Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Utilising projective technologies for object-oriented development of WEB UI

Bc. Jakub Červenka

Supervisor: Ing. Robert Pergl, Ph.D.

5th May 2015

Acknowledgements

I thank my thesis supervisor for guidance, optimism and patience during my work. I also thank my family for their support during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 5th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Červenka, Jakub. *Utilising projective technologies for object-oriented development of WEB UI.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato práce se věnuje analýze, návrhu a implementaci webové aplikace pro vývoj objektově orientovaného webového uživatelského rozhraní. Ukazuje také potenciál využití projektivních technologií a generování kódu v této oblasti.

Klíčová slova uživatelské rozhraní, objektově orientované programování, JavaScript, React, webová aplikace, komponenty

Abstract

This thesis focuses on analysis, design and implementation of a web application for development of object-oriented web user interface. It also shows the potential of projective technologies and code generation in this area.

Keywords user interface, object-oriented programming, code generation, projective technologies, JavaScript, React, web application, components

Contents

Introduction **1**

1 State-of-the-art **3**
1.1 Server-side technologies 3
1.2 Client-side technologies 5
1.3 Summary 11

2 Codiscent's projective technologies **13**
2.1 Codiscent products 13
2.2 Benefits and drawbacks 14

3 Analysis **15**
3.1 Product statement 15
3.2 Use cases 16
3.3 Domain Model 17
3.4 Requirements specification 18

4 Design **19**
4.1 Ambidex 19
4.2 Data model 22
4.3 Server API 23
4.4 Data stores and actions 24
4.5 User interface 25

5 Implementation **29**
5.1 Server implementation 29
5.2 Reflux stores 30
5.3 Components 31
5.4 Deploying to Heroku 32
5.5 Using projective technology 32

5.6 Testing 35

6 Proof of concept **37**
6.1 Component design 37
6.2 Implementing in Instinct UI 38
6.3 Generating the components 38
6.4 Summary 39

7 Future work **41**
7.1 User interface 41
7.2 Better validation 41
7.3 Storage and concurrency 42
7.4 Component features 42
7.5 Ambidex 43
7.6 Direct integration 43

Conclusion **45**

Bibliography **47**

A Acronyms **51**

B Application screenshots **53**

C Proof of concept screenshots **55**

D Contents of enclosed CD **61**

E Installation guide **63**
E.1 Requirements 63
E.2 Installation 64

List of Figures

3.1 Use cases 16
3.2 Domain model 17

4.1 Flux stores and actions 25
4.2 Instinct UI wireframe 27
4.3 Components in Instinct UI 27

B.1 Screenshot of Instinct UI with default components 53
B.2 Screenshot of Instinct UI with proof of concept components 54

C.1 Wireframe of homepage / category listing screen 55
C.2 Wireframe of product detail screen 56
C.3 Wireframe of shopping cart screen 56
C.4 Generated homepage / category listing screen 57
C.5 Generated product detail screen 58
C.6 Generated shopping cart screen 59

List of Tables

4.1 List of available API endpoints 28

6.1 Lines-of-code comparison 39

Introduction

Today, the usage of web applications is on the rise. Users spend a high percentage of their time working with the web browser and desktop applications are shifting to the browser environment. E-mail clients, media players, even office suites are available as web applications. There is even an operating system, Google Chrome OS, which is basically a web browser. [1] Web applications can be used not only on desktop, but on tablets and smart phones, regardless of the operating system. Furthermore, the web app shares the same user data between different machines, supressing the need of explicit synchronization. Therefore, web design and development is growing more and more important.

Object-oriented programming (OOP) is a programming paradigm, which uses real world abstractions and concepts in design and development of software. The entities in the program, called objects, are representing real world entities and ideas. The objects are like black boxes, hiding the data and the implementation from the outside world and expose them only through a defined interface. The program consists of many objects, which are communicating to fulfill the goal of the program. The features and structure of OOP enhance the reusability and testability of the code. Therefore, object-oriented programming became very popular and almost all major programming languages support programming in an object oriented way.

However, the object-oriented programming is only rarely used for development of the user interface of web applications. The web applications are based on HTML and CSS, which are not "real" programming languages. On the client side, the document is manipulated by JavaScript, an object-oriented language, but often in a very imperative way. On the server side, the main used technologies (PHP, Java, .NET) are themselves object-oriented, but the generation of resulting HTML seldom is. Most languages and frameworks use a templating engine, which is just a way to insert the application data into the "plain" HTML.

Bret Victor is a designer, engineer and programmer who used to (among other work) design user interface at Apple. [2] He won the Apple Design

Award in 2007 for his Mac OS X trip planner widget, "The BART Widget". In his talks and essays, Victor prefers visual representation, simulation and interactivity to formulas and symbol manipulation. He presented corresponding demos in different science fields (for example math, data visualisation, programming, electrical engineering, animation or politics).

In his talk "Inventing on Principle" at Canadian University Software Engineering Conference in January 2012, he presented a JavaScript environment capable of showing the values of variables during the computation, manipulating the variables and showing the result in real time, even "showing the future" of an avatar performing a jump in a simple game. [3] The jump changed according to variable setting, which allowed to find the needed values very quickly. He presented his life guiding principle, importance of ideas and the need of an immediate connection between creators and their work. Without this connection, some ideas may never come to a fruition. He creates environments for ideas to grow, because to him, seeing ideas die "feels like a moral wrong, it feels like an injustice".

In another talk, "Drawing Dynamic Visualizations" at Stanford Human-Computer Interaction seminar in February 2013, he presented a tool for representing data in a visual way, which abstracted graph drawing to drawing objects according to input values. [4] For example, for drawing a bar chart, he drew a rectangle scaled to appropriate height. The canvas and the drawing process were higly interactive, immediately showing each step and the result. Each picture's input were the data and its output was the geometry of the image. This allowed to create data-driven, ad-hoc, but reusable and composable visualization very quickly.

The goals of this thesis are to present a non-traditional solution to web user interface development, based on facts and principles stated above. The first goal is to analyse, design and implement and a web application which would respect principles given in Bret Victor's talks. The application would be used to develop user interface components declaratively, with instant feedback to the programmer. Any changes the user makes would be immediately visible in the resulting component. The components should be object-oriented parts of web user interface, with defined structure, appearance and behavior. As they would be objects, they would be encapsulated and composable, increasing their reusability.

The second goal of the thesis is to explore possibilities of projective technology in user interface design. Data from this application would be therefore used with Codiscent Projector Template Generator (PTG) to generate working component code. The output of the application (input for the PTG) would be independent on the programming language and used libraries of the generated components. That would enable using the same data to generate the components in another language (when having appropriate generation template) if the underlying platform changes, for example when a new technology emerges.

Chapter 1

State-of-the-art

There are two main categories of web technologies — server-side and client-side. I will focus on the client-side technologies, as traditional server-side technologies, like PHP, Java or .NET are matured and stable and they are out of the scope of this thesis. On the other hand, client-side technologies are wildly evolving and their importance rises. Almost all client-side technologies are based on JavaScript, a programming language interpreted by the browser. While there are other technologies like Java applets and Flash, they are on the decline. [5]

The client/server distinction is not very strict, as most modern server-generated web sites have some JavaScript part (along HTML and CSS, which are obviously rendered by the client), ranging from input validation to "infinite" scrolling and content fetching to web sockets and server-side events. From the other side, client-side applications have to at least fetch "raw" (XML or JSON) data from the server and sometimes, the page is rendered on the server first, to allow agents without JavaScript (like search engines) to process the page.

1.1 Server-side technologies

The most popular languages for server-side web development are PHP, Java and ASP.NET. [6] I will describe a popular czech PHP framework, called Nette, and a less traditional Smalltalk web freamwork, called Seaside.

1.1.1 Nette

Nette is a PHP web framework. It is developed by David Grudl with contributions from Nette community. [7] Nette uses a MVP (Model-View-Presenter) architecture and templating engine Latte to render the web page. Nette supports using components as part of the application. These components are self-contained — they have their own template and data. Nette provides base

component classes to compose the component tree and map user actions to PHP methods (via so-called signals).

Nette has a sophisticated object model and utilizes dependency injection, having its own DI container (with the configuration written in NEON, Nette Object Notation, text format similar to YAML and superset of JSON). It makes up for some PHP shortcomings, unifying inconstencies in behavior and names and providing simple API for some PHP features (such as image manipulation, string manipulation or JSON parsing and generation).

Nette includes a debugger called Tracy which unifies error handling, displays clear error page ("bluescreen", which is actually red) with call stack in development and customizable "Internal server error" page (and supressing all error messages, improving security) while notyfing site administrator in production. The last part of Nette is Nette Tester, a lightweight testing tool.

Aside from templating and components, Nette includes classes for caching, database connection, iterating through files, generating and validating forms (validation rules are exported to JSON through HTML data attributes and included JavaScript library uses them to perform client-side validation), sending e-mails, user authentification and authorization, date and string manipulations or HTML tag manipulation.

Nette automatically protects from Cross-site scripting, Cross site request forgery (in forms), attacks on PHP session and more. For Nette, the security is the highest priority.

Nette supports client-side scripting via AJAX with so-called snippets — determined parts of page are rendered on server, then send back to the browser as strings in JSON response. JavaScript client script then overwrites target parts of the page in the browser, without refreshing the whole page. Another optional script can be user to push the page change to browser history.

Nette has been recently (in version 2.2) split to 20 packages, which are connected through Composer (dependency manager for PHP), so the developer can get the whole framework or just the needed parts easily.

Aside from the core framework, the community of Nette programmers produced a lot of addons, extending the functionality for specific use cases.

The last stable release of Nette is version 2.3.1, released on 27 March 2015. The 2.2.x branch is also supported, the 2.1.x branch will only get security updates until the end of 2015. All these versions require PHP 5.3.1 or higher. The last version to support PHP 5.2 was 2.0.18, released on 22 December 2014.

1.1.2 Seaside

Seaside is a web application server for Smalltalk. [8] Smalltalk is a pure object-oriented language, with many implementations (for example Pharo or Squeak). Smalltalk applications take form of virtual machine images. Application themselves therefore include environment for developing them — class

browser, workspace, output (called transcript) and debugger. Everything in the apllication can be modified at runtime and the changes are instantly visible.

In Seaside, the page and the application are divided into components, or widgets. Each component has a render method, which describes how the component should be displayed. Seaside provides classes for generating HTML programmaticaly. The programmer therefore does not write text templates filled with data, but rather calls method on an object, HTML canvas.

Application runs in so-called render loop and application state interacts with the browser — a method call may result in displaying a component and the method returns after an user action (for example, choosing from a list); likewise, clicking on a link calls appropriate component's render method. Invokation of Widget states and application screens can be easily composed to create complicated workflows. The state is carried over through requests by an URL parameter. The past application states are stored on the server, so the Back button and server history can be used to return to those states.

The last stable version of Seaside is version 3.1, released in May 2013.

1.2 Client-side technologies

1.2.1 JavaScript

JavaScript is an interpreted, object-oriented language with first-class functions. It is a dynamic-typed, prototype-based language which supports procedural, object-oriented or functional programming. Despite its name, it is not affilated with Java programming language, aside from having similar syntax features.

There are several runtime engines for JavaScript, the most popular being SpiderMonkey, used in Mozilla Firefox, and V8, used in Google Chrome and Opera browsers. Other implementations include Chakra (Internet Explorer), JavaScriptCore (Webkit) and Rhino (Java). [9]

The standard of JavaScript is called ECMAScript. The current version of ECMAScript is 5.1 (ES5), which is supported by all major browsers. Older browsers, like Internet Explorer 8 and older, support at least ECMAScript 3 (ES3) and the 5.1 functionality has to be added to the via so-called polyfills or shims, scripts which simply define the ES5 functions in ES3 terms. This is possible because objects in JavaScript are very extensible.

The new, 6th edition of ECMAScript, called Harmony, is being codified and it includes new syntactic features like class definitions (which are just syntax sugar, the inheritance model is still prototype-based) or shorter lambda syntax. ECMAScript 6 (ES6) support varies browser to browser (or runtime to runtime). It's possible to write ES6 code and then transpile it (compile between programming languages) to ES5 or use polyfills for ES6 functions.

(However, the new syntax features cannot be polyfilled and have to be transpiled.)

The main area of usage of JavaScript are dynamic web applications. As the usage of these applications rises, so does their complexity and demand for their speed. Therefore, the JavaScript engines in browser came up with many optimizations to fulfill those demands. These optimized engines give JavaScript an advantage even in non-browser environments.

1.2.1.1 Node.js

Node.js is a JavaScript platform for server environment. It is based on Google's V8 engine used in Google Chrome browser. It is developed by a community of developers and governed by software company Joyent. It utilizes a single-threaded event model with callbacks to handle I/O asynchronously, in a non-blocking way. This allows for a great number of connections running in parallel. Node.js provides an API for all sorts of operations, including HTTP or file system access. This API is abstracted from the operating system of the server. Node's current stable version is 0.12.2, released on 31 March 2015. [10]

The V8 engine, the runtime engine Node.js is built on, is writen in C++. It efficiently implements property access by creating hidden classes. It generates machine code directly from JavaScript and uses inline caches, which can be patched by the runtime. It employs a generational, stop-the-world, garbage collector to ensure fast object allocation, short garbage collection pauses, and no memory fragmentation. All these features make the V8, and therefore Node.js, extremely fast. [11]

1.2.1.2 npm

npm is the package manager, originally created for Node.js, but now used in various environments. Its intent is to make using shared code and sharing own code easier. It also enables publishing and consuming updates more easily. [12]

npm manages code in so-called packages. Each package consists od several JavaScript files and a file `package.json`, where the information about the package is stated, including the dependencies of the package.

npm consists of three things — the command-line tool, which installs packages' dependencies, publishes packages, etc. (the tool is also shipped with Node.js); the registry, where the packages are stored; and the website, where the information about the packages is presented.

1.2.2 jQuery

jQuery is a JavaScript library by the jQuery Foundation. It has concise API and handles the differences in implementation between browsers. The main

features of jQuery are: traverisng and manipulating the document using CSS3-like selectors, delegated and undelegated event handling or sending AJAX (Asynchronous JavaScript and XML) request.

These features are used mostly to supplement server-generated page with browser functionality. The elements of the page are often accessed directly in an imperative way.

1.2.3 AngularJS

AngularJS is a client-side JavaScript framework developed by Google for creating "reactive" web applications. The code of the application is divided between HTML, which defines the view part of the application, and JavaScript, which defines the application logic in form of controllers. The JavaScript is divided into modules to prevent interference with global namespace.

Angular defines, parses and evaluates special HTML attributes (called directives), which connect the HTML and JavaScript. Angular also replaces JavaScript expressions embedded in HTML in double braces with their value.

Angular uses so called two-way binding, where the changes in JavaScript values are reflected to HTML, whereas user input, like form values are reflected in JavaScript variables. AngularJS detects changes by comparing the current values with values stored earlier in a process of *dirty-checking*, unlike other libraries. The two-way binding circumvents the need of changing the DOM directly (for example, with jQuery).

Angular can be used to create components, which combine the HTML template and the controller. Components form custom directives, which further extend the HTML syntax. Components also have so-called scope, which is different from JavaScript variable scope and which is used as the execution context for expressions in the template. [13]

The last stable release of AngularJS is currently version 1.3.15, released on 17 March 2015. Google announced working on Angular 2, a complete rewrite of Angular, with respect to modern web technologies, which is currently in Development Preview phase. [14]

1.2.4 React

React is an open source JavaScript library for developing user interfaces. It is developed and maintained (mostly) by Facebook. [15]

A React application consists of components — JavaScript objects. These objects form a tree, which is mounted to the browser document.

Data of React components consist of *properties*, called *props*, and *state*. Props are the data passed to the component by its owner, whereas state are the data managed by the component itself. When the state or props of a component is changed, it is re-rendered (along with its children), if neccessary. Data in React flows in one direction — props are passed down the component

tree. Child components then use callbacks (given to them as props) to notify parents about user input. When the callback is invoked, the parent changes its state accordingly, re-rendering in the process.

React utilises so-called Virtual DOM to improve its performance — instead of directly applying every change to the document in browser, it creates in-memory representation of its components, then computes the diffference between the components and the document and then does only the neccessary changes. This improves user experience, too, because the DOM elements are changed instead of removed and re-added.

1.2.4.1 JSX

React includes an enhanced JavaScript syntax, called JSX, which allows to use a XML-like syntax to create React elements. [16] For example, given this JavaScript code:

```
var Component = React.createClass({...});
var variable = "variable";

return React.createElement(
    Component,
    {
        property: "value",
        anotherProperty: variable
    },
    React.createElement("div", null, "Content")
);
```

It can be written in JSX as:

```
var Component = React.createClass({...});
var variable = "variable";

return (
    <Component property="value" anotherProperty={ variable }>
        <div>Content</div>
    </Component>
);
```

JSX also enables some ES6 features, like object desctructuring and spread operator. For example, this takes flag and style from the props passed to the component, chooses class name according to the flag, adds { color: red } to the style and passes the class name, the style and all other props to the child element (div):

```
var Component = React.createClass({
    render: function () {
        var { flag, style, ...other } = this.props;
        return (
            <div className={ flag ? "enabled" : disabled }
                     style={ ...style, color: red } ...other>
                Content
            </div>
        );
    }
});
```

JSX script can be loaded to the browser using provisioned JSXTransformer (for development) or transpiled to JavaScript (for production).

1.2.4.2 Flux

Flux is a JavaScript application architecture developed and used by Facebook. It complements React views and makes for simpler application architecture then convential MVC. [17]

In Flux, instead of calling event handlers criss-cross, changes are propagated in a simple loop. The state of a Flux application consists of so-called stores, which provide access to data. React components get data from the stores. When the user input is handled, so-called actions are created and directed through event dispatcher to stores registered to the dispatcher. Stores modify their appropriate state and notify the components, which redraw if needed.

The dispatcher is crucial for Flux, because it forbids dispatching actions if another action is being dispatched, therefore preventing actions and changes going out of control. It also allows the store to wait for another store(s) to complete their handling of the action before handling the action themselves.

Flux is not a framework or library, it is mostly a pattern, which can be implemented in many ways. Facebook published their implementation of a dispatcher for public usage. There are also many other implementations of Flux,

1.2.4.3 Isomorphic web applications

React can be used to create so-called *isomporphic web applications*. Isomorphic applications share code (not neccessarily all of it) between the server and the client, with only small differences. [18]

To do this, the application needs to detect the environment and render the view appropriately (including the HTML header on the server). Application state also needs to be serialized by the server and then deserialized by the client.

In case of React, the root component is rendered on the server with `React.renderToString` or `React.renderToStaticMarkup` and sent to the client via HTTP. On the client, the component is then mounted to document via `React.render`.

Aside from the inherent benefit of having to write most of the code only once, this approach combines benefits of both server-side and client-side rendering. Because of the initial server-side render, the page load is faster (as the client gets complete HTML in one request instead of loading the blank HTML, API data and JavaScript separately, then rendering the page via JavaScript) and the page is accessible to clients without JavaScript (for example, search engine robots). Because the subsequent rendering happens in the browser, the page does react much faster. Also, the application does not need to transfer the markup again, saving bandwidth in the process. And finally, the server load is reduced, as the server does not have to render the HTML again. [19]

1.2.5 Wakanda

Wakanda is an open-source JavaScript platform that provides a full development stack for creating web applications. It is developed by a french company called 4D. It consists of Wakanda Studio, Wakanda DB (a server), Wakanda Framework (for client-side JavaScript) and Wakanda Web Tools. [20]

Wakanda Studio is an IDE for working with Wakanda Server and creating the application as a whole. The data model, the view, client- and server-side JavaScript are specified, run and debugged in the Studio. The model entities (called Data Classes) and their attributes, methods and relations can be defined visually in Data Model Designer or described in JavaScript. The user interface of the application is created using a WYSIWYG designer. The view is also connected to the model.

Wakanda DB is a server part of Wakanda. It consists of a JavaScript engine, a NoSQL database and an HTTP server. It uses SquirrelFish Extreme, the JavaScript engine of Safari, with added API for server purposes. The NoSQL database, called DataStore, has comprehensive REST interface. The database schema, server-side processing, and querying are all done in JavaScript. The application can be deployed to the server automatically.

Wakanda Framework is used to manipulate the view and accessing the server data in the browser. Wakanda Dev Tools consist of application Administration Panel and Chrome Debugger, which can be used to inspect the server-side code in Google Chrome browser.

1.2.6 Amber Smalltalk

Amber Smalltalk is a client counterpart to Seaside. Amber is a Smalltalk implementation that runs in browser as JavaScript. Amber code is compiled to JavaScript and can interact with JavaScript environment. The whole frame-

work utilizes JavaScript ecosystem, utilities and libraries, like Bower, Grunt or jQuery. It consists of a browser IDE, which looks and works like other Smalltalk implementations — class browser, workspace, trancript, etc. Like Seaside, it uses HTML canvas to interface with DOM elements and render HTML in browser. It also includes a base class for components – Widget. [21]

There is also a project called Tide Framework, which is a layer managing communication between Amber in the browser with Pharo (Smalltalk implementation) on the server, allowing to send data from Pharo and render them by Amber and sending request to Pharo server from Amber. It is however (as of April 2015) still in an experimetal phase and no longer developed (the last change on GitHub was from August 2014). [22]

1.3 Summary

Client-side JavaScript provides a large variety of frameworks and libraries, from low-level library like jQuery to full stack solution like Wakanda. The JavaScript ecosystem is developing swiftly and new ideas are put into practice soon.

Because this thesis focuses on the user interface, I will use the library specialized for user interface — React. React is also well designed with respect to object oriented programming, as its components are well encapsulated and composable. Complementing tools and patterns of React, JSX and Flux, are also a reason to use React.

Chapter 2

Codiscent's projective technologies

Codiscent Ltd. is a consulting and software development company, which specialises in software for code generation. They claim code generation (with their software) allows to escape the "triple constraint" of software development. The triple constraint states that the software development process and the resulting product can have only two of Economy, Quality or Speed. The code generation promises to help develop the software "Cheaper, Better and Faster".[23]

Codiscent's bussiness is commercial software development and consulting services. They offer various levels of cooperation and tool and methods usage according to the situation and estimated frequency of updates to the generated system.

Their methodology is called Agile Model Driven Development. It uses Codiscent's projective technology along with the specification data to generate the desired solution.

2.1 Codiscent products

The main Codiscent products (for the scope of this thesis) are Projector Template Generator and Generative Engineering Studio. Other Codiscent products include Solution Modeling and Integrity Support, Reverse Engineering Studio and Control Center Generator. [24]

The **Projector Template Generator** (PTG) is the core of Codiscent's projective technology. It is a text generator, which combines specification data, metadata describing a particular problem or set of requirements and simple and concise templates to generate output. The format of the output is specified by the template and it can take almost any form.

As the templates resemble the target code with only several more syntax

elements to control the code generation, the template can be created to generate anything which can be written as text. The specifications can also take many forms, including relational data, Excel spreadsheets, XML and UML or ER diagrams. The flexibility to deal with various source types and output formats is the most powerful feature of the PTG.

The output of the PTG is independent of both the PTG and the rest of the CodiScent platform. It can be modified and further developed at will, but there are significant benefits in continuing to use the platform futher.

The **Generative Engineering Studio** (GES) is an integrated developed environment (IDE) supporting template creation and debugging. It uses the PTG to generate code from templates and specification. It manages all artifacts needed for the code generation — models, specification data sources, templates and extensions. It groups related artifacts into a *solution*. It also controls the generation process.

The **Reverse Engineering Studio** (RES) supports similar workflow, but instead of generating code, it parses code to produce the specification data. The data can be then used with GES to forward-generate another solution.

2.2 Benefits and drawbacks

The main benefits of projective technologies are related to code — programmers have to write less code (just the template in ideal case), the generated code is more uniform and therefore more clear than the hand-written code. The templates can be also reused with another set of specification data almost without effort. [25]

Other benefits include reduced maintenance effort, as the solution can be regenerated easily when the specification changes, and easier migration and transformation, as the templates can be easily changed to reflect changed needs of the solution.

The main drawback of projective technologies is that they are not suitable for every problem. It's also a little difficult to extend the generated code, although it is possible with right means.

Chapter 3

Analysis

3.1 Product statement

As stated in the introduction, the goal of this thesis is to create a web application for creation of web user interface components. As the application will be developed using JavaScript library React, I decided to name it **Instinct UI**.

Instinct UI will be an application for creating web components declaratively. User will define a structure and properties of the component and the application will render it. When the user changes the definition, Instinct UI will re-render the component with the changes immediately. For example, when the user changes the size, color or content of the component, the changes would be instantly visible. This ensures the immediate connection between the user, for example a web designer, and his creation, allowing the user to explore his ideas and understand how the component behaves depending on its input.

After creating the component with Instinct UI, the user would then download the specification data and run them through the generation template with Codiscent Projector Template Generator. The PTG will generate a working component code ready to be used and deployed in user's web UI projects.

As components are objects (in a OOP way), they need to be composable and pass data from one component to another. As they are part of the user interface, they should react on user input. React components accomplish this by passing the data to children components as props, while listening for user input and changing their state accordingly. In the scope of this thesis, I focused on the structure of the components and passing the data from the parent to the children. Handling user interaction (and passing the data from the children to the parents) can be integrated in future work.

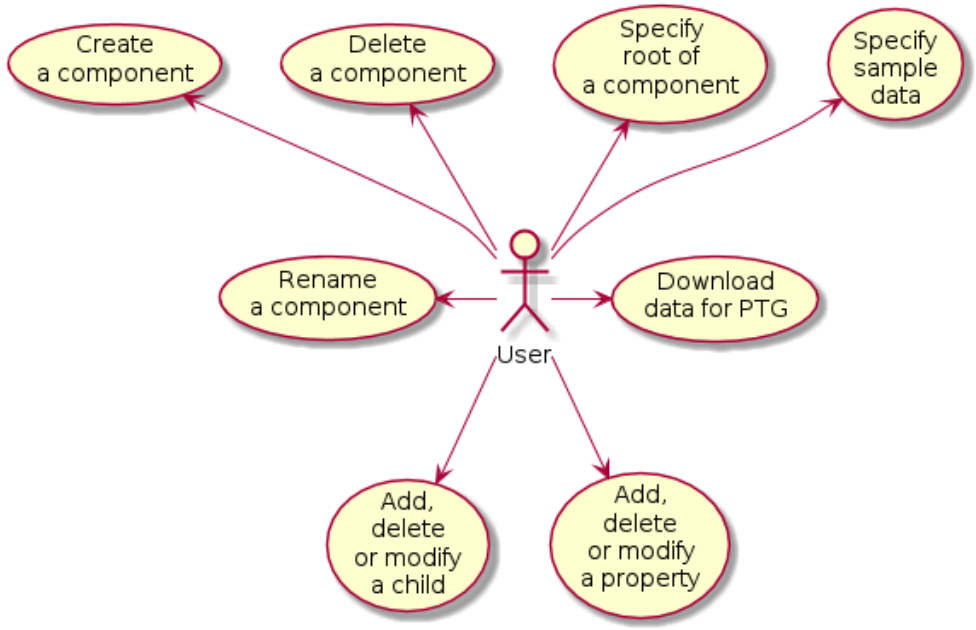

Figure 3.1: Use cases

3.2 Use cases

This application consists of several use cases related to component definition. They are shown in figure 3.1.

The user will be able to *create a new component*. This would add a new empty component to the system. The component would be available for using in another component. The user will be able to define how the component will be rendered, namely *specify its root element*. Component's root element could be a HTML tag or another component. Because components render according to their input, it will be able to *specify sample data* for the component.

Because the components can have a complicated structure, the user has to have a way of defining it. The user will be therefore able to *add, delete or modify a child* of a component or its another child. The user will also be able to *add, delete or modify a property* of a component or its child.

The user will be able to *rename an existing component*. The component's name cannot be empty or the same as the name of another component in the system. The user will also be able to *delete a component* if he or she wishes to. The application should ask the user for confirmation of the deletion.

As the application is not the final destination of components, it also has to prove a way of *downloading the component data for use in the PTG*.

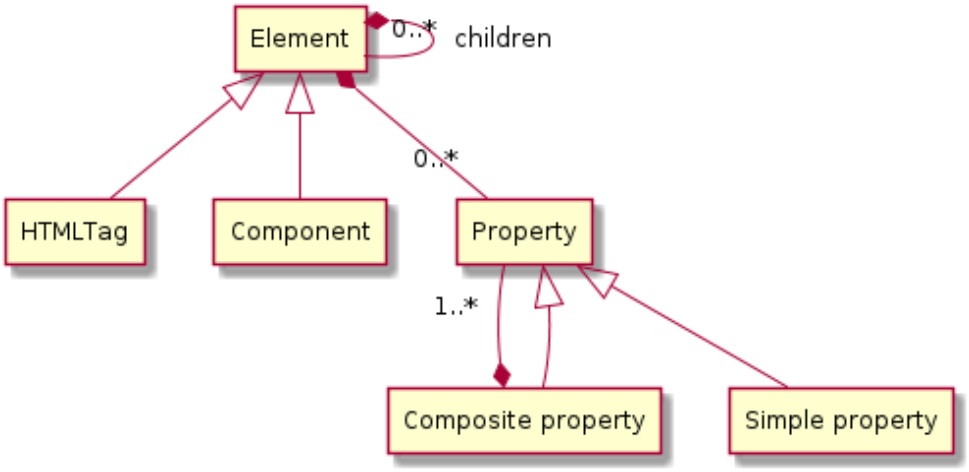

Figure 3.2: Domain model

3.3 Domain Model

The domain model of this application consists of elements, components, HTML tags and properties. Properties can be simple or composite. The domain model is shown in figure 3.2.

- **Elements** are the basic units of the web page. They have two subtypes — an element can be either a HTML tag or a component.
- **HTML tags** are the elements, that are displayed in the browser as a part of HTML document and the DOM tree. Tags can contain child elements as well as text. Tags may also have properties.
- **Components** are non-trivial, named elements, which are defined within the application. They have one root elements and can have child elements as well as properties. Components render themselves according to the input they have. Components can be parts of other components.
- **Properties** are named values which define or modify the behavior of tags or components, for example HTML proprerty `style` or `id`.
- **Simple properties** have atomic values, i. e. a string or a number.
- **Composite properties** have non-atomic values, like arrays or objects.

3.4 Requirements specification

3.4.1 Functional requirements

- The users will be able to specify definitions of components. The component has to have a name and a root element.
- The user will be able to define, modify and remove the children of components. The childern themselves will be able to have children, too.
- The user will be able to define, modify and remove the properties of the component's root element or any of its children.
- The components will be composable into a hierarchical structure.
- In the definition of the component, there has to be a way to define passing of the properties from the component to its children, as well as other data manipulation — accessing fields of an array or an object, iterating through arrays, and conditional values.
- The users will be able to specify sample input data (properties) for each component.
- The application will render the defined components according to the specification and sample input data.
- The application will store the definitions of components and the sample data to a storage for future use.
- The application will provide a way to download component definitons and sample data.

3.4.2 Non-functional requirements

- The rendering of the component would be immediate, no save or page refresh will be needed.
- The component definition and sample data will also be stored immediately.
- The application will be accessible on the internet and available for download and execution on the local computer.
- The downloaded definitions and sample data will be in a format usable with Codiscent Projector Template Generator.

Chapter 4

Design

4.1 Ambidex

Ambidex is a JavaScript library by Brenton Simpson for developing isomorphic React applications. It was presented on React.js Conf 2015. It is developed in the eBay Mobile Innovations lab. It uses several other libraries in concert to bridge the differences between server and browser environments. Doing so, it makes the application isomorphic, rendering on both the server and the client using the same application code. [19]

I will use Ambidex to make Instinct UI isomorphic, i. e. have it render on the server first, then use the browser environment to make its usage dynamic, without the need to re-render the whole page on each change. It will also allow me to develop it more quickly as the changes to the components will be immediately shown within the browser without the need of refreshing the browser.

4.1.1 React

React components are the basis of the view presented with Ambidex. Ambidex leverages the ability of React to render the componet either in the browser, mounted to a DOM node, or on the server as a static string. I described React in a detail in section 1.2.4.

4.1.2 React Router

React Router is a JavaScript library, created by Michael Jackson, Ryan Florence and other contributors, used for mapping the URL to React components that corresponds to that URL. [26] It also includes a component for traversing between the pages without refreshing the page, called `Link`. The router mounts appropriate component on page transition and pushes the new page to browser history, so the Back button remains operational.

React Router also supports having "nested routes", which enable having a common component (for example, the page layout) for the parent route with specific components as children for the descendant routes. It also supports routes with parameters, routes for redirection or page-not-found route.

Aside from the routing, the router supports intercepting the transitions to and from the components, then aborting, redirecting or retrying the transition.

Ambidex uses React Router first to map the request URL to the right component(s), then for calling Flux (Reflux) actions according to the router state. This allows Ambidex to have the right data in the data stores in order to render the page on the server.

4.1.3 React Hot Loader

React Hot Loader is a JavaScript library, created by Dan Ambramov, which uses Webpack (see below) Dev Server to immediately push changes made to the React components, without the browser refresh. This enables live editting of React code, changing the appearance and behavior of the page on the fly. [27]

Ambidex takes this one step further, because it generates the needed `<script>` tags automatically.

4.1.4 Webpack

Webpack is a JavaScript library, created by Tobias Koppers and contributors, used to bundle JavaScript files for the browser. This allows using JavaScript files created for the server environment (for example, Node.js libraries) to be executed in the browser. It also packs multiple files into a single bundle or multiple chunks, that are loaded asynchronously. This reduces number of HTTP requests made, which enhances initial loading time. The bundler also handles dependencies and omits unneeded files, reducing the total bundle size. [28]

Aside from bundling JavaScript files, webpack can also use loaders to bundle various other file formats (for example JSON, CoffeeScript or CSS) and supports their transformation (for example, transpiling CoffeeScript to JavaScript or JavaScript with ES6 features to ES5). It is modular, so a custom loader for any format may be used.

Webpack Dev Server is a JavaScript library by the same author, used in development to serve a Webpack application and reload it when the source files change.

Ambidex uses Webpack to bundle all JavaScript source files, as well as the Sass stylesheets. Ambidex also uses Webpack Dev Server in development along with the React Hot Loader to display the changes of the code immediately.

4.1.5 Reflux

Reflux is a Flux implementation by Mikael Brassman. It deviates a little from original architecture, being more of a traditional publication-subscriber model, but still with actions, stores and one-way data flow. [29]

In Reflux, the dispatcher is merged with the action — the actions are function objects, which can be *trigerred* just by invoking them and listened to by registering the listener. The listener registers themselves to the action directly with a function (method) to be called when the action is triggered. The data stores are also a little different as they can listen to each other, allowing to create aggregate stores.

Reflux does not enforce the restriction put up by Flux dispatcher — an action can be called during handling another action. However, it is a good practice to trigger events only in components (and not during handling store changes) to prevent infinite loops (even though the events can be trigerred asynchronusly).

Ambidex uses Reflux (in the stable version 0.1.4, in 0.2.0, it will be replaced by author's own Flux implementation) as a data store. On each HTTP request to the server, Ambidex triggers actions according to the router state (the page requested) and, waits for the defined stores to trigger, then renders the page. The store trigger when they have loaded the data needed for rendering the page, so the page will be rendered correctly. Ambidex then sends the store state from the server as a part of the webpage in plain JavaScript. The client then sets the state of client stores accordingly. This the circumvents need of loading the data aside from the page in the browser, reducing load times.

On the client side, Ambidex triggers the appropriate actions on each router transition, but renders the components immediately. When the stores have loaded the data, they trigger, notifying the component, which re-render with the loaded data.

Ambidex's author extended Reflux by adding an option to generate Reflux actions and stores from plain object defintion as well as "dehydrating" the stores when sending them from the server add "hydrating" them on the client.

4.1.6 Mach

Mach is a JavaScript HTTP server and client library by Michael Jackson and contributors. [30] It runs both in Node.js and in the browser. It uses the anynchronous model of JavaScript to serve a lot of clients simultaneously.

It maps HTTP requests to (asynchronus) function calls. It also uses JavaScript Promises (see section 5.1.1) for composing asynchronous function calls, for example file operations.

Other features of mach include streaming of both requests and responses, composability and usage of middleware functions and robustness, as it prop-

agates errors along the promise chain and up the call stack to simplify error handling and debugging.

Ambidex uses mach to serve the application on the server, as well as optionally using its middleware, for example to serve static files.

4.2 Data model

The data model for Instinct UI has to be flexible enough to handle all kinds of elements as well hierarchic data. I will therefore use JSON format of the data. The entities in the data model include:

- **Components**, which will be mapped to objects with a `root` property beginning with a capital letter consisting of component's name.
- HTML **tags**, which will be mapped to objects with a `root` property beginning with a small letter consisting of tag's name.
- **Syntax** objects, expressing component's props, loops and conditions, which will be mapped to objects with a `root` property beginning with a dollar symbol, for example `$val`. The parameters of the syntax objects will be mapped to attributes of the syntax object. The syntax values will be:
  - `$val` for referencing component's props
  - `$index` for referencing fields of objects and arrays
  - `$for` for iterating objects and arrays
  - `$if` for conditional values
  - `$merge` for mixing component's props with default or overriding values, for example mixing components's default style with the style passed to the component as prop
- **Text** nodes, which will be mapped to strings.
- Component and tag **children**, which will be mapped to `children` attribute of component and tag objects, either as an array (in case of multiple children), as a component or tag object (in case of single element child), or as a string (in case of single text child).
- Component and tag **properties**, which will be mapped to other attributes of component and tag objects.

Component definitions and sample data will be saved in files `component.json` and `data.json` as an object with component names as keys and component definitions (or the sample data) as values.

For example, the code for defining a box component, which renders the black border around given child elements, as well as adding the given name:

```
{
    "root": "div",
    "style": {
        "root": "$merge",
        "merge": [
            {
                "root": "$val",
                "val": "style"
            },
            {
                "border": "3px solid black",
                "float": "left",
                "padding": "5",
                "margin": "5"
            }
        ]
    },
    "children": [
        {
            "root": "div",
            "children": {
                "root": "$val",
                "val": "name"
            }
        },
        {
            "root": "$val",
            "val": "children"
        }
    ]
}
```

4.3 Server API

Aside from serving the application itself, the server will have to have an API for loading and saving the component definitions and the sample data. The data will be requested from Instinct UI itself and also from the user when downloading the definitions needed for the PTG. Instinct UI will use a JSON API, while the PTG definition will need to be available as XML.

I decided to create a REST API, which uses HTTP methods in a semantic way:

- **GET** method to request the whole collection of data or just a single entity

- **POST** method to create a new entity
- **PUT** method to update (replace) the whole collection of data or just a single entity
- **DELETE** method to delete the whole collection or just a single entity

The list of available endpoints is shown in table 4.1.

4.4 Data stores and actions

The Reflux data stores contain the data displayed in the application. They respond to actions by changing the data and then notifying the components (see below) about the change. In Instinct UI, only the `Main` component listens to the stores. Instinct UI includes following data stores:

- **Components** store contains all component definitions in the application.
- **Data** store contains all sample data in the application. It listens to the component store in case a component is created and creates an empty sample data object in that case.
- **CurrentComponent** store contains the name of the current component, as given by the router.

The Reflux actions are called by the components (see below) as in response to user input. Instinct UI includes following actions (also shown in figure 4.1):

- **getComponents** action is called by Ambidex on every request. Components store listens to it and responses by fetching the data from the API. The store triggers when the data are loaded.
- **createComponent** action is called to create a new component. The Components store listens to it and responses by sending a POST API request. It then GETs the definition of created component and then triggers.
- **updateComponent** action is called when a component definition is changed, with appropriate data and component name. The Components store listens to it and responses by sending a PUT API request, then immediately triggers.
- **renameComponent** action is called when component's name is changed. It is called with both old and new name. Both Component and Data stores listens to it and rename the component and all references to it,

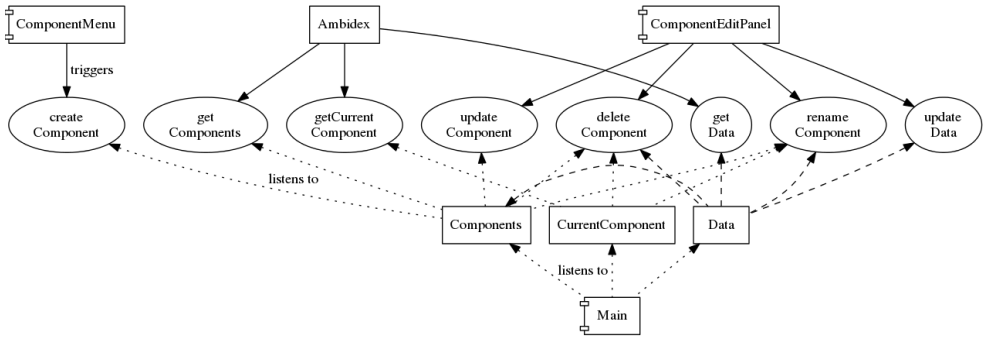

Figure 4.1: Flux stores and actions

then send a PUT API request, then immediately trigger. CurrentComponent also listens to it, and if the old name matches the current component, it changes the current component to the new name and triggers.

- **deleteComponent** action is called when the user deletes a component. Each store listens to it: the Components store removes the component from its state, calls a DELETE API request, then triggers; the Data store removes the component from its state and triggers, too; the CurrentComponent compares the deleted component's name with the current component and if they match, it sets its state to `null` and triggers.

- **getData** action action is called by Ambidex on every request. Data store listens to it and responses by fetching the data from the API. The store triggers when the data are loaded.

- **updateData** action is called when the sample data of a component change, with appropriate data and component name. The Data store listens to it and responses by sending a PUT API request, then immediately triggers.

- **getCurrentComponent** action is called by Ambidex, when the request contains the `name` parameter. The store sets its state to the name given, then triggers.

4.5 User interface

To design the user interface of Instinct UI, I first created a wireframe (shown in figure 4.2), the used the wireframe to determine the components and their relationships. In the end, I decided to create these components (shown in figure 4.3):

- **Main** component will be the container for all other components. It will also own the application state, getting it from the stores and passing it to its children (ComponentMenu, ComponentEditPanel and Renderer) as props.
- **ComponentMenu** will display name of all components in the application as React Router's Links to enable changing the current component. It will also include a button for adding a component (triggering createComponent action) and two HoldButtons for scrolling the menu horizontally.
- **HoldButton** will be a small component, which would fire events each 10 miliseconds as long as it is clicked on.
- **ComponentEditPanel** will manage the editting of the current components. It will consist of an Input for component's name, two JsonEditBoxes, one for component definition, one for sample data. It would trigger appropriate action (renameComponent, updateComponent or updateData) when the name, definition or the sample data changes. It will also contain a button for deleting the current component (by triggering deleteComponent).
- **Input** will be component for editting text data. It would fire an event when the text changes. When its default value (given in props) changes, it will wait until it is not active, then changes the input value. This prevents overrwriting of the changes and changing cursor position while writing.
- **JsonEditBox** will be component for editting JSON data. It would parse the text in its textarea everytime it changes. If the text will be a valid JSON, it will fire an event, otherwise, it would display an ErrorMessage. As with Input, JsonEditBox does not change the text if it is active.
- **Renderer** will be the heart of Instinct UI. It will render the current component according to its definition and sample data. It will display the ErrorMessage in case of an error.
- **ErrorMessage** is a visual component for displaying an error.

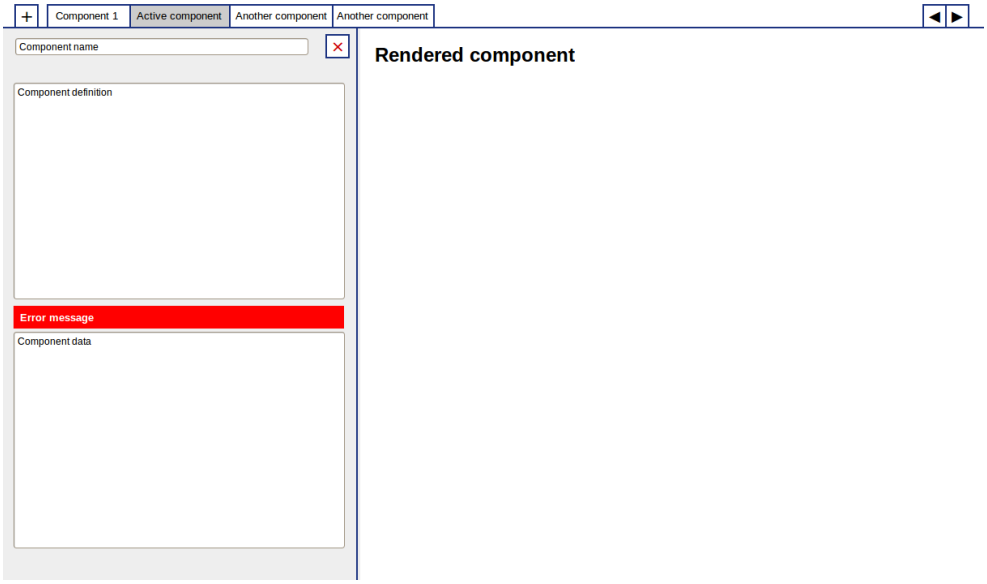

Figure 4.2: Instinct UI wireframe

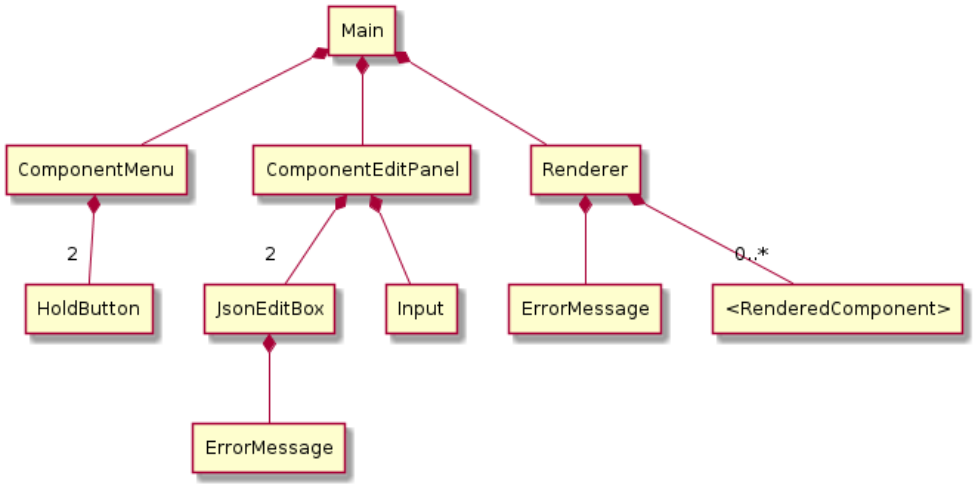

Figure 4.3: Components in Instinct UI

method	URL	Description
GET	/components.json	returns all component definitions as JSON
	/components.xml	returns all component definitions as XML
	/data.json	returns all sample data as JSON
	/data.xml	returns all sample data as XML
	/components/:name.json	returns given component definition as JSON
	/components/:name.xml	returns given component definition as XML
	/data/:name.json	returns sample data for the given component as JSON
	/data/:name.xml	returns sample data for the given component as XML
	/:name.xml	returns the data and the definition for the given component as XML
	/:all.xml	returns all definitions and sample data as XML
POST	/components.json /data.json	both end points create a new component with definition and data given as JSON, then redirect to appropriate GET endpoint
PUT	/components.json	updates all component definitions with the data given as JSON, then returns the definition
	/data.json	updates all sample data with the data given as JSON, then returns the data
	/components/:name.json	updates given component definition with the data given as JSON, then returns the definition
	/data/:name.json	updates sample data for the given component with the data given as JSON, then returns the data
DELETE	/components.json /data.json	not implemented (both endpoints return status 403)
	/components/:name.json /data/:name.json	both end points delete the given component, then return remaining components or data as JSON

Table 4.1: List of available API endpoints

Chapter 5

Implementation

5.1 Server implementation

I implemented the server API (described in section 4.3) using the mach web server (described in section 4.1.6). The JSON part was pretty straightforward, the generation of XML for the XML part is described in section 5.5.1.

Because the API server may run on a different URL than the frontend, I needed to add an option to send the Cross-origin resource sharing (CORS) headers to the response to allow the clients loading the API with an AJAX request from a different origin. I did this by creating a mach middleware (basically a function), which adds the desired headers to each request.

I also decided to add a configuration option to either run the API server directly as a middleware (using the `mach.map` method) within Ambidex or as a stand-alone process. I did this so the server could be reloaded without restarting the client application (as Ambidex does not reload files not used during the rendering and takes relatively long time to start up). This also allows to run the server on another host and/or port, if needed.

5.1.1 Promises

Promises (also called futures or deferred) are placeholder values for computations or operations which have not completed and returned the desired result yet. Past approach to this problem was using the callbacks, which make the code messy and hard to reason about. On the other hand, Promises can create long operation chains (quite similar to Unix pipes) easily and utilize non-promise functions in the process.

As a promise implementation, I used the *when* library by Brian Cavalier and other contributors. When supplies promise wrappers for callback-based filesystem operations of Node.js into promises, which allowed me to easily perform additional operations on the results of the operations. I could also easily chain those operations, for example read a JSON file, then parse the

contents with `JSON.parse`, edit the data, then write them back to the file. [31]

5.2 Reflux stores

I implemented the Reflux stores (described in section 4.4) as plain JavaScript definitions, because Ambidex creates the stores in its own way from these definitions to keep the store creation isomorphic.

Because the Components store and the Data store were very similar and consisted of a lot of duplicated code, I refactored this code into four mixins. The *apiCalls* mixin uses mach to send HTTP requests to the API. The *loadSendData* mixin define the throttled (see below) methods for loading and updating the store data. The *updateCount* mixin adds a counter of active updates to the components. While this counter is greater than zero, the GET requests are not made nor do their result (if the request was sent before the update was made) reflect to the data stores to prevent overwriting of the changes which are on the way to the server. And the *rename* mixin adds helper method for renaming the component and references to it in the store data.

Furthermore, I converted the store state to be immutable in the end — instead of mutating the store object (component or data dictionary), a new dictionary with changed and unchanged data was created. This enabled using PureRenderMixin in components. This mixin overrides default lifecycle method shouldComponentUpdate, so that when component's parent re-renders (and the component receives new props and state), but the props and state remain the same, the component does not re-render itself. Creating of virtual DOM and DOM reconciliation is thus skipped, improving overall performance. [32]

Usage of PureRenderMixin has two prerequirements — the first is aforementioned immutability of props and state. Because PureRenderMixin does only shallow compare, it would skip the update if something was changed deep in the (mutable) state or props objects. (The alternative to immutability is to use forceUpdate method to bypass the shouldComponetUpdate method when the change is detected.)

The second requirement is the namesake "pureness", i. e. the resulting DOM must only depend on the props and state and always be the same given the same props and state. This includes "pureness" of all the child components. Because I used PureRenderMixin for all components (aside top-of-the-hierarchy App component, which uses RouteHandler given by router) and always used just props and state in render methods, this requirement was fulfilled as well.

5.2.1 Throttling

As I don't want to send server request for every typed letter, I use a technique called "throttling", which is time-based regulation of code execution rate. Request for GETting and PUTting the data are limited to run at most each 60 seconds and 1 second, respectively. Latest call of PUT is always executed, sooner or later, to surely save the data, whereas the GET request is discarded, so finishing it later would not overwrite any modifications the user would make between making the delayed request and completing it.

Also, as the PUT and DELETE changes are displaying optimistically, to prevent race conditions, when there is an update waiting for completion (response from the servre), GET request are neither sent nor are the received data written to the data stores.

I implemented the throttling with promises (using the when promise library) — the throttle function returns another function which:

a) returns a fulfilled (resolved) promise when called for the first time or after the given time threshold

b) return a rejected promise when called again within the time threshold and it should not be called later (for the GET requests)

c) return a (undecided) promise when called again within the time threshold otherwise (for the PUT requests); this promise is fulfilled after the time threshold or rejected when the function is called again within the time threshold

This implementation enables cleaning up after the request, whether it was really made or not. Before the throttled function is called, the *updateCount* mixin counter of active update requests (described above) is incremented. Using the finally method (which behaves like finally exception construct, i. e. is executed whether an exception was raised or not), the counter is decremented again after the request was completed or discarded.

5.3 Components

I implemented the Instinct UI components, described in section 4.5, using React and JSX. The implementation is not very complicated, aside from the Renderer component.

The Renderer component contains the core of Instinct UI functionality. It converts component definitions to React classes using the `createComponent` and `React.createClass` methods. These classes are just thin closure wrappers around the definitions, though. Most of the rendering is done by the `createElement` method, which is called when the created class is rendered by React.

When `createElement` method is called, the type of the rendered element is first determined by the `getType` method, which looks at element's `root` attribute. If the rendered element is the root of a component, the syntax objects in it are evaluated using the `replaceSyntax` method. The children of the element are normalized and created recursively. The React element is then created by `React.createElement` method and returned.

5.4 Deploying to Heroku

I decided to host my application on Heroku. Heroku is a cloud (platform-as-a service) hosting service supporting Ruby, Node.js, Python, Java, and PHP. It runs the application using "dynos", dynamic containers which behave like virtual machines. The dynos are named and priced by performance (1X, 2X, PX). The application can be run with multiple dynos, increasing redundancy. Each user has enough monthly free dyno hours per application to run the application in a single 1X dyno free of charge. [33]

Application are deployed to Heroku using the distrubuted version control system Git. The user logs in to Heroku using the Heroku command-line tool, then deploys the application with `git push heroku master` command. The configuration is saved with each deploy, allowing to rollback changes in case of need.

To make Instinct UI work on Heroku, I created a `settings.heroku.js` file and added `NODE_ENV` and `URL` environment variables to Heroku. Instinct UI uses the `NODE_ENV` variable to choose the right settings file and the `URL` variable to send the API requests to the right server. In Heroku settings, the API server runs as a middleware within the application and is mapped to `/api/` path.

I deployed Instinct UI to `http://instinct-ui.herokuapp.com/`. Because Heroku limits the RAM of 1X dyno to 512 MiB and the application used more, I had to bother Node.js a little and limit its old memory space to 420 MiB. This limit was high enough for the app to actually run and low enough that the whole used memory fitted in the 512 MiB provided by Heroku.

5.5 Using projective technology

To use the Codistent's projective technology, I first had to generate XML data specification from the JSON data used by Instinct UI. Then I had to create a template to use with the PTG. Doing this, I had to take different JavaScript environments, in which the generated component could be run, into account. I also had to run the PTG simply and repeatedly.

5.5.1 XML generation

I implemented the XML generation as a part of the API server. The server loads the JSON data from the appropriate file, then, as a part of a promise chain, it generates the XML data from the JavaScript definition. The entry point of the XML generation is the `generateXML` function. It takes the JS definitons as a parameter, restuctures them and then creates the XML builder library by Ozgur Ozcitak (and contributors). [34]

Because the PTG views the data in terms of rows and columns, the JS tree data had to be flattened by the depth-first search. Instead of elements including each other, the parent elements had to reference their children using an element index. Also, the arrays and complicated objects in element properties had to be turned into separate elements. The XML therefore uses different element names for element references and atomic values (like strings and numbers) to distinguish them. The XML also includes the type field to differentiate objects, arrays, tags, components and syntax objects.

5.5.2 Generator template

I chose to generate React components using the PTG. Although they would look the same, they would not be the same as the rendered components in Instinct UI. The Instinct UI rendered generated components dynamically, whereas the generated components will be static and as close as possible to classic React components. The generated components would also be independent on Instinct UI.

The generator template therefore has to create a JavaScript code dereferencing the elements referenced in the XML specification. To do that, I decided to generate a function for each element, then call the function from the parent element. The props of the component would be passed down as a function parameter. The function code differs according to the element type:

- **Objects** assemble their properties, calling other element for non-atomic values, then return the properties.
- **Arrays** assemble their children, calling other element for non-atomic values, then return the children.
- HTML **Tags** assemble their properties, like the objects, assemble their children like the arrays, then call `React.createElement` with their root property, other properties and children, then return the created element.
- **Components** assemble their properties and children like the tags, then call `React.createElement` with the generated component class (see below), properties and children, then return the created element.
- **Syntax** objects evaluate the syntax accordingly, then return the result.

In case of component definitions, a React class is generated for each of them. The class just calls the appropriate function in its render method. In case of sample data, sample render function with the name of the component is also generated. This function returns the React element of the desired class with the sample data.

Aside from the component template, I also created a simple template for generating HTML. The HTML just includes appropriate JavaScript files (it relies on the naming convention) and render the components with sample data in the XML.

5.5.3 Universal Module Definition

When developing Instinct UI, I could rely on having Node.js on the server, while Ambidex and Webpack handled transformation of the script for browser. When creating generated components, I had to take different JavaScript environments into account. These environments differ mainly in the way of separating the code and managing dependencies:

- **CommonJS**, which is used for example in Node.js, where each file defines a module, using a `require` function to load dependencies and the `exports` variable to export objects.
- **Asynchronous Module Definition (AMD)**, which is used for example by module loader RequireJS, where the modules are defined using the `define` function, which also declares the dependencies.
- **Browser environment**, where scripts occupy the global namespace and the dependencies are passed by simply loading the file.

I therefore included an Universal Module Definition (UMD) part to the template code.[35] The UMD wraps the code in the function wrapper, detects the environment and calls the function immediately with appropriate parameters. Inside the function, the code functions the same on all environments. On CommonJS and AMD, the wrapper `require`s React (or declares it a as a dependency) and exports an object consisting of all defined components and sample render functions. In the browser, the wrapper consumes the global React object (which has to be created by including React script in the browser) and defines (or extends) an object called InstinctUI with the components and sample render functions.

5.5.4 Running the generator

As I didn't have access to the GES licence, I was reliant on running the PTG directly from the command line. However, the PTG accepts the template as a command line parameter, which is very inconvenient. Therefore, I decided

to create a Windows command line script to run the PTG while specifying the template file as a parameter. The script reads the file line by line using the `set /p` command and delayed expression expansion. The quotes in the template string are then escaped to use the string as a command line parameter delimited with quotes. The PTG is then run with desired parameters.

I also had a small problem with the character encoding when generating the components. The (non-ASCII) UTF-8 characters in the the template generator application were interpreted as two or more characters, therefore generating scrambled characters. The problem was made even worse by the Windows command line transcribing the output to another encoding, turning the majority of "bad" characters into question marks.

The codepage problem was finally solved by having the specification files in UTF-8 and changing the console encoding (with the `chcp` command) to Windows-1255. Apparently, the generator inperprets the input file (but not the template string) as being encoded in Windows-1255 and produces Windows-1255 output as well. But when the command line saves the scrambled characters to the file unchanged, they form the original UTF-8 characters. This solution is working, because both UTF-8 and Windows-1255 are compatible with ASCII, so the PTG syntax characters are unaffected by the codepage change.

5.6 Testing

For the testing of both Instinct UI and the component generation, I constructed three sets of components. The first set is the default component set, showing the features of Instinct UI. The second set was used to test the XML generation and the PTG, so it uses almost all possible types of elements. The third set was used as a proof of concept, described in chapter 6.

During the testing, I created the components using Instinct UI each time and compared the rendered components to the intended and expected output. After creating the components with Instinct UI, I downloaded the appropriate XML files, ran the PTG and then compared the generated components to the components rendered in Instinct UI.

When developing the API part of Instinct UI, I tested server responses to browser requests from Instinct UI and to requests made by cURL.

Concerning other parts of Instinct UI, I simply tested their behavior in the browser.

Chapter 6

Proof of concept

As o proof of concept, I decided to design the user interface for a simple e-shop. I will create three screens of the e-shop — homepage / category listing, product detail and the shopping cart detail.

6.1 Component design

Before I decomposed the e-shop interface into components, I made a wireframe for each of the three pages. These wireframes are shown in figures C.1, C.2 and C.3.

I used these wireframes to decompose the UI into these components:

- **Homepage**, **ProductPage** and **CartPage** are the root components for the pages. They use the Layout component with ProductList, ProductDetail and CartDetail components, respectively.
- **Layout** is the top-level component of the document. It consists of CartInfo, Header and CategoryList components, displaying the given children as the main content of the page.
- **Header** contains the top heading of the page.
- **CartInfo** shows user's shopping cart state (number of items and the total cost) as well as the (fake) link to cart detail.
- **CategoryList** shows the tree of e-shop categories. Each category is shown as a single CategoryListItem element.
- **CategoryListItem** displays single category as a list item with (fake) link to the category page.
- **ProductList** is the main component of the Homepage. It shows the name of the category and products in the category, each as one ProductBox element.

- **ProductBox** shows one product on the homepage. It displays product's name and image as well as a (fake) link to product's detail page and the AddToCartButton component.
- **AddToCartButton** shows the input field and submit button for adding a product to the shopping cart.
- **ProductDetail** is the main component of the ProductPage. It shows the product's name, description, image and a AddToCartButton.
- **CartDetail** is the main components of the CartPage. It shows the table of user's shopping cart contents, each item as a CartRow. It also show a (fake) link to proceed with the order.
- **CartRow** shows one item of user's shopping cart. It shows the product name, quantity, unit cost and total cost.

6.2 Implementing in Instinct UI

I created an Instinct UI component for each of components. I defined component's style, structure and composition via the component textarea. I added sample data to the components via the data textarea.

All product, category and cart data are passed from the sample input data through the components' props. Note that all the data are statically passed through the props, as Instinct UI (in its current version) is not capable of props manipulation, such as multiplying the item quantity with the unit cost to produce total cost.

A screenshot of Instinct UI with the Homepage component is shown on figure B.2.

6.3 Generating the components

I downloaded the components.xml, data/Homepage.xml, data/ProductPage.xml and data/CartPage.xml to the directory with the components.tpl and html.tpl files and wrote and executed a simple build script, creating four JavaScript files (one with component definitions, three with sample data for rendering the components) and three HTML files (one for each component):

```
@echo off
call "%~dp0\run.bat" "%~dp0\components.xml"^
    "%~dp0\components.tpl" > "%~dp0\components.js"
call "%~dp0\run.bat" "%~dp0\Homepage.xml"^
    "%~dp0\components.tpl" > "%~dp0\Homepage.js"
call "%~dp0\run.bat" "%~dp0\Homepage.xml"^
    "%~dp0\html.tpl" > "%~dp0\Homepage.html"
```

```
call "%~dp0\run.bat" "%~dp0\ProductPage.xml"^
    "%~dp0\components.tpl" > "%~dp0\ProductPage.js"
call "%~dp0\run.bat" "%~dp0\ProductPage.xml"^
    "%~dp0\html.tpl" > "%~dp0\ProductPage.html"
call "%~dp0\run.bat" "%~dp0\CartPage.xml"^
    "%~dp0\components.tpl" > "%~dp0\CartPage.js"
call "%~dp0\run.bat" "%~dp0\CartPage.xml"^
    "%~dp0\html.tpl" > "%~dp0\CartPage.html"
```

The screenshots of the generated files are shown in figures C.4, C.5 and C.6.

6.4 Summary

After generating the files, I measured the lines of code written to the lines of code generated (all data shown in table 6.1). The code generation produced circa twice as much lines as I have written in JSON. This is not very much, considering the structure and "density" of generated code. (However, the JSON is also quite wordy.) Comparing the generated JavaScript code to TPG's actual input, the XML files, generated from JSON by Instinct UI, is even worse — only two thirds of the lines in XML were generated by the TPG. However, the XML has no other information than the JSON, it is just a lot wordier.

Another interesting thing it that 52 of 179 elements in resulting components.js (29%) are the `$val` element. In most cases, they could be generated directly, saving the function call. Their syntax in Instinct UI could also be simplified for easier writing of components.

All thing considered, it would be probably easier to write the components by hand, but one would lose the benefits of the process — immediately seeing the components in the browser and the ability to eventually generate the components, using another template and another technology.

It should also be noted, that while only the static data from Instinct UI were used here, the components could be set up with any data source, for

	JSON	XML*	JavaScript + HTML
components	697	2490	1730
Homepage	74	356	226
ProductPage	50	245	172
CartPage	76	370	216
templates**	172	N/A	N/A
total	1069	3461	2344

Table 6.1: Lines-of-code comparison (* – after formatting; ** – not in JSON)

example another React component getting the data from the server. One could also define a callback parameter of the components, pass the callback down the component hierarchy and use it as the event handler for text inputs and buttons (although this is quite React-specific way). However, this approach would require creating all the callbacks at once, in the parent component and passing it in a way corresponding to data rows. For example, the parent component could add a callback to each of passed product (creating specific callback for each product with `bind` method). The generated component would then set the passed callback as an event handler for a HTML tag.

Chapter 7

Future work

Although the application satisfies the given task and the components can be generated from its output, it is far from perfect. A lot of features were omitted from the application due to scope and time requirements.

7.1 User interface

Because I focused on the rendering components and generating code, the input of component definitons and data is still very basic. It can be improved by using an embedded code editor (with features like code folding and syntax highlighting), for example Ace Editor created by Ajax.org. [36] I even attempted to use React-Ace, a React component for Ace by James Hrisho, but it did not work with Ambidex / Webpack. [37]

The other approach to user interface would be a complete grafic interface for creating and editing the components, similar to desgin view included in many IDEs, including a tree view of component's children, property definition pane and canvas with draggable components.

7.2 Better validation

Server implementation currently does not validate the given input, other rejecting an ill-formed request or JSON. If given and invalid input (such as unknown dollar-sign syntax), it prevents XML generation of the component to succeed until the error is corrected.

The application itself also needs some more validation. Some constructs (such as nested `<p>` tags or content directly in `<table>` tag) are modified by the browser, causing React to leave the invalid component on the page until it is refreshed away. Another error to catch is generating an infinite recusion (that can happen in plain React, too). This kind of validation is unfortunately complex, because by the time child component's render method is called, its

parent's render method has already returned. Thus, some other solution is needed for this "context-aware" validation.

7.3 Storage and concurrency

Instinct UI currently stores the data in JSON files on the file system. This does not work with Heroku (Heroku has so-called *ephemeral filesystem*, which discards the changes everytime the application dyno is restarted) and supports only limited concurrency. There is also no personification as all users work with the same data, potentially overwriting work of the others. The solution would be to use a datatabase storage on the server side, along with more fine-grained requests from the client.

A related task is separating user's components into multiple projects and saving and loading different sets of component data. (This can be currently done only by switching the JSON files in the data folder.)

7.4 Component features

In current state, generated components are just a static representation of input data, they have no state and cannot react to user input, aside from keeping the state in a (non-generated) parent component and passing event handlers down the generated component structure. There should be a way of propagating changes from user up the component structure or (in case of React and Flux) to a data store.

Inspired by another Bret Victor talk, "Stop Drawing Dead Fish"[38], the solution would be adding a *behavior* attribute (possibly with parameters) to components. This attribute would define component reactions to user input or other events in an platform-independent way. The behavior would be implemented by the Renderer and in the generator template, manifesting the behavior in Instinct UI app and the generated components.

Also, the specifications could be extended with more functionality themselves, to add well-defined abstract things like string concatenation or arithmetics. The platform-dependent things could be abstracted into "built-in" components implemented directly done in target technology. It is up to consideration whether to implement these functions in a library (so there would be an Instict UI library for React and then for any other platform needed), or whether to include the code in templates and add it to components with code generation (for example, the current `$merge` is generated directly to the using element).

7.5 Ambidex

While the server-side rendering is powerful and Ambidex is definitely cutting-edge, it is still in early stage of development (as of April 2015) and undergoes a shift in included libraries (between its versions 0.1.4 and 0.2.0, currently in beta). It causes Instinct UI to consume a lot of memory (400 MiB, paradoxically more in production mode) and to start up slowly (which is partly because I limited its memory, so it barely fits Heroku memory limits). The question is whether to replace Ambidex of forfeit isomorphism altogether.

7.6 Direct integration

Generating created components now requires downloading the XML files and running the template generator manually, which is not optimal. The generator could be executed directly from the application server, although this would force the server to run on Windows. Another solution would be to have a desktop applications which communicates with Instinct UI backend and run the generator locally. Both solutions require integrating (and distributing) Codiscent software, which may not be possible (e. g. for licensing reasons).

Conclusion

The goals of this thesis were to create a tool for object-oriented development of web user interface, which would respect principles given in Bret Victor's talks, and to explore possibilities of projective technology in user interface design.

To meet the first goal, I performed a review of current object-oriented languages, libraries and tools for creating user interface (see chapter 1). I analyzed the problem (see chapter 3) and designed (see chapter 4) and implemented (see chapter 5) a tool for creating them, called Instinct UI. This application was deployed to `http://instinct-ui.herokuapp.com/` and its source codes are published on GitHub (`https://github.com/redwormik/instinct-ui`), allowing other people to contribute to its development.

To meet the second goal, I first performed a review of Codiscent's projective technologies (see chapter 2). Then, I wrote a template for Codiscent Projector Template Generator and processed the XML output of Instinct UI with the PTG to generate the React component code (see section 5.5). I tested Instinct UI and the code generation by using two other component sets (see section 5.6) as well as designing the user interface of a simple e-shop (see chapter 6). I publised all the templates, scripts, and input data used for the generation as well as the generated code on GitHub (`https://github.com/redwormik/instinct-ui-tg`).

Although the generation of components in current state is probably not the best application of projective technologies, it has its benefits. The intermediate form (JSON and XML) is independent on the technology used for the resulting components. Therefore, the technology can be changed simply by creating a new template without changing the underlying specifications. The specifications can be also changed and used to generate a new code.

Furthermore, with a better interface for creating the specification (discussed in section 7.1), the effort needed to design a component would be greatly reduced. Also, as the functionality would be added to Instinct UI (discussed in section 7.4), the effort to write the code with the same function-

ality as the generated code would grow, so the code generation's advantages would be more prominent.

Therefore, I see this thesis more as a proof of concept (of specifying the components declaratively, then generating a working code) and a jumping-off point, than as a definitive solution.

Bibliography

[1] Google, Inc. What is Google Chrome OS? [online, video]. 2009, [Cited 2015-05-03]. Available from: https://www.youtube.com/watch?v=0QRO3gKj3qw

[2] Victor, B. Bret Victor, beast of burden [online]. 2014, [Cited 2015-05-03]. Available from: http://worrydream.com/#!/cv/bret_victor_resume.pdf

[3] Victor, B. Inventing on Principle [online, video]. 2012, [Cited 2015-05-01]. Available from: https://vimeo.com/36579366

[4] Victor, B. Drawing Dynamic Visualizations [online, video]. 2013, [Cited 2015-05-01]. Available from: https://vimeo.com/66085662

[5] W3Techs. Usage of client-side programming languages for websites [online]. May 2015, [Cited 2015-05-04]. Available from: http://w3techs.com/technologies/overview/client_side_language/all

[6] W3Techs. Usage of server-side programming languages for websites [online]. May 2015, [Cited 2015-05-04]. Available from: http://w3techs.com/technologies/overview/programming_language/all

[7] Grudl, D.; Nette Foundation. Nette Framework [online]. 2015, [Cited 2015-05-03]. Available from: http://nette.org/

[8] Seaside.st. seaside.st: About [online]. [Cited 2015-05-03]. Available from: http://www.seaside.st/about

[9] Mozilla Development Network. About JavaScript [online]. 2015, [Cited 2015-05-03]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

[10] Joyent, Inc. About Node.js® [online]. 2015, [Cited 2015-05-03]. Available from: https://nodejs.org/about/

[11] Google, Inc. Design Elements - Chrome V8 [online]. 2012, [Cited 2015-05-03]. Available from: https://developers.google.com/v8/design

[12] npm, Inc. npm [online]. 2015, [Cited 2015-05-03]. Available from: https://www.npmjs.com/

[13] Google, I. AngularJS [online]. 2015, [Cited 2015-05-03]. Available from: https://angularjs.org/

[14] Google, I. Angular [online]. 2015, [Cited 2015-05-03]. Available from: https://angular.io/

[15] Facebook, Inc. React [online]. 2015, [Cited 2015-05-03]. Available from: https://facebook.github.io/react/index.html

[16] Facebook, Inc. JSX in Depth [online]. 2015, [Cited 2015-05-03]. Available from: https://facebook.github.io/react/docs/jsx-in-depth.html

[17] Facebook, Inc. Flux [online]. 2015, [Cited 2015-05-03]. Available from: http://facebook.github.io/flux/

[18] Bédard, J. R. Isomorphic JavaScript: Applications which run both client-side and server-side [online]. 2015, [Cited 2015-05-03]. Available from: http://isomorphic.net/javascript

[19] Simpson, B.; contributors. Ambidex [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/appsforartists/ambidex

[20] Wakanda SAS. Application Development Tool — wakanda [online]. 2015, [Cited 2015-05-03]. Available from: http://www.wakanda.org/application-development-tool

[21] Petton, N.; Amber contributors. Amber Smalltalk [online]. 2015, [Cited 2015-05-03]. Available from: http://amber-lang.net/

[22] Petton, N.; contributors. Tide [online]. 2014, [Cited 2015-05-03]. Available from: https://github.com/tide-framework/tide

[23] Codiscent Ltd. codiscent.com [online]. [Cited 2015-05-03]. Available from: http://codiscent.com/

[24] Codiscent Ltd. Codiscent Tools [online]. [Cited 2015-05-03]. Available from: http://codiscent.com/?page_id=296

[25] Codiscent Ltd. Application Development Using Codiscent Generative Technology and Methodology [online]. 2013, [Cited 2015-05-03]. Available from: http://ccm.fit.cvut.cz/wp-content/uploads/2013/10/CodiScent-Technology-and-Methodology.pdf

[26] Jackson, M.; Florence, R.; contributors. React Router [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/rackt/react-router

[27] Abramov, D.; contributors. React Hot Loader [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/gaearon/react-hot-loader

[28] Koppers, T.; contributors. Webpack [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/gaearon/react-hot-loader

[29] Brassman, M.; contributors. RefluxJS [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/spoike/refluxjs

[30] Jackson, M.; contributors. Mach [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/mjackson/mach

[31] Cavalier, B.; contributors. when.js [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/cujojs/when

[32] Facebook, Inc. PureRenderMixin [online]. 2015, [Cited 2015-05-03]. Available from: https://facebook.github.io/react/docs/pure-render-mixin.html

[33] Heroku, I. Heroku [online]. 2015, [Cited 2015-05-03]. Available from: https://www.heroku.com/home

[34] Ozcitak, O.; contributors. xmlbuilder-js [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/oozcitak/xmlbuilder-js

[35] Burke, J.; Osmani, A.; contributors. UMD (Universal Module Definition) [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/umdjs/umd

[36] Ajax. org. Ace [online]. 2015, [Cited 2015-05-03]. Available from: http://ace.c9.io

[37] Hrisho, J. React-Ace [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/securingsincity/react-ace

[38] Victor, B. Stop Drawing Dead Fish [online, video]. 2013, [Cited 2015-05-01]. Available from: https://vimeo.com/64895205

[39] Rajlich, N.; contributors. node-gyp [online]. 2015, [Cited 2015-05-03]. Available from: https://github.com/TooTallNate/node-gyp

Appendix A

Acronyms

AJAX Asynchronous JavaScript And XML

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ASP Active Server Pages

CSS Cascading Style Sheets

DI Dependency Injection

DOM Document Object Model

ES ECMAScript

GES Generative Engineering Studio

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Desktop Environment

JS JavaScript

JSON JavaScript Object Notation

MVC Model-View-Controller

MVP Model-View-Presenter

NEON NEtte Object Notation

OOP Object-Oriented Programming

OS Operating System

PHP PHP Hypertext Preprocessor

PTG Projector Template Generator

REST REpresentational State Transfer

UI User Interface

UTF Universal Transformation Format

WYSIWYG What You See Is What You Get

XML eXtensive Markup Language

YAML YAML Ain't Markup Language

Appendix B

Application screenshots

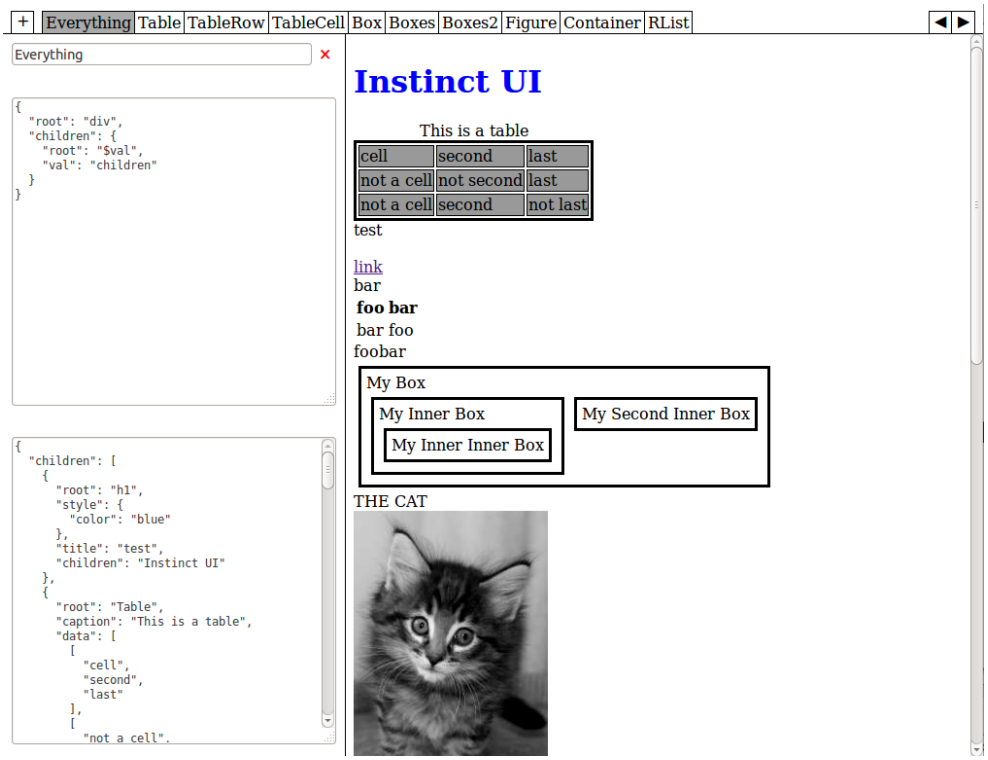

Figure B.1: Screenshot of Instinct UI with default components

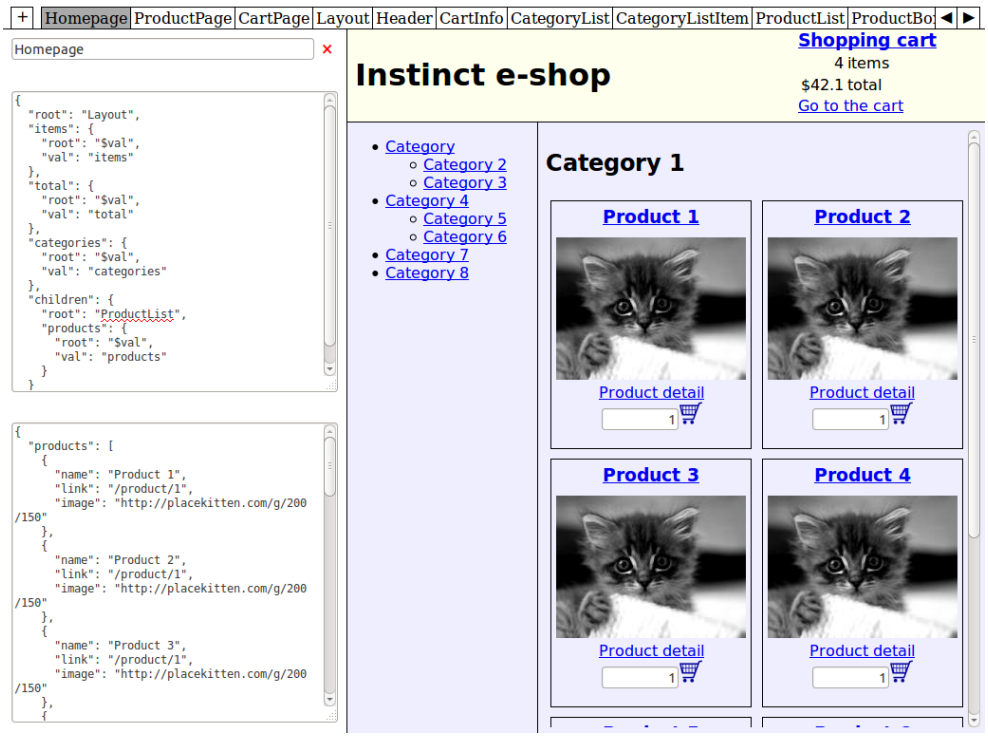

Figure B.2: Screenshot of Instinct UI with proof of concept components

Appendix C

Proof of concept screenshots

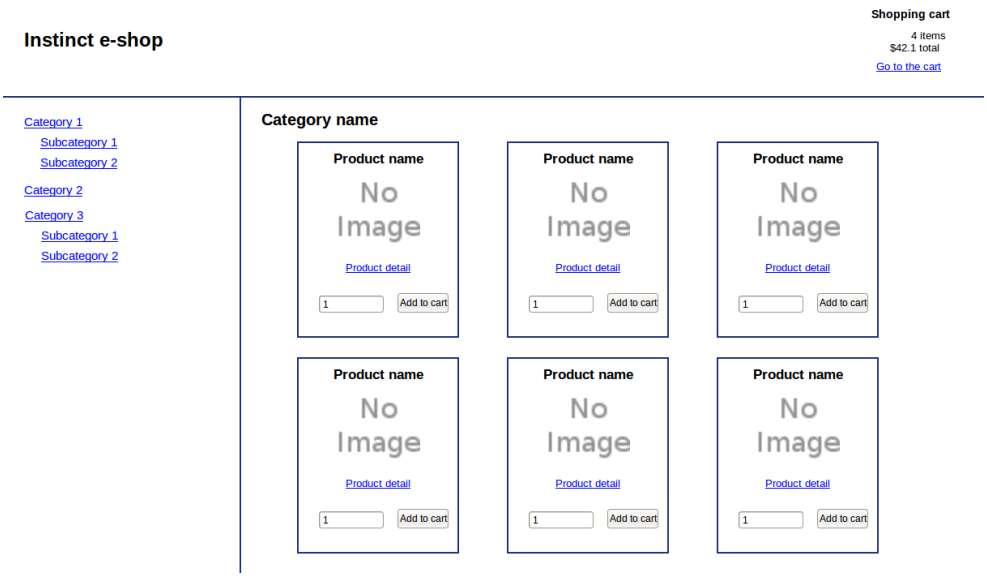

Figure C.1: Wireframe of homepage / category listing screen

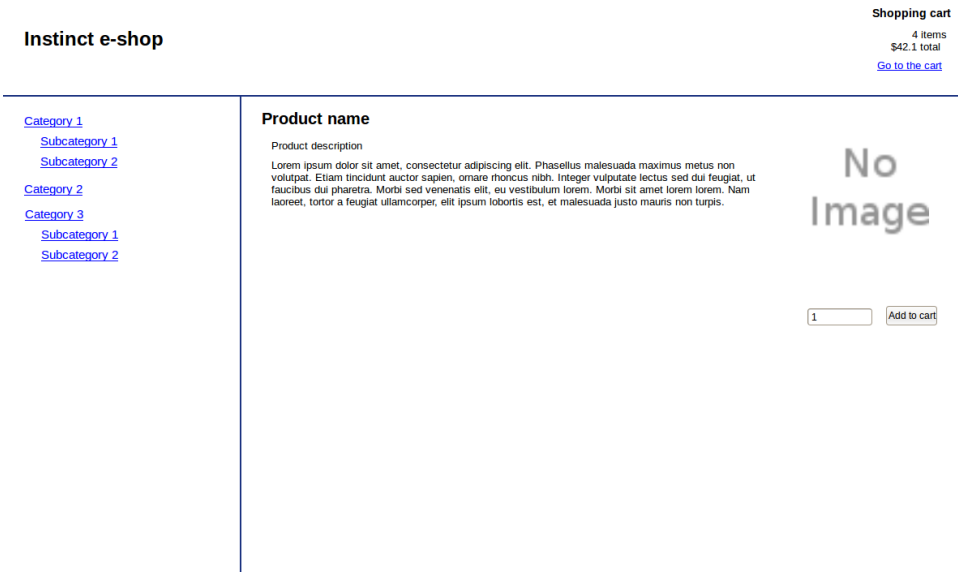

Figure C.2: Wireframe of product detail screen

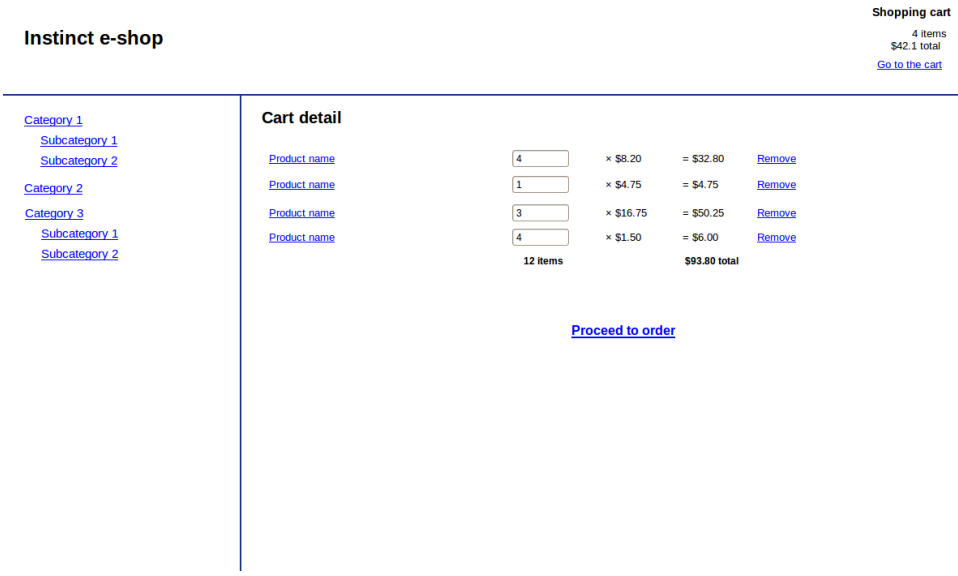

Figure C.3: Wireframe of shopping cart screen

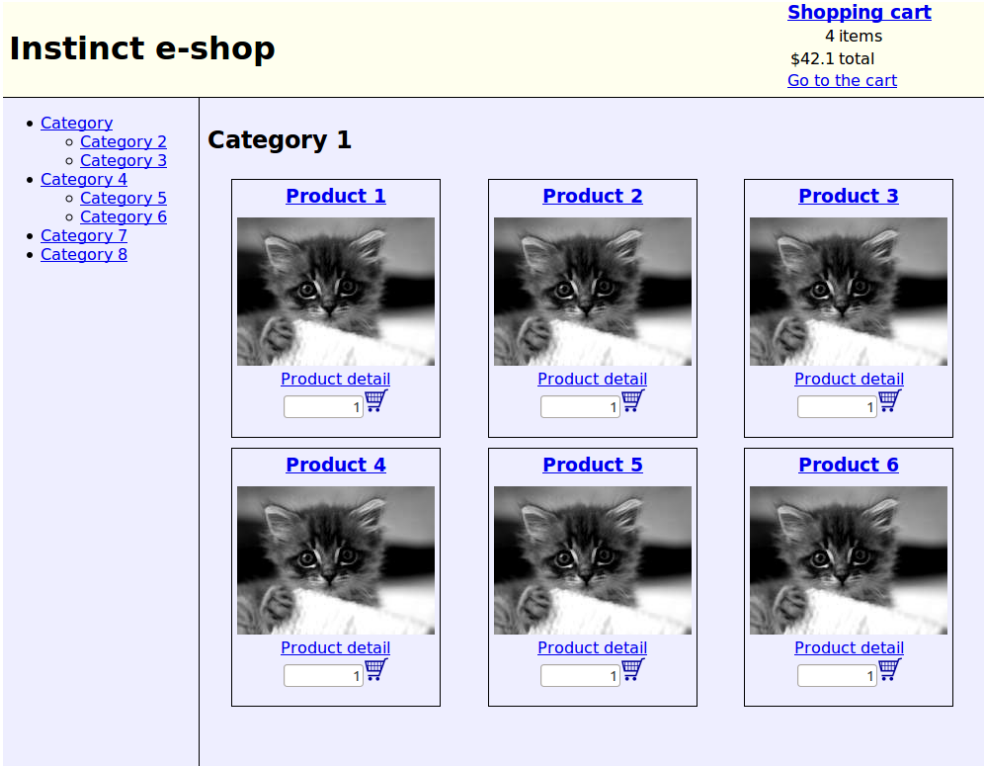

Figure C.4: Generated homepage / category listing screen

Figure C.5: Generated product detail screen

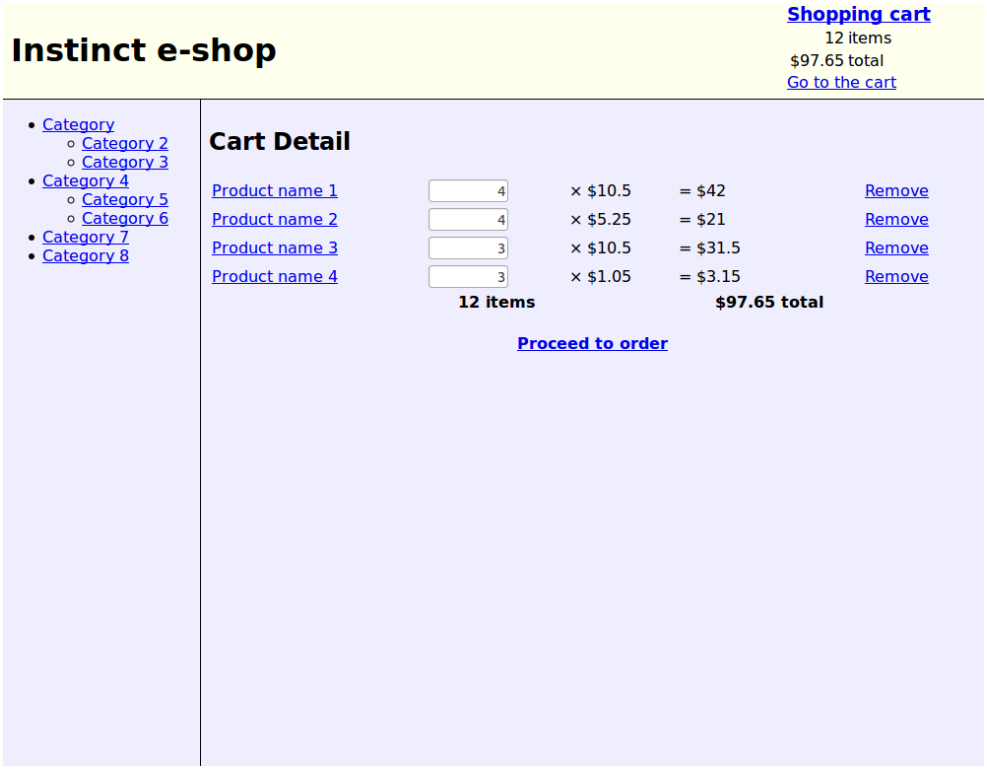

Figure C.6: Generated shopping cart screen

Appendix D

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `instinct-ui` ..... implementation sources
  - `instinct-ui-tg` ... code generation templates and data and generated components
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format

Appendix E

Installation guide

E.1 Requirements

- Node.js v0.12 with npm v2.7
- Requirements for node-gyp library[39]:
  - On Unix:
    - python (v2.7 recommended, v3.x.x is not supported)
    - make
    - A proper C/C++ compiler toolchain, like GCC
  - On Windows:
    - Python (v2.7.3 recommended, v3.x.x is not supported)
    - Windows XP/Vista/7:
      - Microsoft Visual Studio C++ 2010 (Express version works well)
      - For 64-bit builds of node and native modules you will also need the Windows 7 64-bit SDK
      - If the install fails, try uninstalling any C++ 2010 x64&x86 Redistributable that you have installed first.
      - If you get errors that the 64-bit compilers are not installed you may also need the compiler update for the Windows SDK 7.1
    - Windows 7/8:
      - Microsoft Visual Studio C++ 2012/13 for Windows Desktop (Express version works well)

E.2 Installation

1. Copy, download or clone the code from `https://github.com/redwormik/instinct-ui`.
2. Install the dependencies with `npm install`.
3. Edit file `application/settings.local.js` to set up the host and the port. If you don't want to run the API server separately, set the option `RUN_SERVER_WITH_CLIENT` to `true` and add appropriate `API_PATH` setting to `CUSTOM_SETTINGS` (see `application/settings.heroku.js`).
4. Run the application with `npm start`. If the server is set to run separately, run `npm run server` in parallel.