was only available with Angular 2.0 and Typescript, which made it necessary to refactor the already existing AngularJS 1.4 project into a component-based architecture[4]. The application in general is a small application which can be used to manage tasks. The tasks themselves are stored in a remote database or in the local storage of the browser. The HTML structure is almost identical across all framework implementations. The components used to build the interface differ between each implementation, but they all share the architecture aspect of having two

[1]http://todomvc.com/
[2]https://github.com/tastejs/todomvc/tree/gh-pages/examples/react
[3]https://github.com/tastejs/todomvc/tree/gh-pages/examples/polymer
[4]https://github.com/tielitz/todomvc/tree/master/examples/angularjs

basic components: TodoApp and TodoItem, shown in Figure 4.1, although their naming differs between implementations.

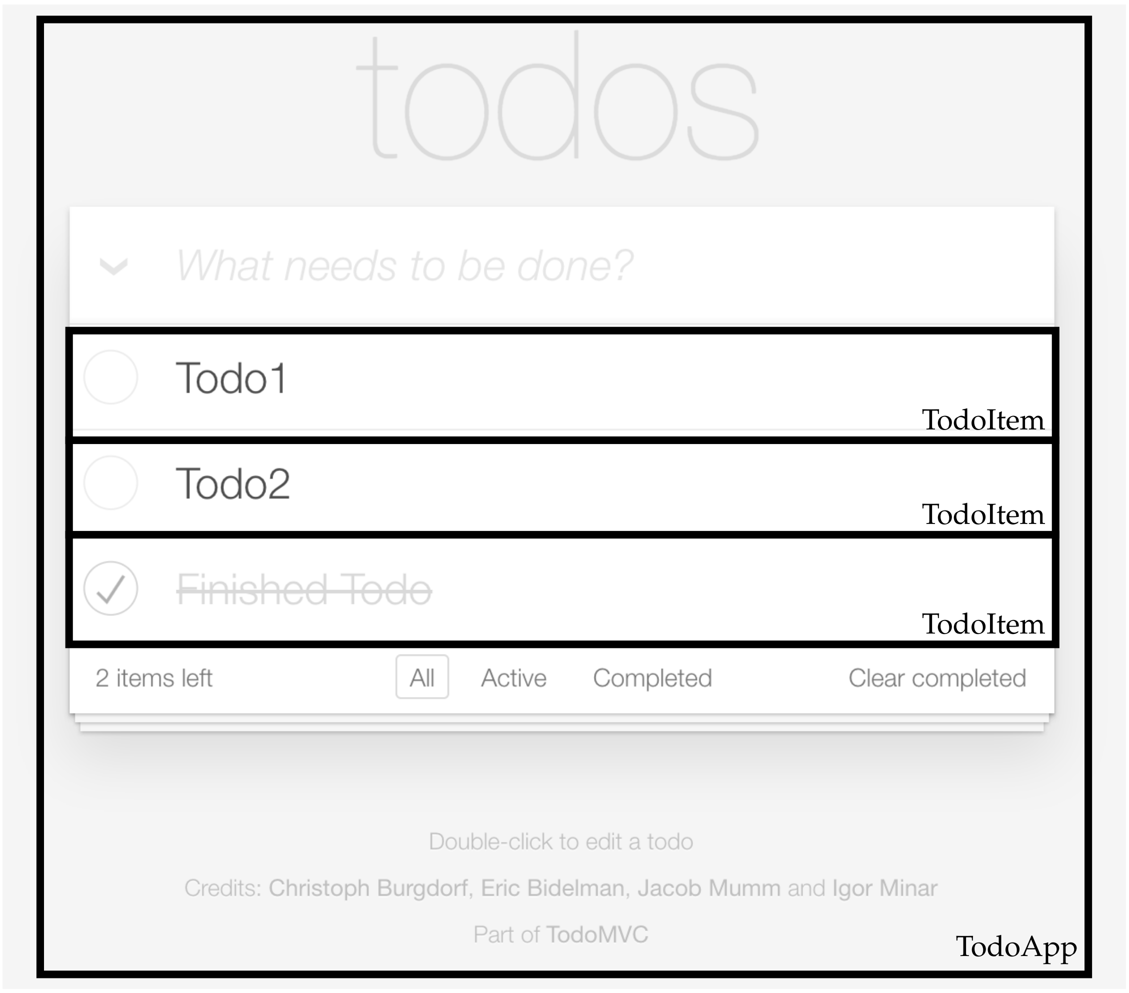


Figure 4.1.: Component composition of TodoMVC's user interface

4.1.1. Evaluation Approach

The evaluation of each framework/library implementation of TodoMVC was performed by comparing expected and actual view models with each other. The expected view models were created by hand and contained the complete component definition as far as the view model supported the found constructs.

The entities in both the hand-made and genereted view models were counted and compared. A detailed comparison of expected and actual entities provided insight into how successful Columbus was in extracting the view model. The entities were counted as correct if all attributes of the entity matched the reference in the hand-made model. Entities were counted as incorrect, if they were missing one or more attributes or were a wrong entity type altogether. Every entity not present in the generated model was counted as missing.

4.1.2. React

The React implementation of TodoMVC[5] divides the architecture into three components. The TodoApp component links together the child components TodoItem and TodoFooter. The TodoFooter encapsulates the filter parameters at the bottom of the todo list and shows a count of all unfinished todos.

The view model for each of the three component was first created by hand. The life cycle rules in the view model of the TodoApp (Figure A.1) and TodoItem (Figure A.3) are implied by the note connected to the behaviour in order to reduce the height of the figure. The TodoFooter in Figure A.5 shows the life cycle methods as actual entities. During creation of the models, certain constructs were discovered which are not supported by Columbus. These occourences will most likely cause problems during the extraction process and lead to incorrectly extracted entities.

The repository content of tastejs/todomvc[6] was fetched by Columbus and filtered with the regular expression `^examples/react/.*\.jsx$`. Columbus processed the three files *app.jsx*, *footer.jsx* and *todoItem.jsx*. The generated view model output of the extraction process was split up into each component and visualised in Figure A.2 for the TodoApp, Figure A.4 for the TodoItem and Figure A.6 for the TodoFooter. As expected, deviations from the defined conventions lead to incorrect or missing information in the view model. The following problems were identified when comparing the automatically generated view models by Columbus and the ones created by hand:

Problems identified with the structure

Multiple return statements in render function
Columbus uses static code analysis to extract information and is not capable of determining which return statement should be evaluated. Due to this limitation, the last return statement of the render function is used per default. All template parts that are not declared in the last return statement are not part of the view model.

Building the DOM piece by piece with variables
If part of the DOM is built beforehand and printed out as a variable in the final return statement, Columbus cannot extract the information contained in the variable. The complete DOM must be constructed inside the last return statement of the render function.

Misinterpreted references / missing parts
Columbus traverses through the template structure and tries to identify the nodes

[5]https://github.com/tastejs/todomvc/tree/gh-pages/examples/react
[6]Commit SHA ec8823c25437d206337b7c3450e9a3c82c530d40

visited. Depending on the type of node, different subsequent actions are taken, like differentiating between an HTML element and a component or determining if the current node contains any children. If the tool encounters an unexpected construct, like a variable output which contains further template tags, the traversal can not determine how to proceed. The fallback routine for these cases replaces unknown constructs with an empty part entity. No further traversal of children is possible for these misinterpreted nodes.

Problems identified with the style

Property usage without declaration
Columbus requires the component to list all properties in the `propTypes` section. If a property is used without a preceding declaration, Columbus attempts to detect it by tracking the `this` context in conjunction with variable usage. Special objects like `this.props` and `this.state` simplify the search process to some degree. Variables that start with one of these constructs can always be treated as component properties. The properties detection is currently limited to the render statement which means that some properties might not be detected.

Properties from missing parts
If a property was never used in the JavaScript part of the component, the detection is limited to all parts of the template detected during the corresponding extraction process. If a part was not correctly processed by the template extraction rule, any property is inevitably lost, and therefore unavailable during the view model generation.

Detecting default values
Columbus can only detect simple default values as long as they are declared in the `propTypes` and `getDefaultProps` section of the component. More sophisticated values, like output of a function or assignments of global variables, are not detected. In these cases the default value of a property remained blank.

Problems identified with the behaviour

Rules from missing parts
Columbus is only capable of extracting behaviour rules which are either declared in the JavaScript configuration or in the template. The extraction of the behaviour is not limited to a specific return statement in the render function, but instead tries to capture all usages. If the corresponding source part was not extracted due to the limitations listed before, the source id of the behaviour event does not point to a valid part of the view model. Nonetheless, each rule contains a unique part id which hints at an existing behaviour.

Results

The quantitative results of the TodoMVC React evaluation are shown in Table 4.1. The possibility to analyse the template in React resulted in a high extraction rate of properties and behavioural rules. The missing and incorrect entries are mostly due to the misinterpreted part constructs which are not supported by Columbus.

Table 4.1.: Evaluation of extracted entities of the TodoMVC React application with Columbus

	Correct	Incorrect	Missing	Total	Rate
Parts	32	2	2	**36**	88%
References		1	1	**2**	0%
Properties	24	1	2	**27**	88%
Behaviour	16			**16**	100%
Life cycle	21			**21**	100%

4.1.3. Polymer

The Polymer implementation of TodoMVC divides the application into four components: td-todos, td-item, a dedicated component for the model, called td-model, and td-input. Each file is written in the Polymer template format, meaning that the JavaScript content is enclosed in a `script` tag. First the files had to be stripped of all non-JavaScript content before the project could be evaluated with Columbus. An example of the applied refactoring is shown in Figure 4.2. Everything, except the contents of the `script` tag, was removed from the file. Any import statement was transformed into the ECMAScript 6 `import` statement. The modified files are published on GitHub[7]. No alterations were made to the component definitions.

[7]https://github.com/tielitz/todomvc/tree/master/examples/polymer/refacelements

```
// component.html
<link rel="import" href="other-
    component.html">
<dom-module id="componentId">
  <template>
    <style>
      /* styles */
    </style>
    <!-- HTML -->
  </template>
  <script>
     Polymer({
      // component definition
     });
  </script>
</dom-module>
```

```
import OtherComponent from "other-
    component";
// component.js
Polymer({
  // component definition
});
```

Figure 4.2.: Applied refactoring to the TodoMVC Polymer application

The view model for each component was created by hand. The life cycle rules in the view model of the td-todos (Figure A.7) and td-item (Figure A.9) are implied by the note connected to the behaviour in order to reduce the height of the figure. The td-input in Figure A.11 and td-model in Figure A.13 show the life cycle methods as actual entities. The relationship between the components is shown in Figure 4.3. The main `td-todo` component uses the `td-item` to display the list entries of the todos and the `td-input` component extends the standard input element with custom behaviour.

The repository content of the refactored version at tielitz/todomvc[8] was fetched by Columbus and filtered with the regular expression `^examples/polymer/refacelements /.*\.js$`. Columbus processed the four files *td-todos.js*, *td-input.js*, *td-item.js* and *td-model.js*. The generated view model output of the extraction process was split up into each component and visualised in Figure A.8 for the td-todos, Figure A.10 for the td-item, Figure A.12 for the td-input, and Figure A.14 for the td-model component. The following problems were identified when comparing the generated view models by Columbus and the ones created by hand:

[8]Commit SHA c1e9587169843a9ff3b5519ac91855738c203273

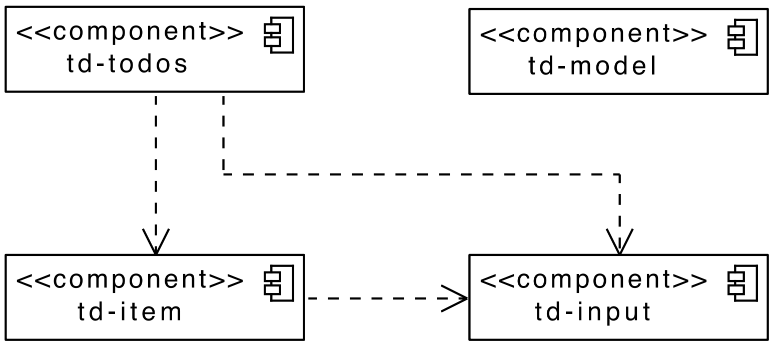


Figure 4.3.: Component diagram of the TodoMVC Polymer project

Problems identified with the structure

Unable to parse the template

The proprietary template syntax of Polymer is not supported by Columbus, which means that all information from the template was unavailable to the extraction process.

Problems identified with the style

Missing properties

Without the template, any property defined in the DOM is inevitably lost to the view model generation. This is critical to ordinary text fragments in HTML elements, since these are not defined anywhere inside the component definition.

Property usage without declaration

Columbus requires the component to list all properties in the `properties` section. If properties are used without declaration, Columbus attempts to detect them by tracking the this context in conjunction with a variable usage, but it cannot guarantee that all variables are detected.

Problems identified with the behaviour

Rules from missing parts

Without the template, any behavioural rules defined in the DOM are inevitably lost. The requirements for view model extraction of Polymer projects requires the usage of the `listeners` section to define event listeners.

Results

The quantitative results of the TodoMVC Polymer evaluation are shown in Table 4.2. Without the possibility to analyse the template, the extraction process could not extract most of the properties and behavioural rules. Only two of the behaviour rules were defined according to the conventions for Polymer application required by Columbus.

Table 4.2.: Evaluation of extracted entities of the TodoMVC Polymer application with Columbus

	Correct	Incorrect	Missing	Total	Rate
Parts			51	**51**	0%
References			3	**3**	0%
Properties	11		9	**20**	55%
Behaviour	2		16	**18**	11%
Life cycle	20			**20**	100%

4.1.4. AngularJS

TodoMVC provides an implementation in AngularJS in either version 1.4 or as component-based application in version 2. Neither are suitable for the evaluation without refactoring. Due to that, a modified version was created and published on GitHub[9]. The refactored application implements the two components todoApp and todoItem which are implemented in the same fashion as the React and Polymer components.

The view model for both components was created by hand first. The model for the todoApp is shown in Figure A.15 and todoItem in Figure A.17.

The repository content of tielitz/todomvc[10] was fetched by Columbus and filtered with the regular expression `^examples/angularjs/js/components/.*\.js$`. Columbus processed the two files *todoApp.component.js* and *todoItem.component.js*. The generated view model output of the extraction process was split up into each component and visualised in Figure A.16 for the todoApp and Figure A.18 for the todoItem component. The following problems were identified when comparing the generated view models by Columbus and the hand created ones:

Problems identified with the structure

Unable to process the template

The proprietary template syntax of AngularJS is not supported by Columbus, which means that all information from the template was unavailable to the extraction process.

[9]https://github.com/tielitz/todomvc/tree/master/examples/angularjs
[10]Commit SHA 39ccb61a054bb6068fe5e9cd697e72fb66005b1b

Problems identified with the style

Missing properties
Without the template, any property defined in the DOM is inevitably lost to the view model generation. This is critical to ordinary text fragments in HTML elements, since these are not defined anywhere inside the component definition.

Property usage without declaration
Columbus requires the component to list all properties in either the `bindings` section or as a variable declaration in the constructor function. If properties are used without declaration, Columbus attempts to detect them by tracking the `this` context in conjunction with variable usage but it cannot guarantee that all variables are detected.

Function properties
Some of the properties are functions which are passed down from parent components. Currently the view model does not differentiate between a property which is an attribute or a function. It is necessary to alter the model in order to correctly reflect the different properties.

Problems identified with the behaviour

Rules from missing parts
Without the template, any behavioural rules defined in the DOM are inevitably lost.

Results

The quantitative results of the TodoMVC AngularJS evaluation are shown in Table 4.3. Without the possibility to analyse the template, the extraction process could not extract most of the properties and any behavioural rules. AngularJS relies heavily on annotations on template tags to define custom event listeners which cannot be processed by Columbus.

Table 4.3.: Evaluation of extracted entities of the TodoMVC AngularJS application with Columbus

	Correct	Incorrect	Missing	Total	Rate
Parts			36	**36**	0%
References			1	**1**	0%
Properties	7		14	**21**	50%
Behaviour			12	**12**	0%
Life cycle	10			**10**	100%

4.2. Mobility-Services-Lab/TUMitfahrer-WebApp

The TUMitfahrer application was a real life application developed by students at the TU München. It was developed using the React library. The TUMitfahrer web application[11] is composed of over thirty individual components. The evaluation was performed on the latest version[12] of the repository and restricted to *.jsx* and *.js* files in the *src/components* subdirectory.

Dependencies

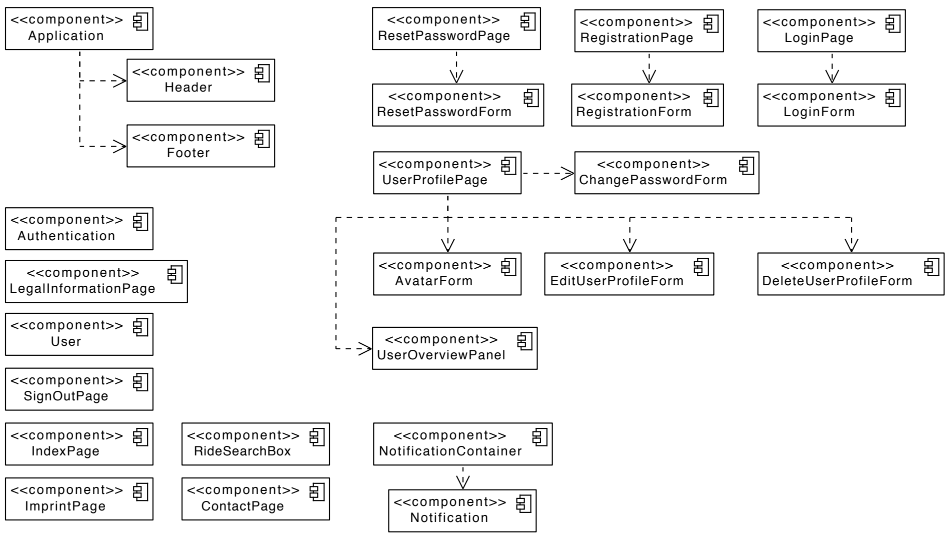
Columbus requires dependencies to be defined with the ECMAScript 6 import statement, but the project used the previous `require` statement. It was not expected to extract any dependencies at all, due to the deviation from the requirements. Coincidentally, the JavaScript compiler would have translated the import statement into a construct with a similar syntax and the same `require` statement. Figure 3.14 shows the transformation between `import` and `require` syntax.

The dependency extractor managed to extract all dependencies between the components. An excerpt of the extracted dependencies is shown in Figure 4.4. For dependencies to components in different directories, the `require` path contained path expressions to navigate within the folder structure. The entries in the dependencies section, can contain path expressions which are capable of traversing the folder structure. Columbus was not capable of resolving path expressions.

[11]https://github.com/Mobility-Services-Lab/TUMitfahrer-WebApp
[12]Commit SHA 71838eefe07f710f46f533ac7362caec11584dc6

```
...
"src/components/users/userProfilePage.jsx": {
  "components": [
    "UserProfilePage"
  ],
  "dependencies": [
    "react",
    "react-intl",
    "./changePasswordForm.jsx",
    "./editUserProfileForm.jsx",
    "./deleteUserProfileForm.jsx",
    "./userOverviewPanel.jsx",
    "./avatarForm.jsx"
  ]
}
...
```

Figure 4.4.: Extracted dependency excerpt of the Mobility-Services-Lab/TUMitfahrer-WebApp project



<<component>> Application
<<component>> Header
<<component>> Footer
<<component>> Authentication
<<component>> LegalInformationPage
<<component>> User
<<component>> SignOutPage
<<component>> IndexPage
<<component>> RideSearchBox
<<component>> ImprintPage
<<component>> ContactPage
<<component>> ResetPasswordPage
<<component>> ResetPasswordForm
<<component>> RegistrationPage
<<component>> RegistrationForm
<<component>> LoginPage
<<component>> LoginForm
<<component>> UserProfilePage
<<component>> ChangePasswordForm
<<component>> AvatarForm
<<component>> EditUserProfileForm
<<component>> DeleteUserProfileForm
<<component>> UserOverviewPanel
<<component>> NotificationContainer
<<component>> Notification

Figure 4.5.: Extracted dependencies of the TUMitfahrer project

Figure 4.5 shows the dependency graph after dependencies to external components were filtered out and the directory structure was flattened.

View Model

The view model generation successfully extracted most of the information from each file. Information, like component declarations, life cycle methods or proptypes was present in all extracted components. Using the `require` statement to declare dependencies resulted in the correct naming of the dependencies inside the component. No alterations to the names of the imported components were performed by the compiler.

```
...
// src/components/auth/resetPasswordForm.jsx
render: function() {
  return (
    <Formsy.Form onValidSubmit={this.submitForm} ...>
      <FRC.Input name="email" ... />
...
```

Figure 4.6.: Excerpt of the file resetPasswordForm.jsx

Figure 4.6 shows an excerpt of a component that included the Formsy library to generate a form. The extraction process of Columbus was not able to correctly determine which specific component of the library was used. The responsible rule only extracted the first part of the definition before the dot and the model listed `Formsy` as the component instead of `Formsy.Form`. The same issue occurred for the input element as well. Nonetheless, even though the underlying element was incorrect, the custom events like `onValidSubmit` were correctly extracted and added to the model.

```
...
// src/components/auth/authentication.jsx
render: function() {
  return (
    <div>
      {React.cloneElement(this.props.children, {user: this.props.user,
          ...})}
    </div>
  );
}
...
```

Figure 4.7.: Excerpt of the file authentication.jsx

Figure 4.7 shows an example when the extraction of the structure failed. The method `React.cloneElement` alone did not provide any information towards which component should be rendered. `this.props.children` was not declared before and the extraction process could only determine the type of the variable.

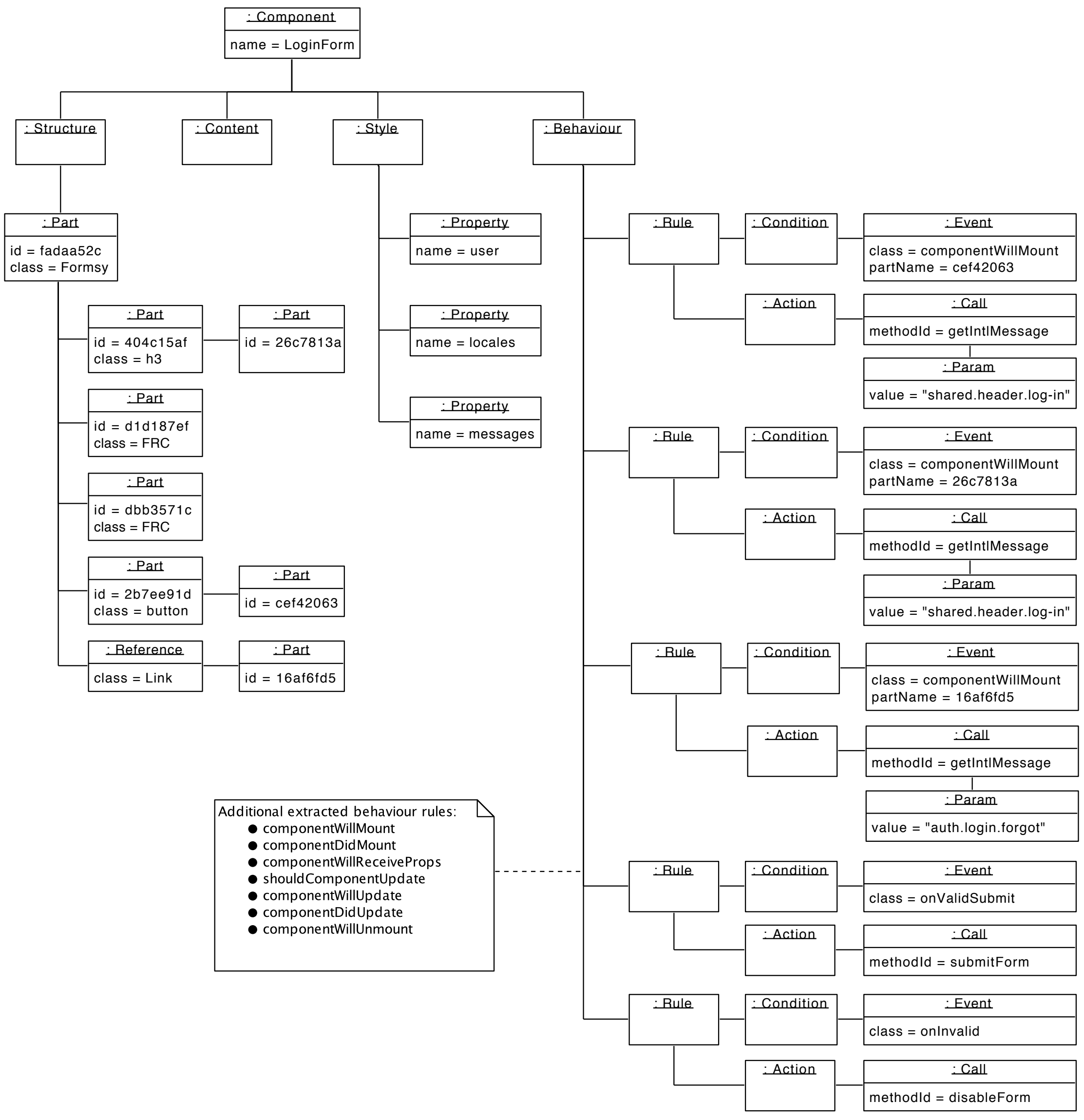


Figure 4.8.: Extracted view model of the LoginForm component

The generated view model in Figure 4.8 shows the model for the login form component without the undeclared life cycle behaviour events. This component relied on internationalisation to show textual information in various languages. Therefore, Columbus was not able to extract the fragments as part of the style, but rather as behaviour. Whenever a translation for a text was needed, the `getIntlMessage` method was evaluated at the *componentWillMount* event which was correctly reflected in the view model. In addition to the event and call, the parameter, or in that case internationalisation key, was also extracted and added to the model.

4.3. Official Polymer Shop Example

The Polymer shop project [13] is an example implementation of a component-based web application. The whole application is composed of over thirty individual components. A typical Polymer component contains the template and script tags inside a dom-module element, but Columbus is not capable of extracting the contents of the script tag itself. Due to this limitation, a modified shop example[14] was created in which the files were stripped of all non-JavaScript fragments. Without that groundwork, the Polymer-based project could not be evaluated. The evaluation was performed on the latest version[15] of the repository.

Dependencies

Dependencies in Polymer are usually defined with an HTML link tag as shown in Figure 4.9. This syntax could not be processed by Columbus. The semantic analyser required a valid JavaScript syntax and was not able to process the foreign tag. It was necessary to transform the import syntax into an ECMAScript 6 compliant import statement before any further extraction took place.

```
// Typical import tag in Polymer
<link rel="import" href="shop-home.html">

// Transformed
import ShopHome from 'shop-home';
```

Figure 4.9.: Excerpt of the file x-app.jsx

After the transformation, Columbus was able to correctly determine the dependencies between the components. In general, the project relied heavily on third party libraries, which would be ignored by Columbus for the view model generation. Without a component definition in the input source files, Columbus will not create a blank view model for required components.

View Model

The view model in Figure 4.10 shows a partial model of the shopApp component. The remaining lifecycle events were left out to keep to model complexity manageable.

[13]https://github.com/Polymer/shop
[14]https://github.com/tielitz/columbus-polymer-modifiedshop
[15]Commit SHA 32e9b83831231344f7febfa29d7b8e604eea01e5

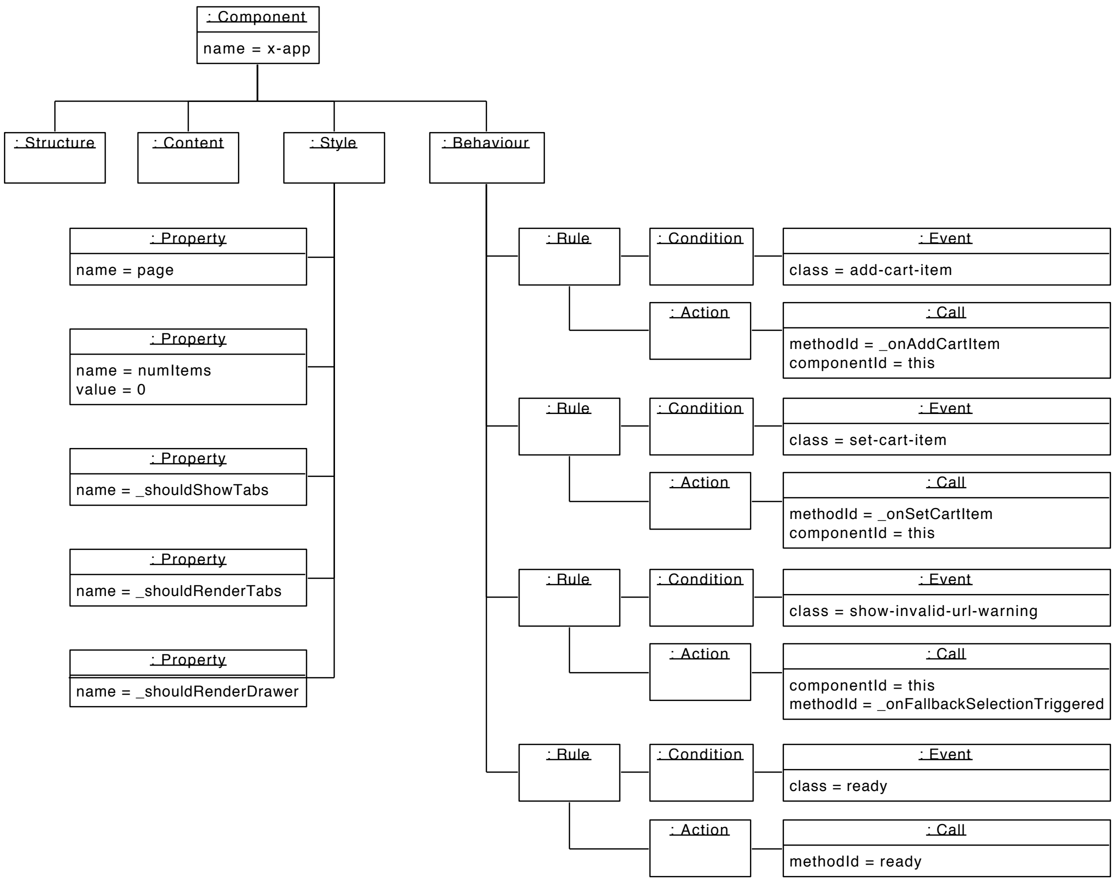


Figure 4.10.: Extracted view model of the shopApp component

Columbus managed to extract the name of all five declared properties. The component defined the value of the last three variables as the computed result of a method call, which is beyond the capabilities of the extraction tool. The page property had an observer callback assigned, which was not extracted and was missing in the behaviour of the model. In fact, all observers of the component (Figure 4.11) could not be detected, including the registered callbacks in the `observers` entry.

```
...
// src/shop-app.js
properties: {
  page: {
    type: String,
    reflectToAttribute: true,
    observer: '_pageChanged'
  },
},
observers: [
  '_routePageChanged(routeData.page)'
],
...
```

Figure 4.11.: Excerpt of the file shop-app.js

The events and callback methods in the `listeners` tag were extracted by Columbus and added in the model (Figure 4.12). Columbus was not able to correctly identify the parameters of the callback method. Therefore, some behaviour rules were missing the parameters defined in the method signature.

```
...
// src/shop-app.js
listeners: {
  'add-cart-item': '_onAddCartItem',
  ...
},
_onAddCartItem: function(event) {
  ...
};
...
```

Figure 4.12.: Excerpt of the file shop-app.js

4.4. Angular 1.5 Component Architecture App

The project Angular 1.5 components app[16] was built to showcase the AngularJS framework and the usage of components in version 1.5. Each component definition is split into three files: controller, component definition and template. That structure was incom-

[16]https://github.com/toddmotto/angular-1-5-components-app

patible with Columbus and a merged version of the project[17] was evaluated instead. The applied refactoring is shown in Figure 4.13. The references in the component definition were replaced by the implementation of the controller and the content of the template. The evaluation was performed on the latest version[18] of the repostory.

```
// contact.component.js
var contact = {
  templateUrl: './contact.html',
  controller: 'ContactController'
};
angular
  .module('components.contact')
  .component('contact', contact);

// contact.controller.js
function ContactController() {
  // ...
}
angular
  .module('components.contact')
  .controller('ContactController',
      ContactController);

// contact.html
<div><!-- HTML template --></div>
```

```
// contact.component.js
angular
  .module('components.contact')
  .component('contact', {
    template: '<div><!-- HTML
        template --></div>',
    controller: function
        ContactController() {
      // ...
    }
  });
```

Figure 4.13.: Applied refactoring to the AngularJS component definitions in the application

Dependencies

The project did not use any import statements to resolve dependencies, but rather relied on the AngularJS dependency injection. Columbus was not able to extract any component dependencies at all.

View Model

The extraction of the view model suffered from similar problems as the HelloWorld application did. At least the refactoring of the component declarations allowed Columbus to extract properties and behaviour rules.

[17]https://github.com/tielitz/angular-1-5-components-app
[18]Commit SHA 03e6991abc2e8ea05f8e4872c01dd85f37c0b5f4

The project relied on bindings to connect and call methods in other components. Columbus assumed that every property declared in the `bindings` element was a property, but additionally, they could also be functions. As shown in Figure 4.14, the component *authForm* declared a function `onSave` as a binding. In this example, Columbus added the property *onSubmit* to the view model.

```
angular.module('components.auth').component('authForm', {
    bindings: {
      onSubmit: '&'
    },
...
```

Figure 4.14.: Function binding in the authForm component

4.5. Analysis of Evaluation

The evaluation showed that the model extraction of Columbus is not sufficient enough to be used in real-world projects. The severely limited capabilities work well in a controlled environment but lack sophistication and diversity. React provided the best support for the extraction, thanks to an available JavaScript compiler for the JSX syntax. AngularJS and Polymer contributed very limited information to the model, at least until a comparable compiler for the template was used.

The extraction of the components and their life cycle methods worked across all projects, as long as they adhered to the required structure of component definitions. In the same way, Columbus managed to extract nearly all properties, only struggling with function bindings of AngularJS.

Taking the evaluation of each project into account, the major feature improvements of Columbus would be:

1. Support for definitions of variables and functions to be split upon multiple files. Often times the template is separated into its own HTML file and not provided inline in the JavaScript files. This affects AngularJS and React in particular. In order to parse Polymer, Columbus should be able to process the HTML dom-module structure and separate template and JavaScript code.

2. Compiler for AngularJS and Polymer's template syntax.

3. Future-proofing Columbus by supporting the new ECMAScript 6 syntax.

5. Conclusion and Future Work

It was successfully shown that it is possible to create a view model extraction process for web-applications. The definitions of the view model itself turned out to be sufficient for the first development iteration, but refinement is necessary in order to support the full range of functionalities offered by modern applications.

The development of a view model extraction tool turned out to be more complex than originally anticipated. Strict limitations on the allowed input format of source code severely impair the applicability of the application outside of scientific purposes. The extraction logic must become more sophisticated before it can be used in real world projects. Modern web-applications tend to deviate from others in their architecture and functionality, which makes it difficult to support a wide variety of systems. The process itself is promising and was designed to be extendable in the future.

So far the development process shown in Figure 5.1 has been completed once. The results of the evaluation conducted should be taken into account when refining the view model definition. Future iterations will eventually lead to a complete model.

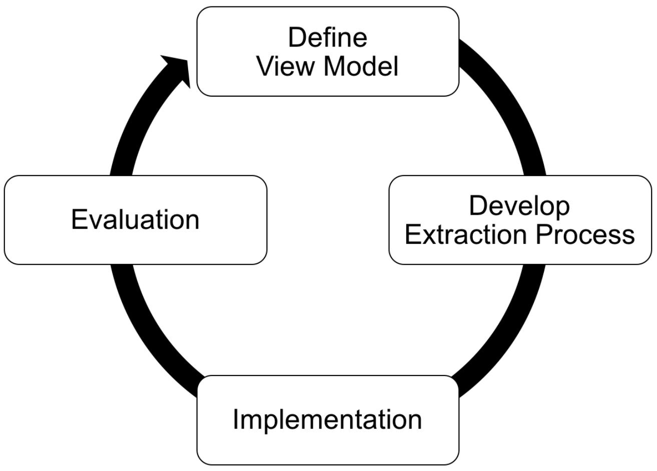


Figure 5.1.: Continuous development process

A. Appendix

A.1. TodoMVC View Models

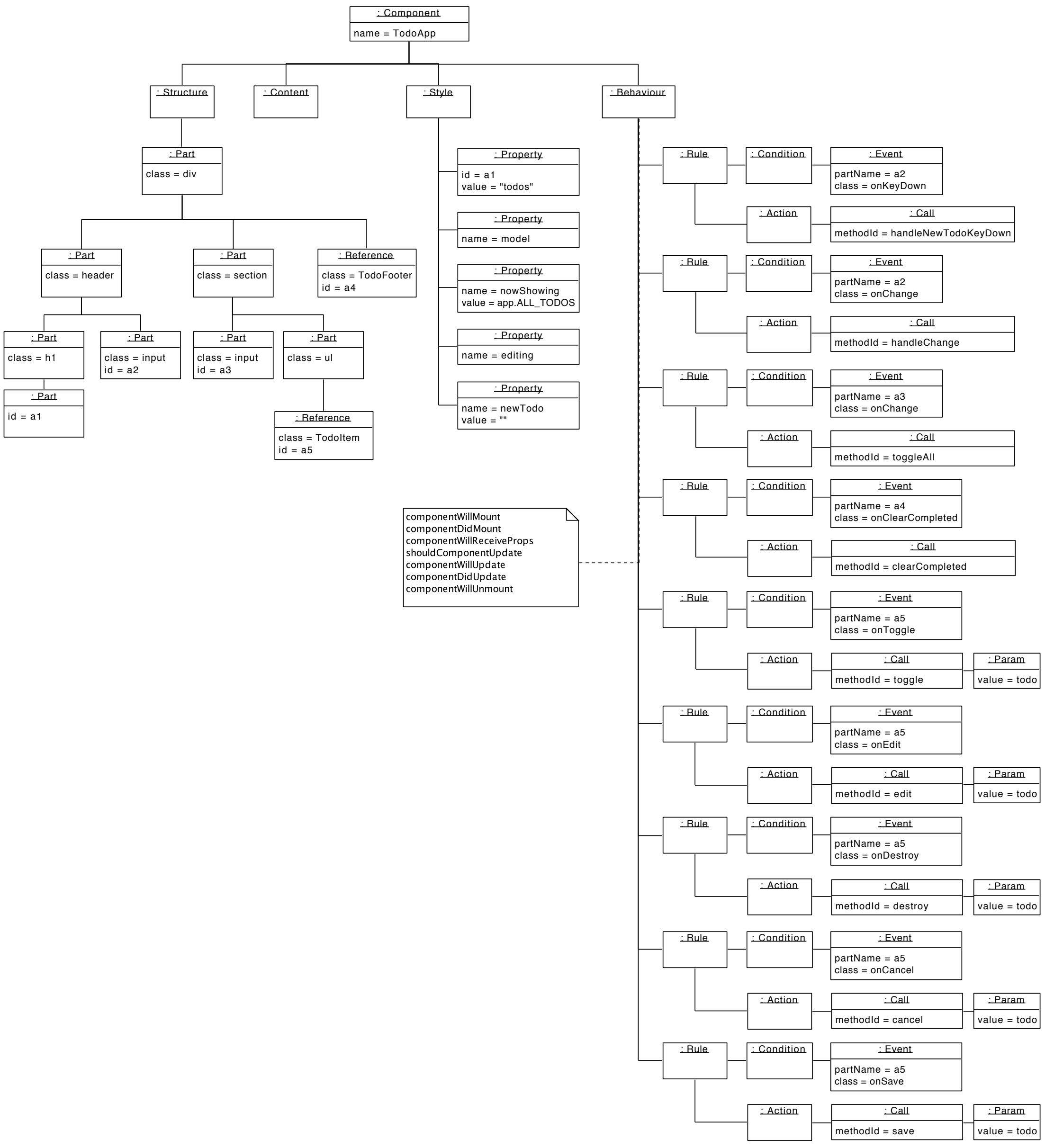


Figure A.1.: Hand crafted view model of the TodoApp component in the React implementation of TodoMVC

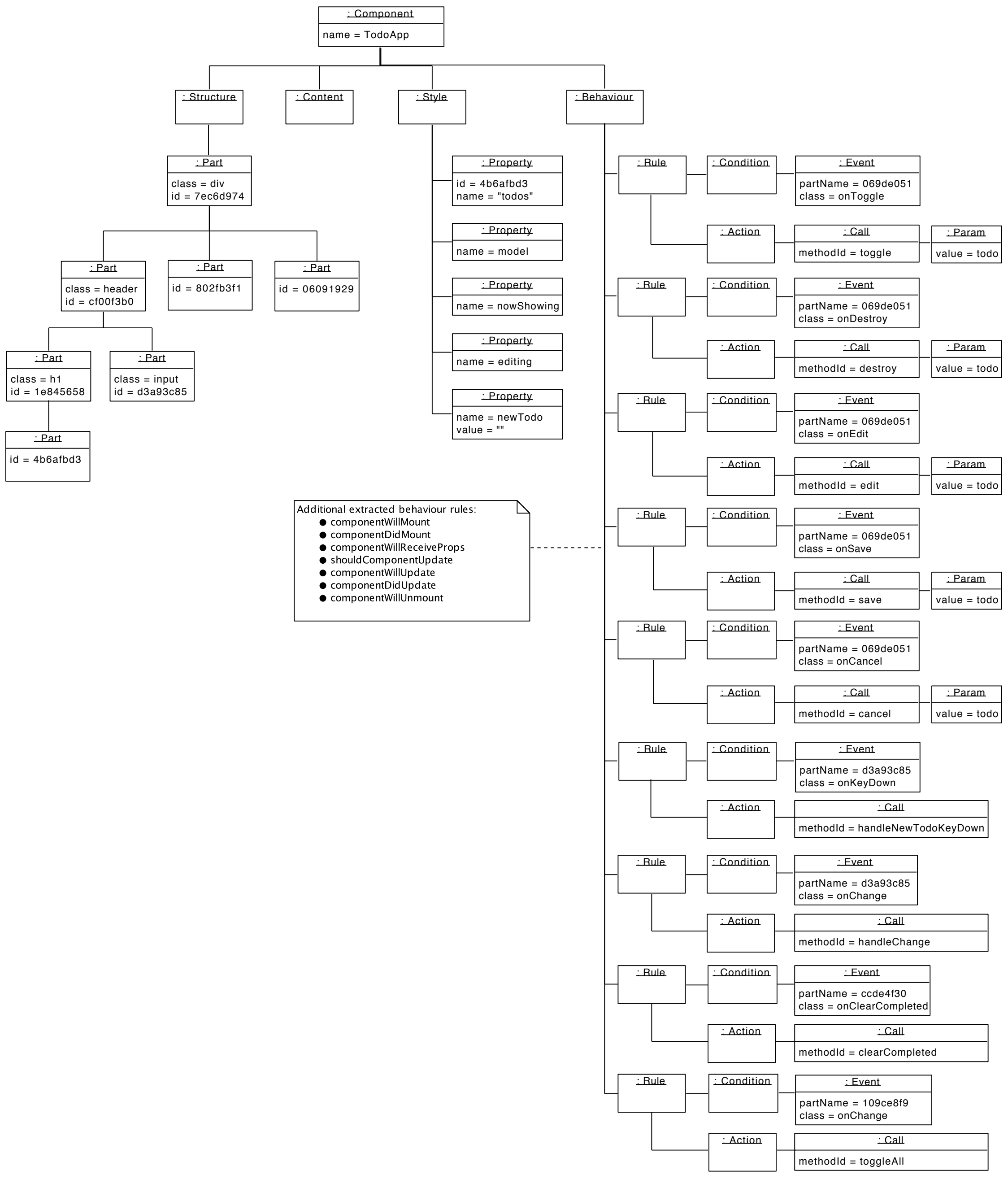


Figure A.2.: Extracted view model of the TodoApp component in the React implementation of TodoMVC

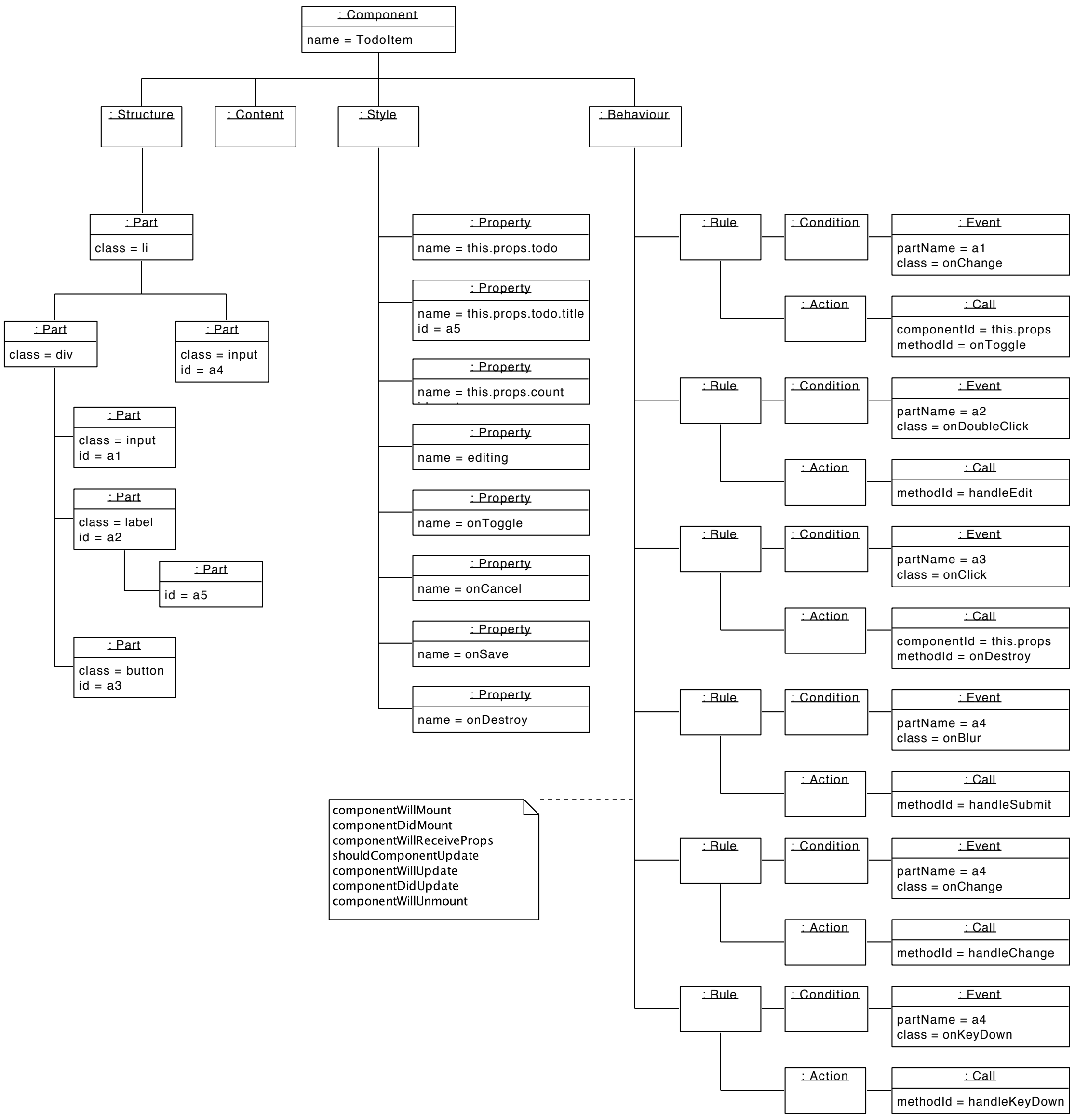


Figure A.3.: Hand crafted view model view model of the TodoItem component in the React implementation of TodoMVC

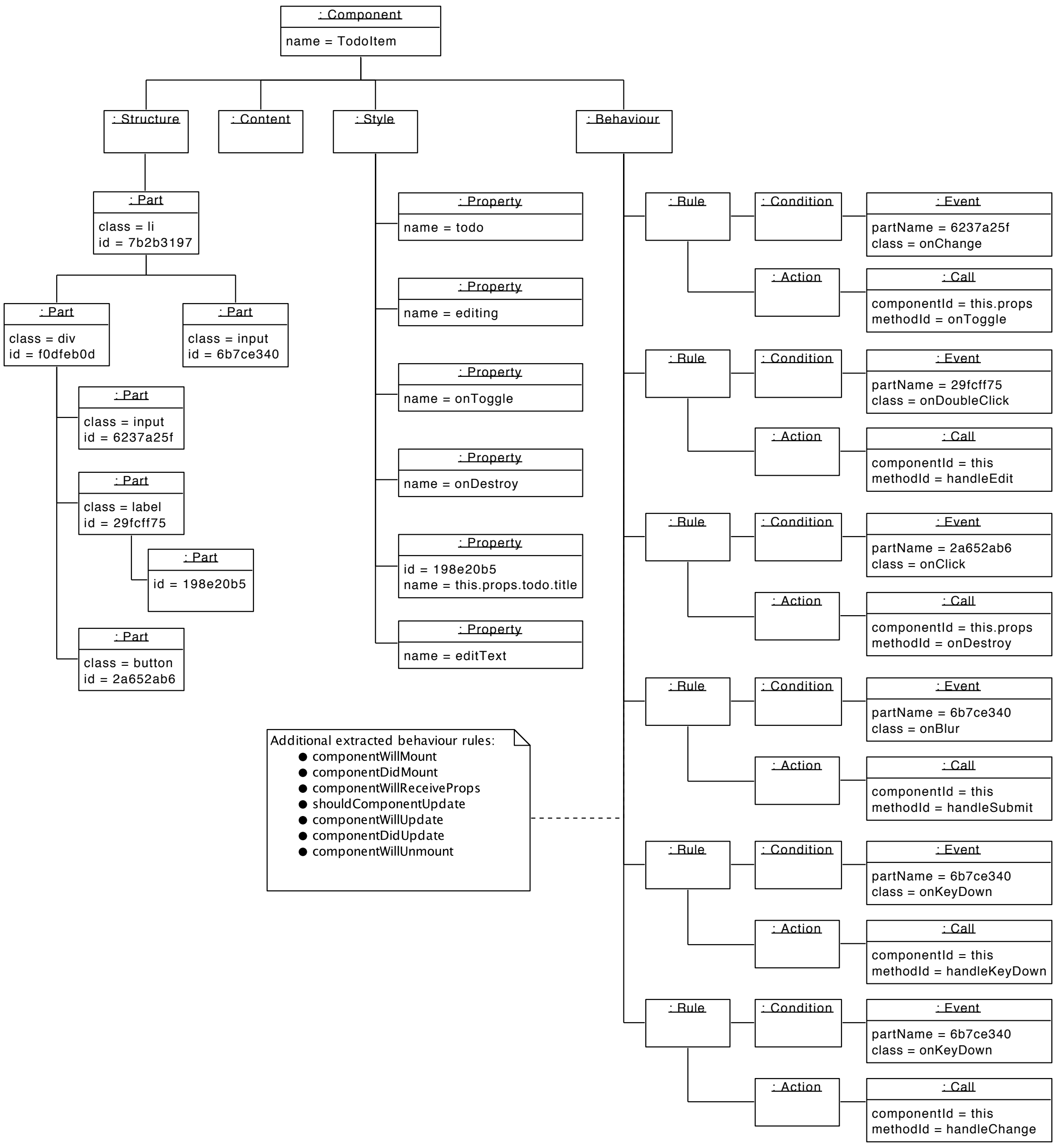


Figure A.4.: Extracted view model view model of the TodoItem component in the React implementation of TodoMVC

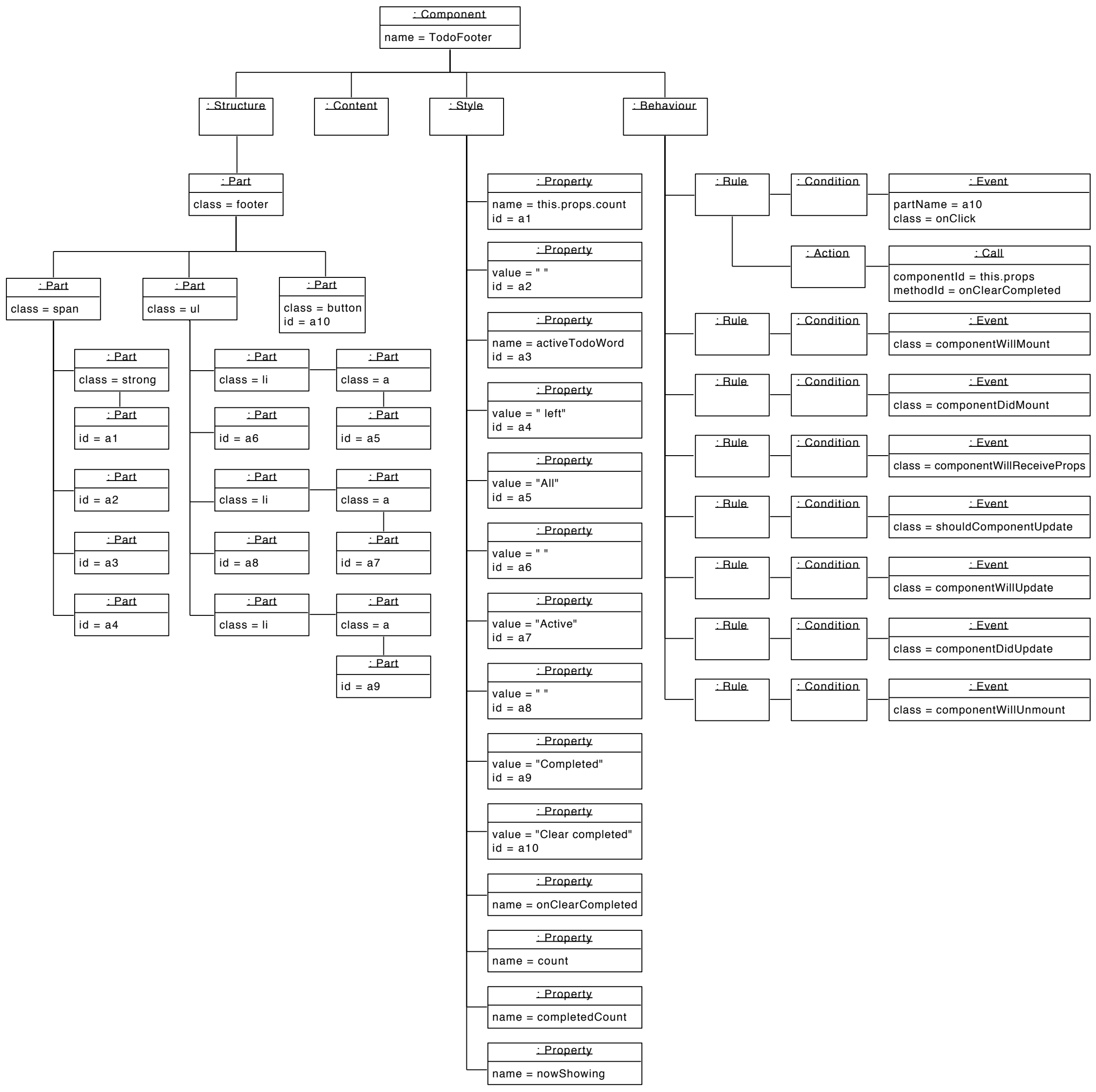


Figure A.5.: Hand crafted view model of the TodoFooter component in the React implementation of TodoMVC

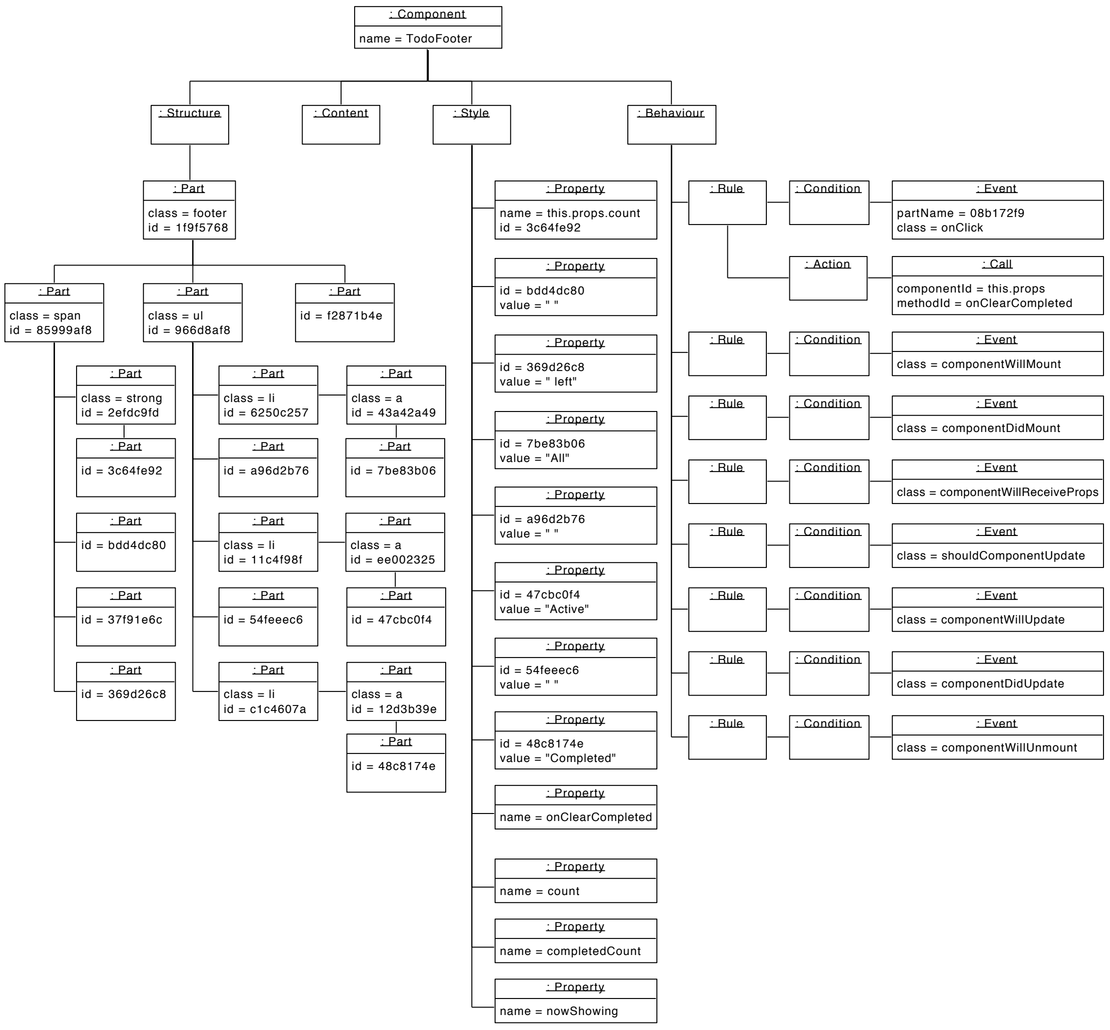


Figure A.6.: Extracted view model of the TodoFooter component in the React implementation of TodoMVC

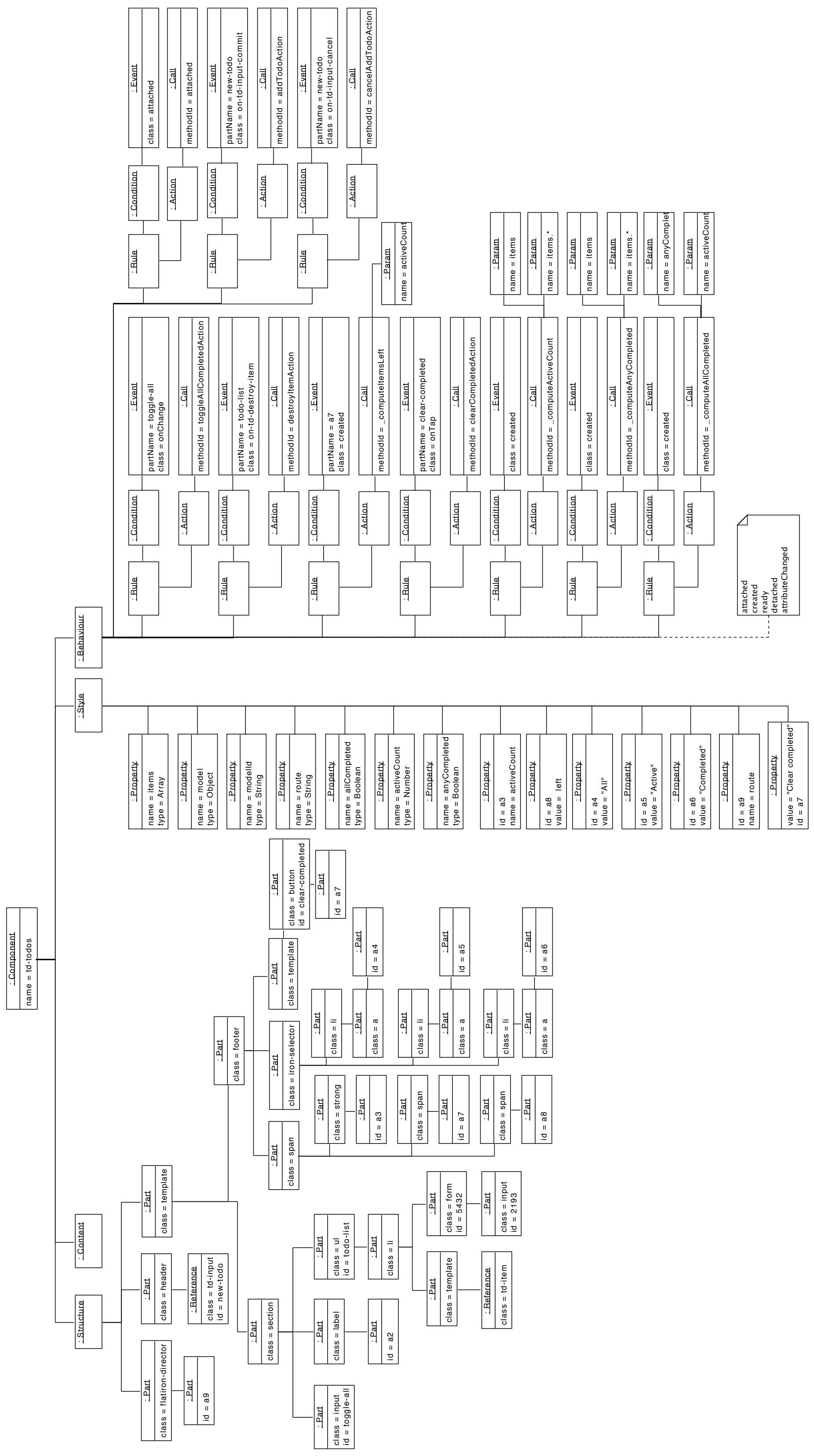


Figure A.7.: Hand crafted view model of the td-todos component in the Polymer implementation of TodoMVC

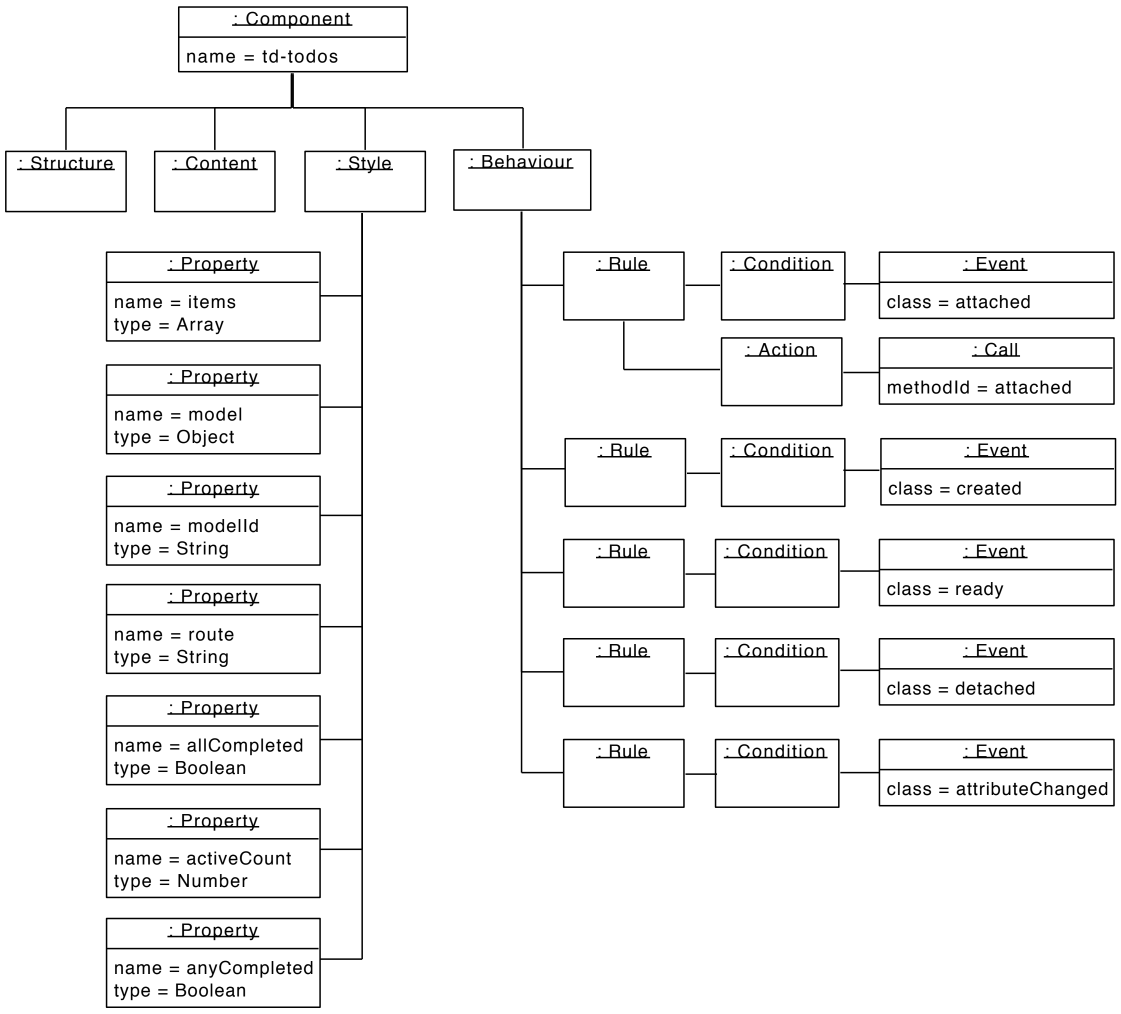


Figure A.8.: Extracted view model of the td-todos component in the Polymer implementation of TodoMVC

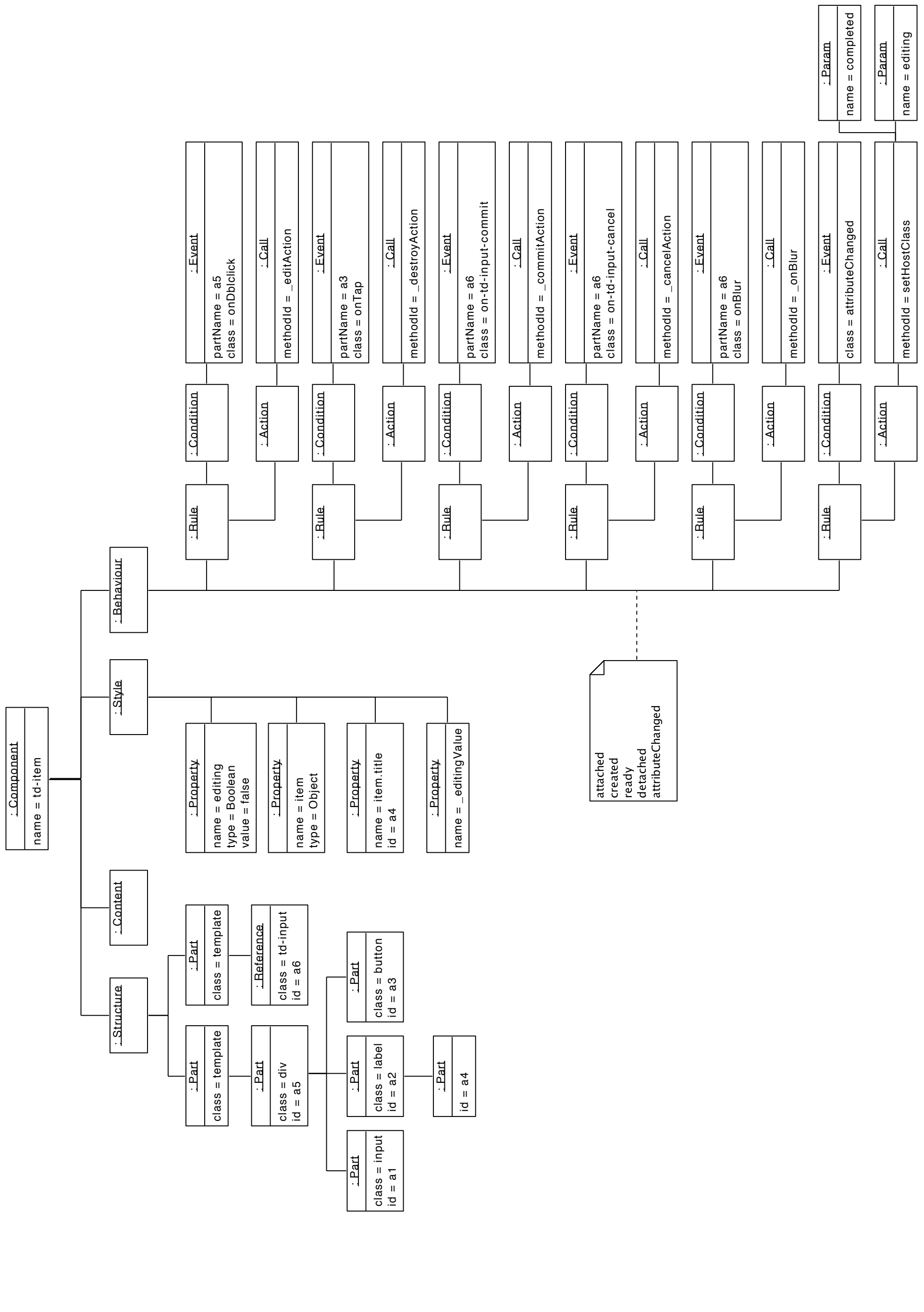


Figure A.9.: Hand crafted view model of the td-item component in the Polymer implementation of TodoMVC

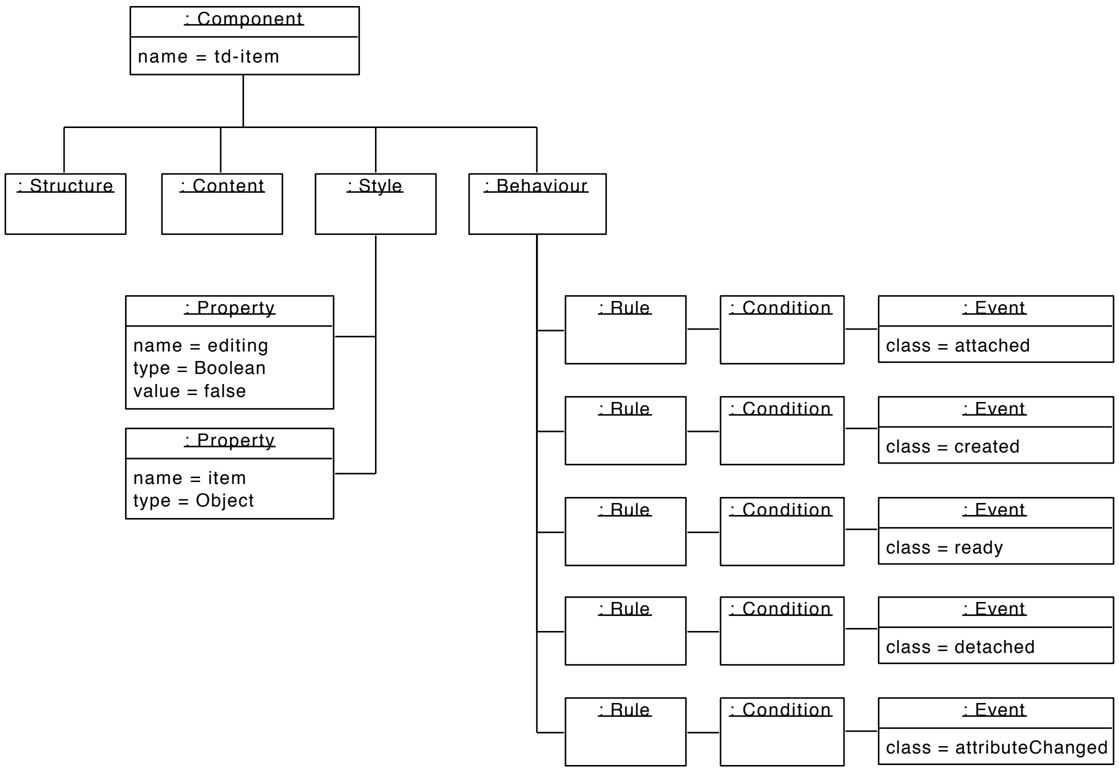


Figure A.10.: Extracted view model of the td-item component in the Polymer implementation of TodoMVC

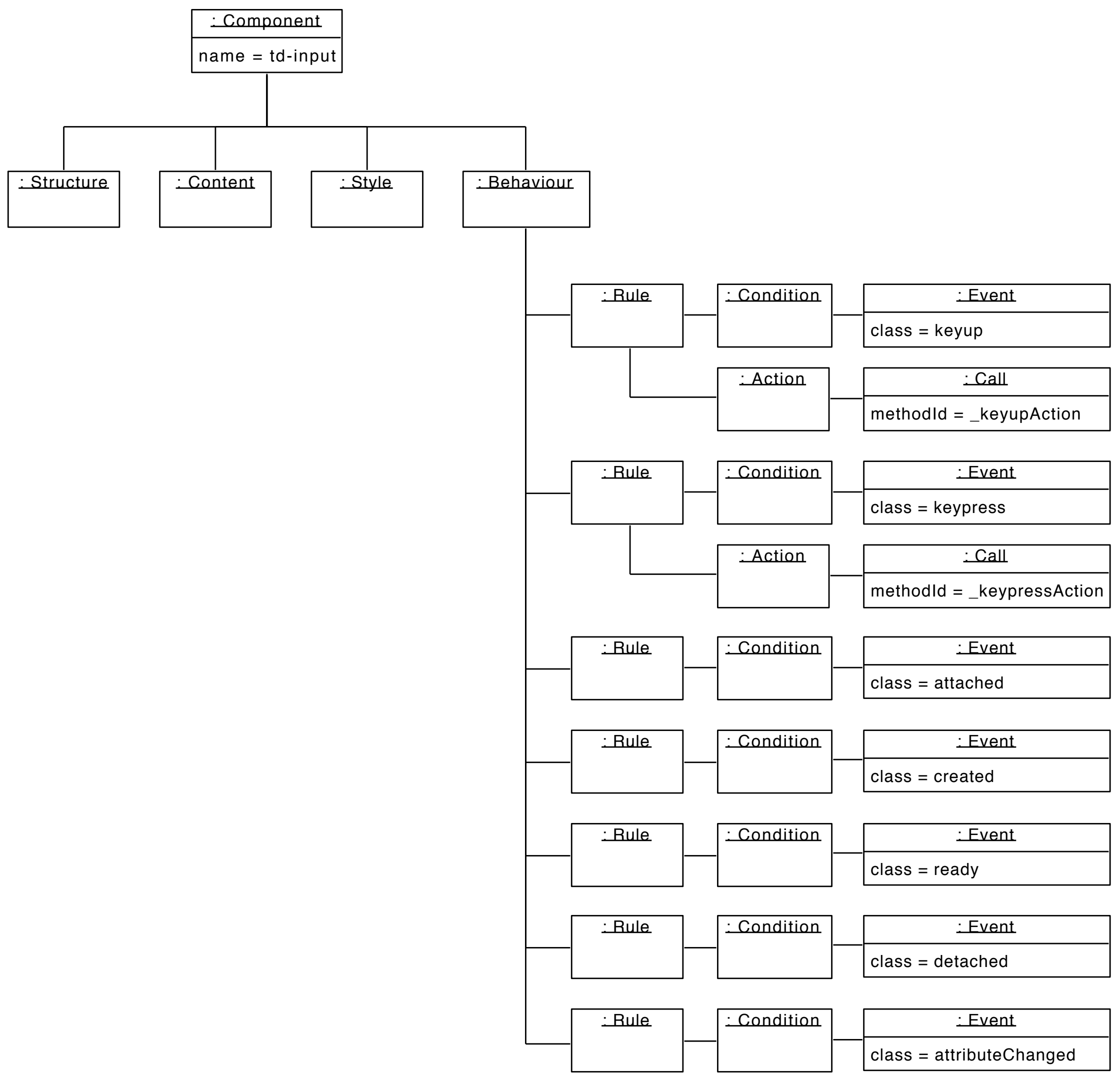


Figure A.11.: Hand crafted view model of the td-input component in the Polymer implementation of TodoMVC

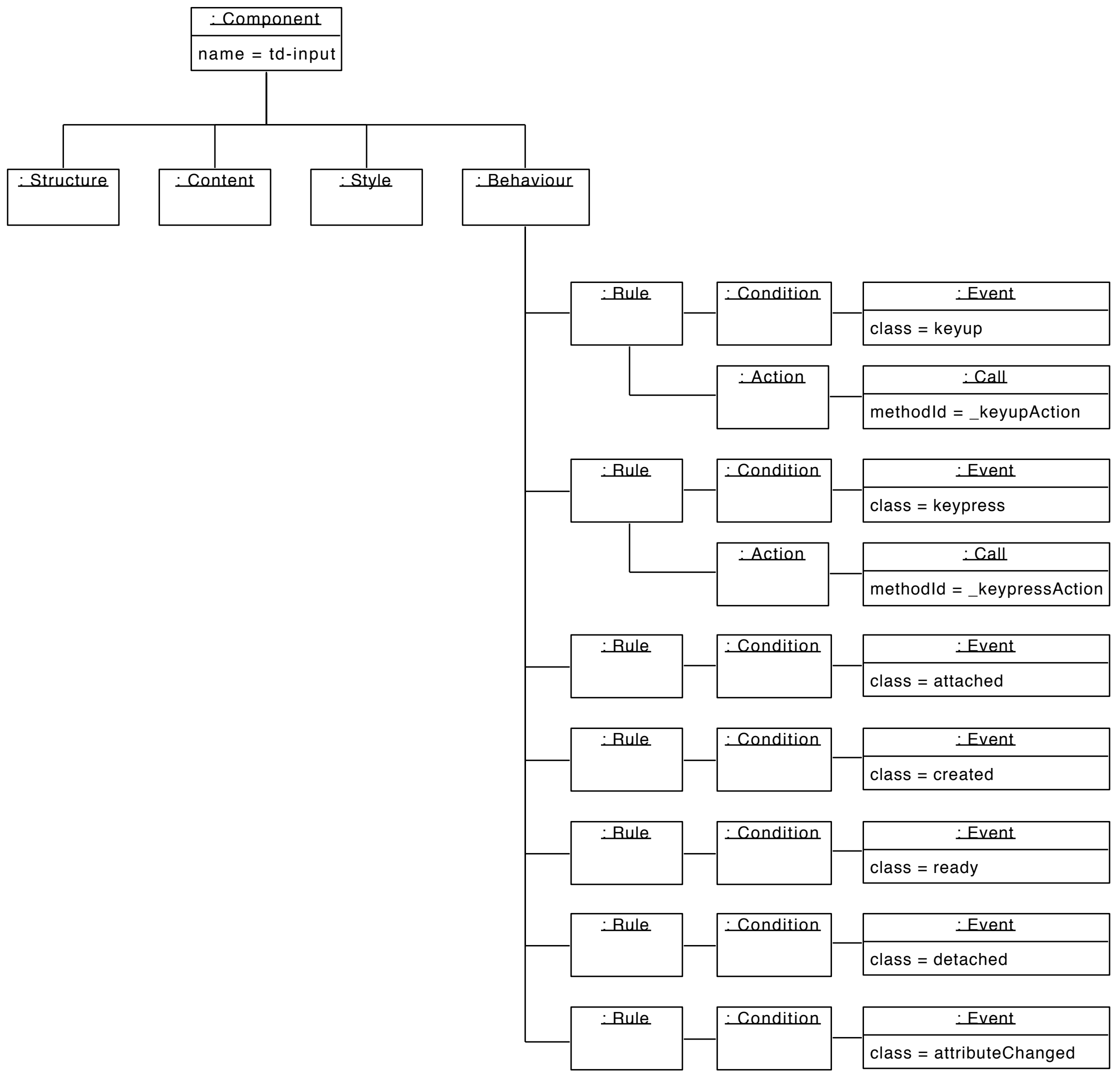


Figure A.12.: Extracted view model of the td-input component in the Polymer implementation of TodoMVC

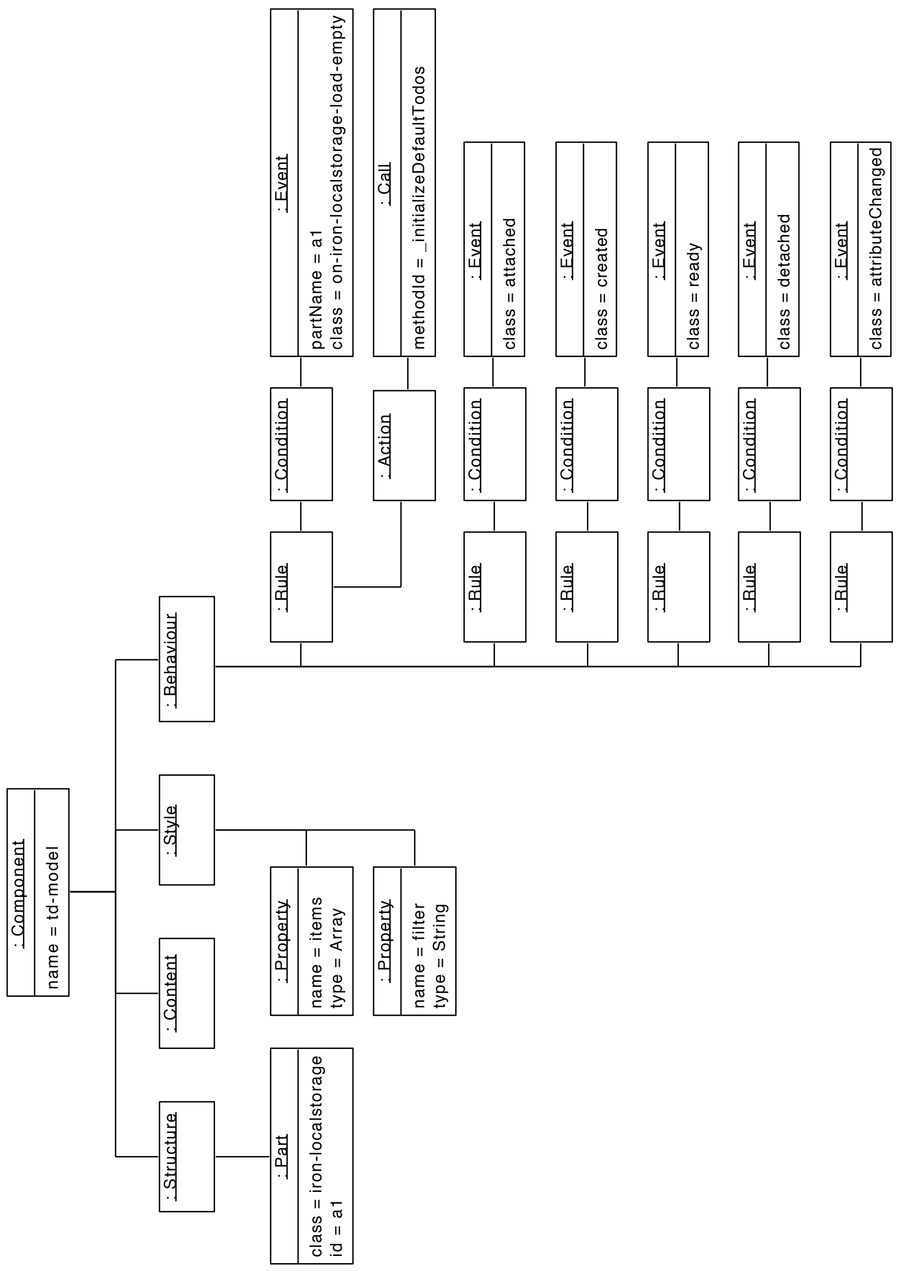


Figure A.13.: Hand crafted view model of the td-model component in the Polymer implementation of TodoMVC

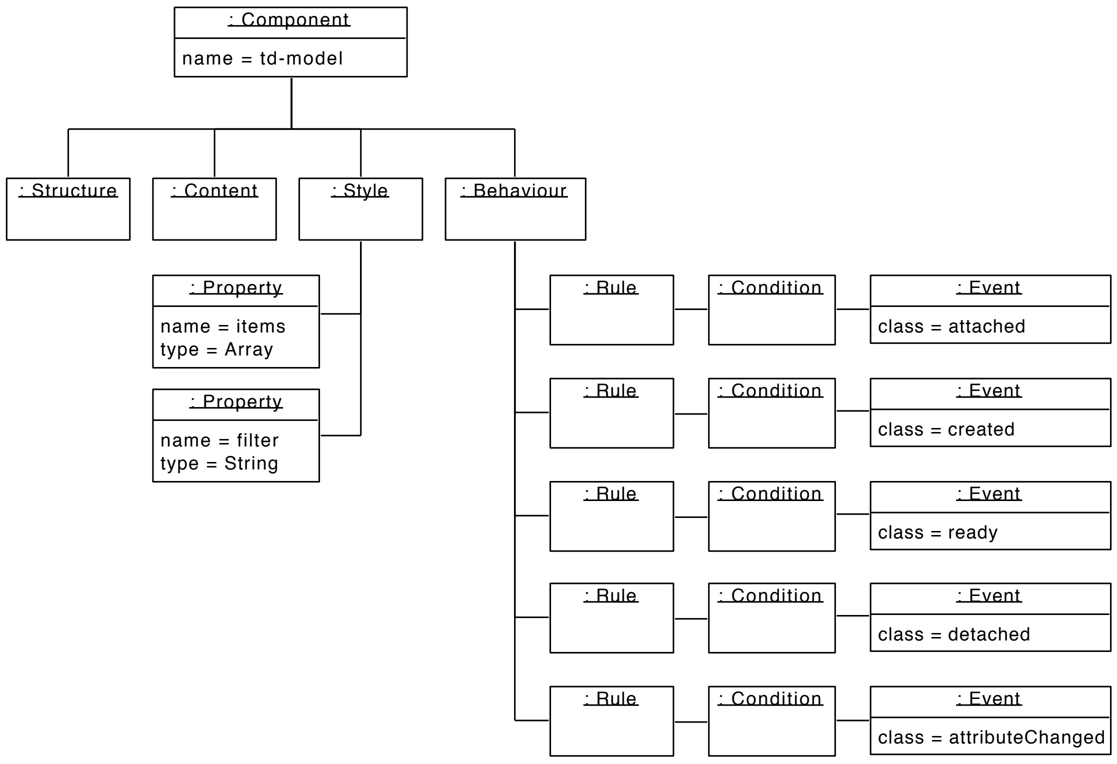


Figure A.14.: Extracted view model of the td-modle component in the Polymer implementation of TodoMVC

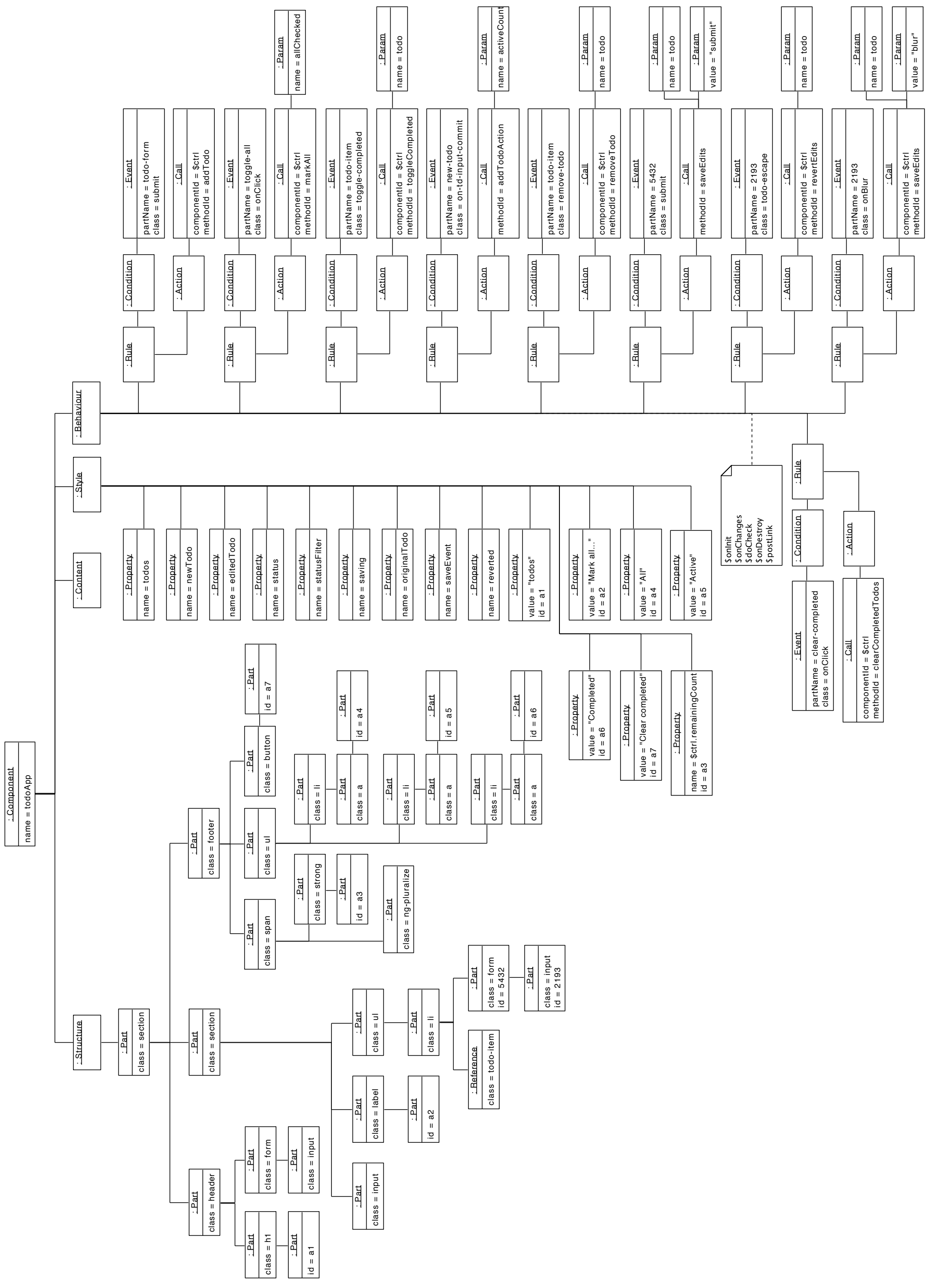


Figure A.15.: Hand crafted view model of the todoApp component in the Angular implementation of TodoMVC

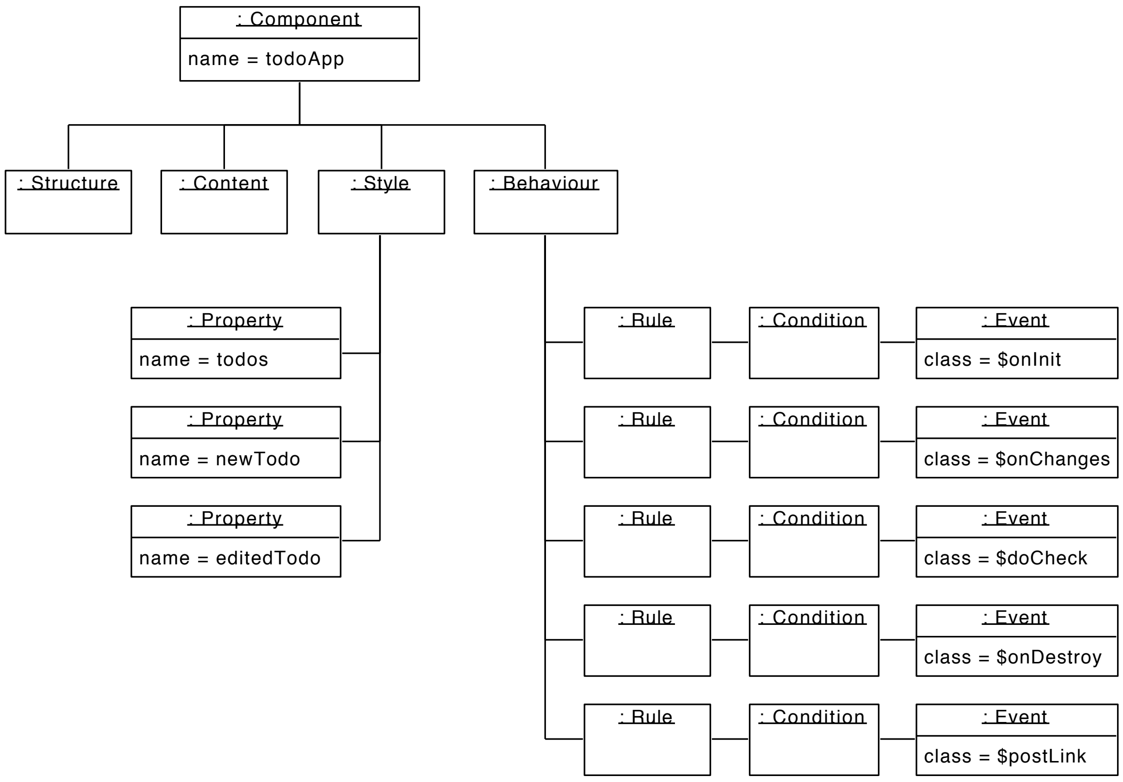


Figure A.16.: Extracted view model of the todoApp component in the Angular implementation of TodoMVC

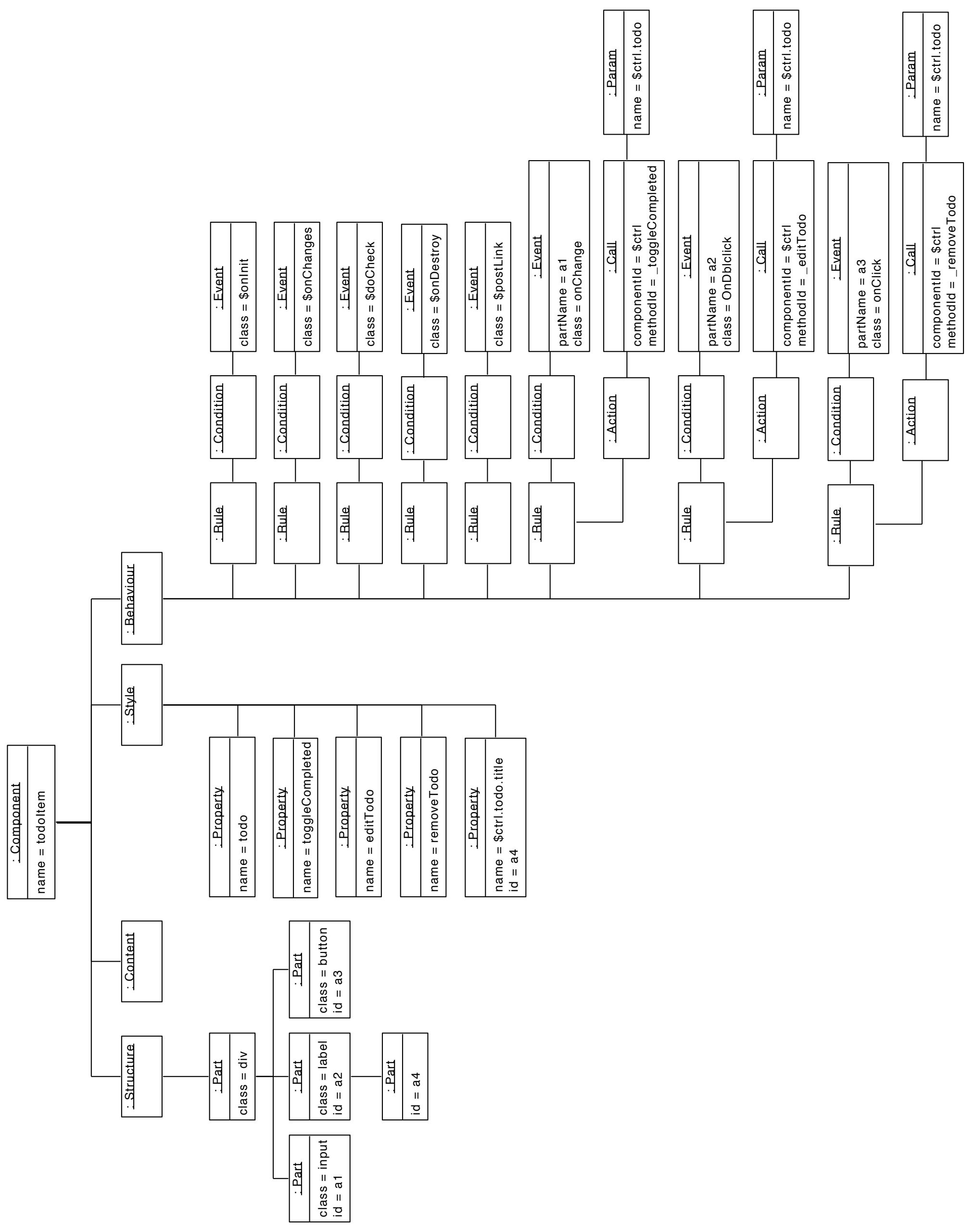


Figure A.17.: Hand crafted view model of the todoItem component in the Angular implementation of TodoMVC

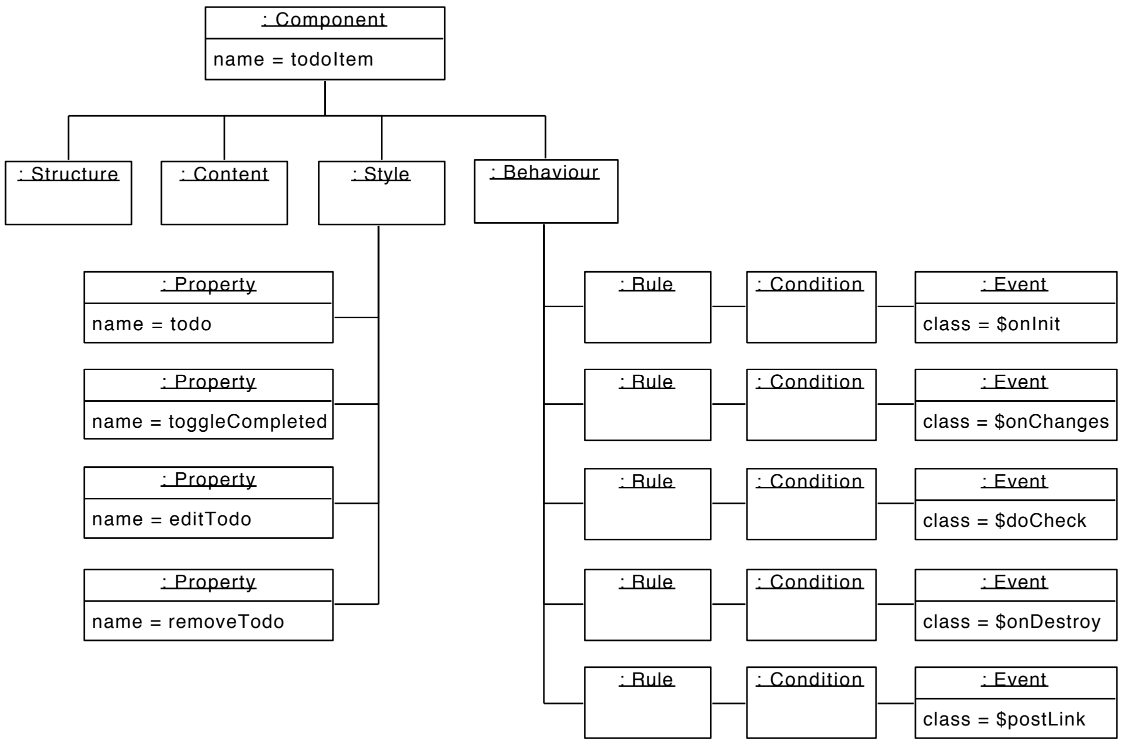


Figure A.18.: Extracted view model of the todoItem component in the Angular implementation of TodoMVC

List of Figures

List of Tables

Bibliography

[Aho+86] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers Principles Techniques and Tools*. Addison-Wesley, 1986.

[Ala+04] V. Alan, R. Hevner, March, S. T, Park, Jinsoo, Ram, and Sudha. "Design science in information systems research." In: *MIS quarterly* 28.1 (2004), pp. 75–105.

[CC90] E. J. Chikofsky and J. H. Cross. "Reverse engineering and design recovery: a taxonomy." In: *IEEE Software* 7.1 (Jan. 1990), pp. 13–17. ISSN: 0740-7459. DOI: `10.1109/52.43044`.

[Eic11] B. Eich. *New JavaScript Engine Module Owner*. `https://brendaneich.com/2011/06/new-javascript-engine-module-owner/`. [Online; accessed 25-October-2016]. 2011.

[Gam+94] E. Gamma, R. Johnson, R. Helm, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[Her15] D. Herman. *MDN Spidermonkey Parser API*. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API`. [Online; accessed 25-October-2016]. 2015.

[ISO16262] *Information technology – Programming languages, their environments and system software interfaces – ECMAScript language specification*. Standard. 2001.

[Jon03] J. Jones. "Abstract syntax tree implementation idioms." In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10.

[Koc16] I. Kochurkin. *Tree structures processing and unified AST*. `http://blog.ptsecurity.com/2016/07/tree-structures-processing-and-unified.html`. [Online; accessed 25-October-2016]. 2016.

[OAS08] OASIS. *User Interface Markup Language (UIML) Version 4.0*. Committee Draft. 2008.

[WM15] D. Wendel and P. Medlock-Walton. "Thinking in blocks: Implications of using abstract syntax trees as the underlying program model." In: *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. Oct. 2015, pp. 63–66. DOI: `10.1109/BLOCKS.2015.7369004`.