

COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

TWO-WAY DATABINDING OF MODELS AND
VIEWS IN DART

Diploma thesis

2014

Bc. Jakub Uhrík

COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

TWO-WAY DATABINDING OF MODELS AND VIEWS IN DART

Diploma thesis

Study programme: Computer Science

Field of Study: 9.2.1. Computer Science, Informatics

Department: FMFL.KI - Department of Computer Science

Thesis supervisor: RNDr. Tomáš Kulich, PhD.

Bratislava, 2014

Bc. Jakub Uhrík



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jakub Uhrík
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Obojsmerné previazanie dát s pohľadmi v jazyku Dart / *Two-way databinding of models and views in Dart*

Cieľ: Porovnajte rôzne spôsoby obojsmerného previazania modelov s pohľadmi vo webových aplikáciách. Vyberte si prístup navrhnutý Angular-om resp. Model Driven Views, alebo prístup, ktorý používa Facebook React, implementujte tento prístup v jazyku Dart. Zdôvodnite svoje rozhodnutie a odôvodnite, prečo je zvolený prístup pre Dart vhodnejší.

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.
Dátum zadania: 28.10.2013

Dátum schválenia: 29.10.2013

prof. RNDr. Branislav Rován, PhD.
garant študijného programu

.....
študent

.....
vedúci práce



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jakub Uhrík
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Obojsmerné previazanie dát s pohľadmi v jazyku Dart / *Two-way databinding of models and views in Dart*

Cieľ: Porovnajte rôzne spôsoby obojsmerného previazania modelov s pohľadmi vo webových aplikáciách. Vyberte si prístup navrhnutý Angular-om resp. Model Driven Views, alebo prístup, ktorý používa Facebook React, implementujte tento prístup v jazyku Dart. Zdôvodnite svoje rozhodnutie a odôvodnite, prečo je zvolený prístup pre Dart vhodnejší.

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.
Dátum zadania: 28.10.2013

Dátum schválenia: 29.10.2013

prof. RNDr. Branislav Rován, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

I would like to thank my supervisor RNDr. Tomáš Kulich, PhD. for his guidance, support, and encouragement throughout writing this thesis.
Special thanks belong to my family for all their support.

Bc. Jakub Uhrík

Abstract

Abstract in english.

Key words: Databinding, Dart, Facebook React, AngularJS, ...

Abstrakt

Abstrakt v slovenčine.

Kľúčové slová: Databinding, Dart, Facebook React, AngularJS, ...

Contents

Introduction	1
1 Motivation	2
2 Data binding	4
2.1 One way data binding	4
2.2 Two way data binding	5
3 Existing solutions	6
3.1 Template driven	6
3.2 Component driven	9
3.2.1 React	9
3.3 Conclusion	10
4 Our solution	12
4.1 Requirements	13
4.2 Architecture - Tiles	13
4.2.1 Architectural overview	14
4.2.2 Structure	15
4.2.3 Core	16
4.2.4 Life-cycle	20
4.2.5 Rendering	26
4.2.6 Events	29
4.2.7 Injecting	30
4.3 API	31
4.3.1 Component	33
4.3.2 DOM component API	34
4.3.3 Browser specific API	34
4.3.4 Server specific API	35

5	Performance	36
5.1	Initial render	36
5.2	Update of the UI	37
5.3	Continuous resource consumption	38
5.4	Performance optimizations	38
5.4.1	Batched updates	38
5.4.2	Up-down update process	39
6	Benchmarks	40
	Conclusion	41
	Bibliography	42

List of Figures

4.1	The Idea	14
4.2	Virtual DOM	15
4.3	Packages	16
4.4	Core of the library	17
4.5	Life cycle of a Component	21

Introduction

As one of the results of this magister thesis is our new databinding library in dart, which is called **Tiles** . In next text, we will use only **Tiles** to mention *our new databinding library in dart*.

Chapter 1

Motivation

The first question, as always should be, is the motivation of this work. What is the motivation to create another library, that will handle ~~data~~binding-data binding in Dart?

The motivation to create the **tiles** library contains from several aspects, which are not contain in no other Dart library ¹.

Dart as a programming language

Dart language is young programming language with an active development and progress. One of its advantages is optional typing, the build-in compilation to the JavaScript, which enable programming a browser applications, and a Java-like virtual machine, which runs the Dart in the most commonly used operating systems.

It is designed for the web applications with all necessary support for them. As it enable the compilation into the JavaScript and running directly under the OS, it also enable to share a source code between server part of an application and its client application running in the web browser.

Dart also guarantees browser compatibility, what is important for ease of web application development.

Testability

Very important aspect in a building complex application is the testability of the source code.

Because of this, it is essential to use libraries, which enables easy testing and mocking components.

¹We didn't find any suitable library and didn't hear about it

Server side rendering

Server side rendering is very important for user experience and for search engine optimization.

When we have the CPM² which can be used as on the server, so in the clients browser, it is natural to think about a use of the same source code to create an in-browser application, and to render its page on the server.

No templates

This aspect is important from two point of views: the testability and the server side rendering.

From the testability point of view, it is easier to test and mock structures created in only one CPM. If the template is used to create a component of the UI, it is much more difficult to test it and also think about this testing. If this component is only one class in the CPM without dependences on another type of the information, testing is more natural and easier to think about.

From the server side rendering point of view, if we want to work with templates, we need to access them differently when we work in the OS and in the clients browser. If we have the structure fully composed in one CPM, it is easier to compose the same HTML structure on the server as in the browser, then if we have the structure composed by the template and the CPM.

Only one language

This aspect is very related with the previous one. When the application is created fully in one programming language, it is easier for programmers to work with it (they don't have to switch between different CPM).

Also it is easier to compile whole application into the JavaScript, analyze the source code or refactor it.

Reliability

The reliability has significant importance in complex applications. This reliability can be achieved by automatic tests, a robust design and quality development.

When we take into account these aspects, there exists no library, which fulfill all of them.

As there is a need for this kind of a library, we decided to design and create one.

²Computer Programming Language

Chapter 2

Data binding

~~In this chapter we will introduce the area of databinding more deeply than in introduction~~ Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data between the different layers of the application.^[wika] When we talk about the data binding in the UI of the web application: Data binding is the process that establishes a connection between the application UI (User Interface) and business logic.^[Wikb] The application logic of the web application is represented by a data model (JavaScript/Dart used on the website) and the UI by the HTML view of the data.

The data binding is widely used in frameworks using MVC patterns. The most common way to implement the data binding is by using a template engine. The template engine takes the template, merge it with data and produce the view of the UI.

The model(data) can be connected with the view in a one direction, $model \mapsto view$, or in both directions, $model \mapsto view$ and $view \mapsto model$. The single way $view \mapsto model$ is not used.

2.1 One way data binding

~~Discuss one way databinding~~ The one way data binding implements the $model \mapsto view$ direction.

The server side rendering is an example, where the one direction data binding is the perfect way to map data into the view. From the server side rendering point of view, the other direction of the data binding doesn't even exist.

~~Discuss two way databinding~~ The more interesting usage of the one way binding direction is in the browser application. The one way data binding is used to reflect the model of the application into the UI view. This reflection is maintained continuously

and keep the UI in the sync with the model. Therefore each change of the data model is reflected into the view.

2.2 Two way data binding

Two way data binding extends the one way by implementing the second direction: $view \mapsto model$.

One direction, when the model is updated, the view is modified to reflect the data. In the second direction, when the view is changed (e.g. value in the input field), the model is update to represent actual view. This way, the model and the view are continually synchronized. The model is the *single-source-of-truth* for the application state[Goo].

The most common understanding of the two way data binding is the reflection of the change of input elements (input, select, text-area etc.) into the data model.

There are additional informations stored in the view, which can be reflected into the model. For example the order of items in a list, a position in the DOM or any other information stored by the structure of the view, which can be changed e.g. by drag-and-drop actions. In the majority of two way data binding implementations, this type of the information isn't reflected.

As the information from input elements can be reflected to the model by event listeners, we decided to not implement the two way data binding in the first phase of the **Tiles** library. We plan to implement it later, by using the observable pattern.

Chapter 3

Existing solutions

When we think about the building of the user interfaces, we can think about the building them from components. The component is a part of the UI, which has a functionality, own look and maybe some interaction.

The HTML is a basic component structure. Every element is a component, all elements are composed into tree structure. Elements have some functionality, own look(e.g. image) and some of them have interactions(e.g. input).

So when a library want to bind the data with a view, it basically bind the data to a component.

There are different approaches of the creating these components and the connection between them and a data. Components can be created directly by a programming language (Component driven), or by using a template engine, which create components based on the template, which describe component structure, in the most cases by HTML-like syntax(Template driven).

These two approaches do the same thing, create structure of the components, different way.

3.1 Template driven

Template driven approach is, as the name predicts, based on the usage of a template engine. Template engines take a template and the data and create a component structure, which is reflected into the HTML representation of passed data in the form of the template. They can be considered as a function $t : \mathbb{D} \mapsto \mathbb{H}$, where \mathbb{D} is a set of all possible data and \mathbb{H} is a context-free language of valid HTML.

An easy example of a template, for example using *handlebars.js* can look like this one (from *handlebars.js* website):

```
<div class="entry">
```



```

<h1>{{title}}</h1>
<div class="body">
    {{body}}
</div>
</div>

```

When programmer want to use this template, he should create data object, which minimal version in JSON format is in next example:

```

{
  "title": "Some title",
  "body": "This is the content of the page"
}

```

When template is filled by this data, following HTML will be produced

```

<div class="entry">
  <h1>Some title</h1>
  <div class="body">
    This is the content of the page
  </div>
</div>

```

Most of template engines also offer logic markup, which add possibility of the better control of a composed structure. This is highly usable when programmer want to create more complex structures based on the data. The typical example of this structure is the `` list generated from the array of items to render.

This "in template" logic has on one hand some advantages, on the other hand, the HTML syntax was not created to represent a logic, but an information. Because of this, more complex templates witch not so trivial logic in it becomes hard to read and understand.

Easy use of the logic in the template is shown on the next example:

```

<h1>Comments</h1>

<div id="comments">
  {{#each comments}}
  <div class="entry">
    {{#if author}}
    <h1>{{firstName}} {{lastName}}</h1>
    {{else}}

```

```

    <h1>Unknown Author</h1>
  {{/if}}
<div>{{body}}</div>
</div>
{{/each}}
</div>

```

The template driven view are highly used because of the syntax similarity between the template and the resulting HTML. It is easy to convert the HTML produced by a graphic designer to the template used in the source code. Also programmers used to work in the HTML more easily write templates then some other representation of the component structure.

Different libraries work with templates in a different way. Some of them really parse the input template as a string, recognize component tree in it and work with the template that way. Others uses in-browser HTML parser to parse the template and then fill it with the data. This approach, because of its usage of tools accessible only from the browser, is more difficult to render on the server.

Templates are mostly used two different ways:

Template used as View

Template is used to render HTML structure into some element. Functionality of the HTML structure is then realized separately and attached to it. This is used for example in the CanJS.

Template used as Component

The other (and more modern) use of template is to represent one component with attached functionality, which can be represented later as custom HTML element in other templates. In this template, other custom components can be created by using their custom HTML element representation. It is not necessary to create them separately in the code of an application.

This approach is used e.g. in Polymer project, which work with so called "shadow DOM" which use similar concept.

Table 3.1 Comparison of template driven libraries compares some of existing solutions which are standalone libraries or MVC frameworks. The aspects of the comparison are a natural rendering in the browser and on the server and if the library is a standalone UI library, or is a part of the more complex MVC framework. We don't compare a possibility to render the view on the server other then the natural way, because it is always possible to render it by usage of tools like the *PhantomJS*.

Solution	Language	Standalone	In browser	On server
handlebars	JavaScript	yes	yes	yes
mustache	JavaScript, python...	yes	yes	yes
dust	JavaScript	yes	yes	yes
AngularJS	JavaScript	no	yes	no
meteor	JavaScript	no	yes	no
EmberJS	JavaScript	no	yes	yes
Derby	JavaScript	no	yes	yes
Polymer	JavaScript	yes	yes	no
Polymer.dart	Dart	yes	yes	not now

Table 3.1: Comparison of template driven libraries

3.2 Component driven

Component driven views, in opposite to the template driven, don't use any additional type of data like templates. Components are created by the same programming language as the functionality and are composed into the tree structure which is mapped into the DOM.

When the tree of components (we will call it "Virtual DOM" later) is constructed, it is rendered to the DOM by the depth-first search of the component tree. When components and HTML elements are connected by stored associations, every change in the component structure can be applied to the DOM tree.

In addition, if we have the tree of components, we can easily, by the similar depth-first search, create the markup string representing the HTML markup of the component tree. This enable the rendering of the whole component tree on the server without use of browser-specific features.

An example of the component driven UI library is the JavaScript library *React* created by the Facebook . *React* is standalone UI library which enable native rendering of the component structure as in the browser, so on the server.

We decided to use a similar approach to *React* library, so we briefly describe it.

3.2.1 React

Lots of people use *React* as the V in MVC.[\[col\]](#)

React is JavaScript UI library from Facebook . Its main concept is to pack parts of the web application into reusable components, which are represented as object in JavaScript.

This components can be mounted into elements in DOM, for now, we will call it *mount root*. This will create *virtual DOM* "mounted" to *mount root*. This virtual DOM is then reflexed into the real DOM under the *mount root*.

React uses a virtual DOM diff implementation for ultra-high performance. It can also render on the server using Node.js — no heavy browser DOM required.[\[col\]](#)

Components are organized to the virtual DOM tree, where a data flows from the root component to leaves. This data flow is implemented by the props of the component, which are read-only. Component have also a state, which should be stored in the state attribute and updated by methods `setState` and `replaceState`. The state shouldn't be updated directly to preserve the invariant, that the real DOM always represents the actual state of the virtual one.

Component describe the structure under it by its method `render`, which should return one instance of a component, which will be added as a child of this component. The render function also add props to the child component. This realize the data flow in "down" direction. The render also add children to the child component, which is the way, how to create a spreading tree, not just a line. The child component have read access to passed children and can reuse them in the `render` method or ignore them.

The *React* offer own events system with synthetic event bubbling. This enable programmer to listen to events independently from browser. The *React* manage the browser compatibility.

Components can listen to events on DOM components (internal *React* components, representing DOM elements). They are attached trough props by the event listeners syntax(`onChange`, `onClick` etc.).

State change(by mentioned methods) trigger redrawing of virtual DOM.

React implements component life-cycle methods, which notify the component about its actual state of living (just mounted, just updated, before unmount, etc.). They are the superset of life-cycle methods implemented in the tiles library.

For more information about *React*, it's architecture and API reader can go to the website of the *React* project [\[http://facebook.github.io/react/\]](http://facebook.github.io/react/).

3.3 Conclusion

We decided to use **Component driven** views and databinding, because it is not dependent on the template engine, whole source code can be written in the same CPL with all advantages gained by that and naturally easier thinking about testing and mocking.

Our solution is based on the idea of the *React* library.

As we decided to work in the Dart language, we don't have to implement a browser compatibility, synthetic events, mixins, etc.

The architecture of the tiles library will be described in the [chapter 4 Our solution](#).

Chapter 4

Our solution

The first attempt was to create a wrapper of the *React* library into the Dart language. This wrapper was successfully created, tested and also used in an independent commercial project.

The problem occurs in the performance of the wrapper, where the bottleneck of the speed was the communication between *React* created in the JavaScript and the wrapper in the Dart language. This bottleneck can be reduced by some adjustment and some Dart hacks, but it was still a bottleneck.

That's why we decided to build our own library called **Tiles**. Most of the performance benchmarks of the **Tiles** library later in this work will be compared with the *React* wrapper.

As we told in the previous part of the work, we decided to take inspiration from the Facebook *React* library, mainly in the API of the library, which is component based, with some differences in architecture.

We don't have to implement some of the additional features of the *React* library, because of the nature of the Dart language.

- Synthetic events (*Dart unified events*)
- Mixins (*Dart support native mixins as a part of a language*)
- Props type checking (*Dart is optional-typed language*)
- Get default props and initial state (*Dart work with classes which have constructors*)
- Changed class name (*Map in dart use string, so string "class" is no more reserved word*)
- Test utilities (*Dart has own unittest library and we work with classes and with native events, it is easily tested*)

In this next sections of this chapter we will introduce and deeply describe our Dart library **Tiles** .

4.1 Requirements

When we designed the **Tiles** library, we take into account requirements derived from the motivation of this work:

Rendering in both environments, the browser and the server

One of the main advantages of **Tiles** library is a possibility to render the same content, with the same code as on the server, so in the browser. This resolved into the package structure and several architectural decisions.

No template usage

To achieve a possibility to render content in both environments, easy testing and mocking, we decided not to use templates.

Easy to use API

The solution can be very interesting and powerful, but if it don't offer a reasonably easy to use API, almost no one will use it.

React -like API

Because we use a similar concept as the JavaScript *React* library, which is widely used and known, if we offer similar API, more people will quickly get used to it.

Performance

We want to offer the useful library, and if want someone to use it, we need to offer a good performance in the competition of Dart and JavaScript UI libraries.

How we fulfilled these requirements is in detail described in next sections of this chapter.

4.2 Architecture - Tiles

The **Tiles** library implements a component driven databinding. It enable declarative definition of the user interface by the **Component** class. The component define the structure of the virtual DOM tree, which is rendered to the real DOM. Updates in the virtual DOM are reflected into the real one with usage of the smallest difference between old and the new structure.

The composition of the virtual DOM is used in the **Tiles** to create the system of synthetic event bubbling, which offer the programmer a simple way of listening to events in the DOM from a component perspective.

The architecture of the **Tiles** library offer the possibility to think about UI as a set of the functional components, instead of HTML elements and a Dart/JavaScript application manipulating with them.

4.2.1 Architectural overview

Our high level idea is based on the Facebook *React* library attitude. We created api, whose main class is **Component**, which represents construct very similar to *React* 's **Component**. This component is mounted to an element, where it renders itself. This relationship is described on [Figure 4.1 The Idea](#).

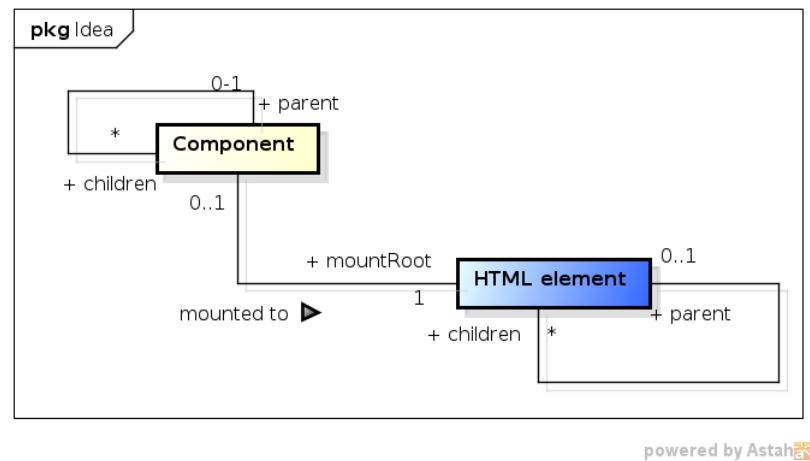


Figure 4.1: The Idea

These components are placed into tree structure, which represents **Virtual DOM**, which is then translated to the real DOM of client's browser or to the markup rendered by server application.

There can be the event listeners attached to these components. The **events**¹ are then bubbled through a virtual DOM, instead real one. By this there can be the listener attached to a custom component, which doesn't have element representing it in a real DOM.

As we work in Dart language, it is natural to try to reuse the most of code on the both, client and server side. The next important part of architecture is **server-side rendering**.

It is very important for SEO purposes and smooth user experience.

¹We work at Dart, which create browser compatibility for us, so we don't have to create synthetic events like *React* .

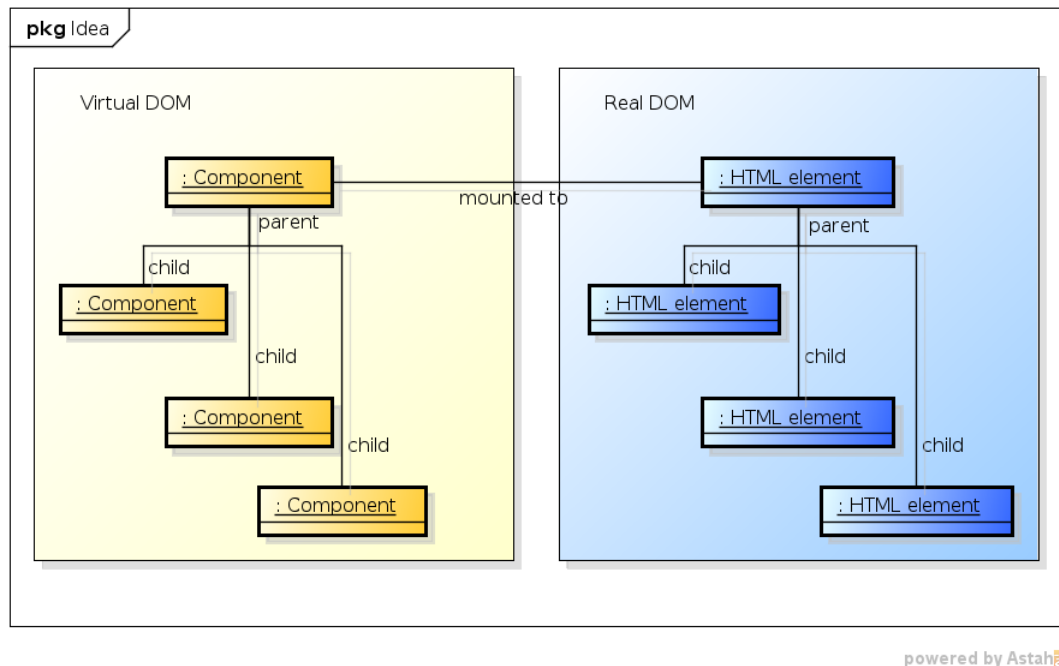


Figure 4.2: Virtual DOM

4.2.2 Structure

We split our library into 3 partially dependent packages.

Tiles

Tiles package creates the core component's of library, focused to create and maintain virtual DOM and provide API for programmer. This package should be included by programmer in the files, where he defines custom components. These components then can be used on both, server and browser sides.

Tiles Browser

This package is used for mounting components to the HTML elements. It maintains relationships between elements and components, simulates events bubbling and keeps real DOM in sync with virtual one.

Tiles Server

Tiles Server package maintains server-side rendering. It offers an API to render component structure to string with markup based on DOM components.

Based on the mentioned packages structure, it is quiet obvious what are the dependences between these packages. **Tiles** package is independent, and both of **Tiles Browser** and **Tiles Server** are dependent on **Tiles** package. These dependences are shown on [Figure 4.3 Packages](#).

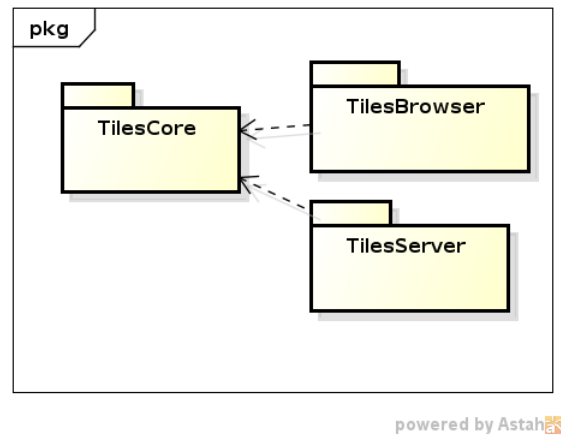


Figure 4.3: Packages

4.2.3 Core

There are 4 main classes in the core of the library.

Component represents closed block of user interface, that should be rendered in application.

Node is a vertex in a tree of virtual DOM. It contains an instance of [Component], which represents the type of this **Node**.

ComponentDescription is self-explanatory. It is returned from the component to describe it's children. The principles are described later at this document.

NodeChange represents one change in a virtual DOM, which should be applied into the real DOM. This way, we are able to achieve minimal changing of the real DOM. Types of change are: **CREATED**, **UPDATED**, **MOVED**, **DELETED**

In contrast with Facebook *React* **Component**, our component provides only an API to a programmer. This class is the main class for a programmer using our library. He doesn't need to use any other class created by our library. Just some methods.

We also have an inspiration from *React* with idea of **virtual DOM**. Vertices of an virtual DOM are represented by the class **Node** instead of the class **Component** to separate the functionality.

Each node contain an instance of **Component**. The node represents the component in a virtual DOM.

The diagram of relationships is shown on [Figure 4.4 Core of the library](#). In the next chapters we describe the main classes more in details.

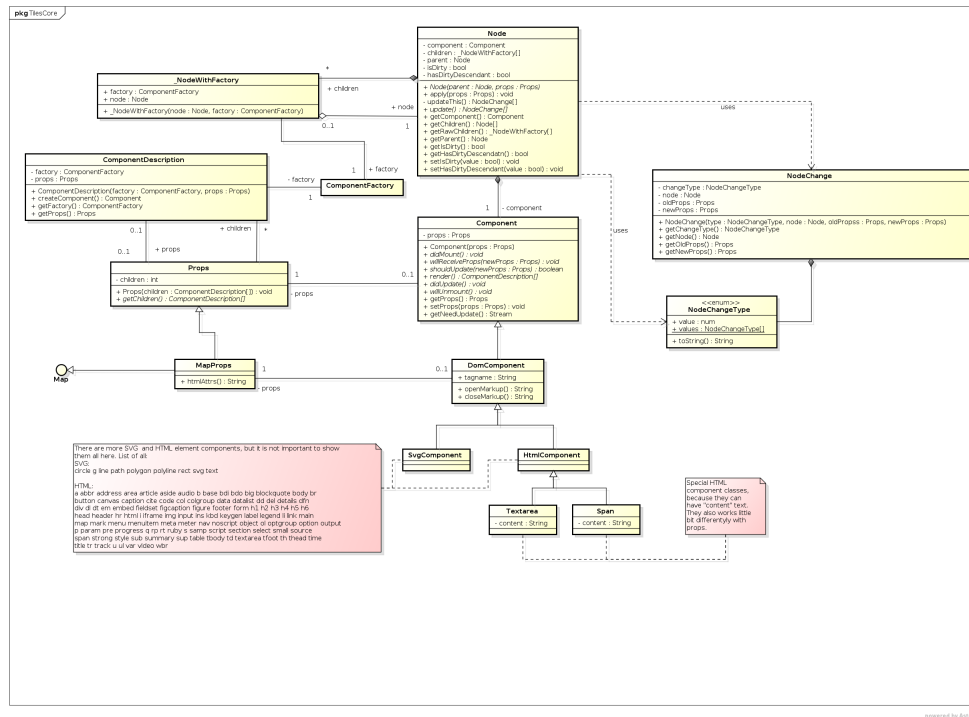


Figure 4.4: Core of the library

Component

The Component is the main building brick of application(library). It offers api to the programmer with life-cycles, props etc.

Component is a class, which represents functionality of certain part of UI² in an application. It is created with some props and children acting as parameters of a function. Main purpose of the component is to create a structure below³ it, add the event listeners and update itself based on event listener input.

The main method of the Component is `List<ComponentDescription> render()`. By this method component describes it's substructure. It will return list of children of this component, represented by instance of the class `ComponentDescription`. Node, which owns this component (and called it's render method) will manage the rest. Basically, it will return the message like *"This is how I should look like"*.

Second important method is `void redraw()`, which trigger redraw of the component. This redraw will be executed on the next animation frame.

Redraw is powered by `needUpdate` stream offered by Component, which is automatically created in default constructor of class Component, so it is very important, to call superclass constructor in each custom component class.

ComponentDescriptionFactory registerComponent(ComponentFactory

²UI = User Interface

³From the virtual DOM tree point of view

`factory`) is additional method helping programmer for easier `ComponentDescription` creation. Factories are described late in [section 4.3 API](#)

DomComponent

`DomComponent` is a subclass of class `Component`. This is specialized class, which represents HTML elements in the component structure.

It has `props` saved as `Map`, because HTML element has attributes saved in `Map`. `render` method returns `children` member variable and `svg` and `pair` flags.

Specific HTML elements are created based on different `ComponentFactory` and `ComponentDescriptionFactory`. `ComponentDescriptionFactory` is used to easily create `ComponentDescriptions` of `DomComponent` in a custom component render method.

ComponentDescription

`ComponentDescription` is a description of the component. It describes which type of the component should be rendered by using which parameters.

For this purpose, it needs 4 types of information:

- **Type of the component**

To create instance of a component, we need to know, what type (class) of the component it should be. This information is represented by `ComponentFactory`, which is function with 2 parameters, `props` and `children`, which returns instance of a subclass of a `Component`.

- **Properties**

Data which should be passed to the factory. This data are used as a properties of the component.

- **Children**

Children of described component. This is useful mainly when programmer wants to render more complex structure of `DOMComponents`.

- **Key**

Key is an identifier of a child. It is used to recognize reordering of children of the component. When components `render` method returns list of descriptions, keys they contain are recognized and matched with keys stored in virtual DOM.

If there is a match in key of the child in different position, child is only moved an updated. If there is no match in key, default process follows.

Description is once created with all the parameters and then these parameters can't be changed. All these parameters is set up by constructor.

`ComponentDescription` has one important method, which is `Component.createComponent()`, which creates `Component` instance with props and children from the description.

Node

`Node` is the most important and complex class in the library. It provides following functionality:

- creates virtual DOM tree, maintains creating and updating of the tree based on results of component's `render` method,
- listens to component's `needUpdate` stream and marks self as "*dirty*" when it's component need update,
- and handles updating process that is rearranging children of the `Node`.

The node is also a vertex of the virtual DOM. It store children as a list of children. To use all possible optimization, node contain a `ComponentFactory` of the contained component, which is used when the virtual DOM is updated. It also contain `key` which is used to recognize changed position of the same child.

Node has two important flags: `isDirty` and `hasDirtyDescendant`. These flags represent information, whether the node, or its descendants, needs to be redrawn. If `isDirty` is true, the node needs to be updated, because component of this node called `redraw` method. If `hasDirtyDescendant` is true, there exist a descendant of this node, which wants to be updated. When `hasDirtyDescendant` is true and `isDirty` is false, the node doesn't have to update itself, it is enough to call update on child nodes.

Method `update(List<NodeChange> changes, boolean force: false)` is executing the update process. It take 2 named arguments:

`List<NodeChange> changes` is used to be filled by changes generated by the update process,

`bool force: false` is a flag signaling if to update despite that node is not dirty.

The `update` method is used mostly in the browser part of an library, where it offers a possibility to get changes in the virtual DOM, which should be used to update the real one.

A methods logic consists of several main steps.

1. check, if update is needed by flags `isDirty`, `hasDirtyDescendant` or `force`, if no, exit,
2. if component of this node needs to update (`isDirty == true`) or `force == true`, update this node with rearrangement of children,
3. if any changes was generated, add them into the `changes` list
4. set this node as not dirty and not have dirty descendant.

The algorithm of rearrangement of children by calling `render` method of this component and adapting node's children to returned descriptions is fully documented in the source code related to this work: https://github.com/cleandart/tiles/blob/master/lib/src/core/node_update_children.dart

NodeChange

`NodeChange` takes place as a record of a change in the virtual DOM. It is used to mirror changes in the virtual DOM with the real DOM.

When some node in the virtual DOM is updated by method `update`, the list of changes is collected. This list is subsequently processed by browser part of a library, which mirrors changes to the real DOM.

`NodeChange` class has no methods (except constructor) and acts as a data chunk dedicated just for it's purpose. It contains node, type of change, old and next properties.

Type is stored as an instance of `NodeChangeType` enumerable and can be one of: `CREATED`, `UPDATED`, `MOVED` and `DELETED`. When type is `UPDATED`, old props and new props take effect.

4.2.4 Life-cycle

Every instance of `Component` has its own life-cycle. As every object, first it is created. Subsequently, it is mounted and rendered into the virtual DOM, and then in to real DOM.

When a "higher" node wants to update the node of the component, the component will first receive props, subsequently is asked if want to update, if yes, it is asked for actual description of its children.

Sometimes the component wants to update itself (e.g. because event occurs). It calls `redraw`, then, it will be asked if really should update, and if yes, it is rendered and updated.

At the end of component's life in the real DOM, component should be notified about that.

The whole life-cycle is shown on the [Figure 4.5 Life cycle of a Component](#).

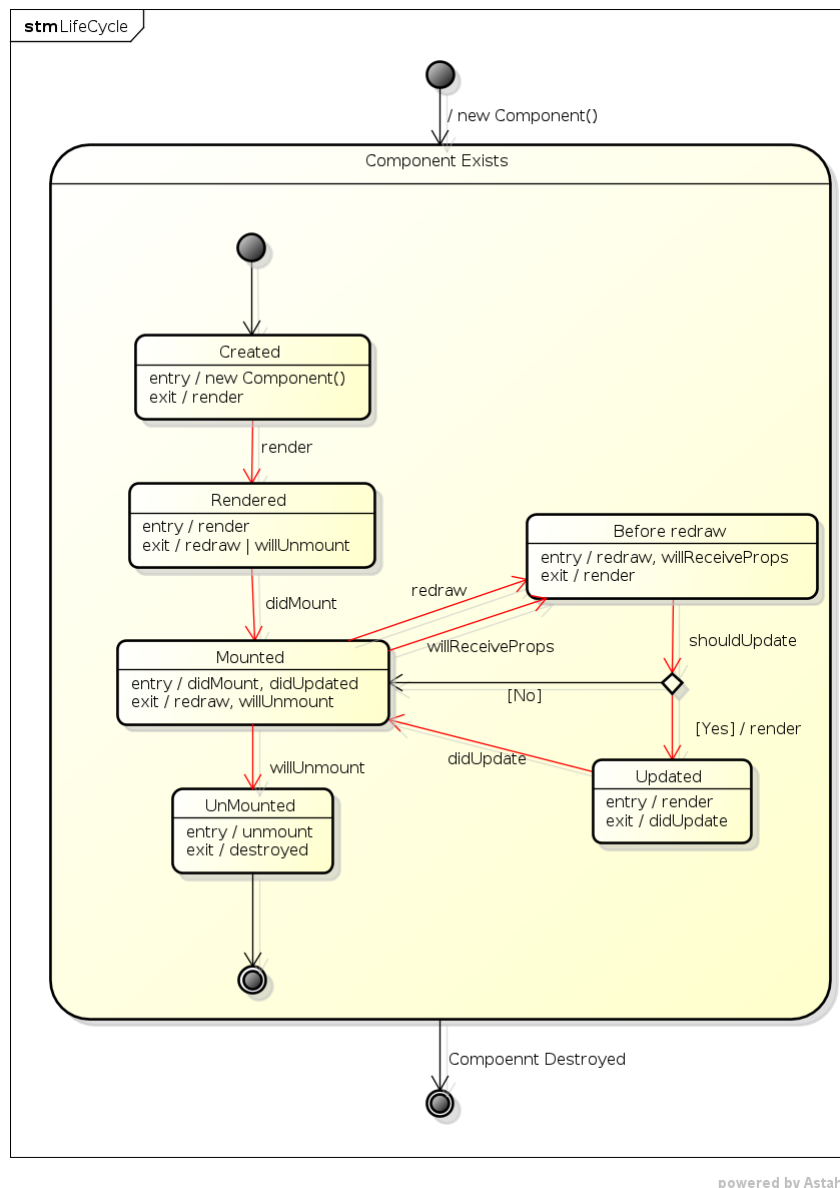


Figure 4.5: Life cycle of a Component

Create

The create part of life-cycle is implemented by constructor of **Component**. It will receive props and optionally children as arguments and it should prepare the whole state of object to live.

An trivial example of constructor of **Component** is displayed below.

```
class MyComponent extends Component {
```

```
MyComponent(props, [children]): super(props, children) {}
}
```

Constructor of an example only calls constructor of superclass **Component**.

Example of more complex constructor should be e.g. the **Todo** component example:

```
class MyTodoComponent extends Component {
  Todo todo;
  MySearchComponent(props, [children]): super(props, children) {
    if (props != null && props.todo is Todo) {
      this.todo = props.todo
    } else {
      this.todo = new Todo();
    }
  }
}

// ...

}
```

Did mount

When the component is mounted to the real DOM, user of the library should be notified about this event. It is done by triggering the **Did mount** life-cycle event implemented the method `didMount`.

This is the place in time, where the component *start to live its life* with the connection to the real DOM. This is the correct place to initialize for example the timers, stream listeners etc.

Our **MyTodoComponent** example component should listen for a change of the todo on the server, and if it was changed, we can redraw the component.

```
class MyTodoComponent extends Component {
  Todo todo;
  StreamSubscription subscription;

  // ...

  didMount() {
    this.subscription = this.todo.changedOnServer.listen((change) {
```



```

        this.redraw();
    });
}

// ...

}

```

Will receive props

When the component is updated by "higher" ancestor, it will receive new props. A user of the library can need the possibility to compare these props with the old one and perform needed changes.

This is the correct place for subsequent life-cycle event *Will receive props* implemented by the method `willReceiveProps`.

The example of `willReceiveProps` in `MyTodoComponent` should compare the `todo` of the old and the new props and if they are not equal, update change listener.

```

class MyTodoComponent extends Component {
    Todo todo;
    StreamSubscription subscription;

    // ...

    willReceiveProps(dynamic newProps) {
        if (this.todo != newProps.todo) {
            this.subscription.cancel();
            this.subscription = newProps.todo.changedOnServer.listen((change) {
                this.redraw();
            });
        }
    }

    // ...
}

```

Should update

An optimization of the performance of an application can be done by rejecting "redraw" of the component. To reject this "redraw", component should be asked, if the redraw is needed.

This is implemented by the `shouldUpdate` method, which returns true if the component want to be redrawn.

By default `shouldUpdate` returns true, what resolves to always updating of custom component, which doesn't implement this method.

In a basic scenario this method recognize, if the component will be rendered differently with the new props. If not, it return false, else it return true.

Example in `MyTodoComponent`:

```
class MyTodoComponent extends Component {
  Todo todo;
  StreamSubscription subscription;

  // ...

  shouldUpdate (newProps, oldProps) {
    if (newProps.todo == oldProps.todo) {
      return false;
    }
    return true;
  }

  // ...

}
```

Render

Render is the main part of the `Component`.

It is implemented by the method `render`. It should return array of component descriptions which should be considered as *"this is how this component should look like"*.

For example, in our `MyTodoComponent` render will return `<div>` which contains title and description of `todo`.

```

class MyTodoComponent extends Component {
  Todo todo;
  StreamSubscription subscription;

  // ...

  render () {
    return div ({"class": "todo"}, [
      h2 ({}, todo.title),
      p ({}, todo.description)
    ]);
  }

  // ...
}

```

Did update

When the life-cycle method `didUpdate`, by which is implemented the life-cycle *"Did update"* event, is triggered, the component is notified, that it is mounted and there exist HTML elements in the DOM for each `DomComponent` descendant.

Will unmount

The *"Will unmount"* event is implemented by the method `willUnmount`. It is called right before the component is removed from the virtual DOM and therefore from the real one. This is the correct place to stop all timers and listeners.

```

class MyTodoComponent extends Component {
  Todo todo;
  StreamSubscription subscription;

  // ...

  willUnmount () {
    subscription.cancel();
  }
}

```

```
// ...  
  
}
```

4.2.5 Rendering

The main target of the **Tiles** library is the rendering of a content. By the **Component** and the **Node** we can create the virtual DOM tree, which should be reflected into the real one, or into the HTML markup string.

As we described earlier in the [subsection 4.2.2 Structure](#), these rendering types are separated to separate packages from the core package and are independent from each other.

Server side

On the server, we don't have DOM elements available, therefore we want to render our virtual DOM structure into the string representing HTML markup of the virtual and also the real DOM.

The markup string constructed on the server can be reused in the browser to smooth user experience.

Our target, render virtual DOM into the markup string is not so complicated. From the programmers point of the view, he will use a **ComponentDescription** to describe the component to be rendered. Our server-side package of the library will get the description and construct the node tree structure, with node containing described component in the root. The node tree represents the created virtual DOM.

From the virtual DOM we can render corresponding markup string by the depth-first search of it's tree. The markup string is created recursively for every subtree

starting from a node by this algorithm:

Data: The node in virtual DOM tree

Result: A string with the markup of the virtual DOM subtree with a root in the node

```

if the component of the node is DomComponent then
  if the component is not pair then
    write the markup into the result with attributes from the props and a tag
    name from the component;
  else
    write the open markup into the result with attributes from the props and
    a tag name from the component;
    write the markup for all children recursively into the result;
    write the close markup into the result;
  end
else
  write the markup for all children recursively into the result;
end

```

Algorithm 1: Write node into the markup string.

In browser

The rendering in the browser is more difficult than the rendering into the markup string. It is possible to use the same *render to the markup string* method and add this markup to the DOM. This will create DOM structure representing the virtual DOM, but doesn't create connections between nodes in the virtual DOM and HTML elements in the real one. This connections is necessary when the virtual DOM is updated. Therefore the browser side rendering will work in a different way.

Initial mount The first thing, the user of the library need to do, is to mount the component into the HTML element(*mount root*). For this purpose the **ComponentDescription** describing the component is used. The mount process consist from creation of the component from the description, placing it into the node and first render of the virtual DOM with the root in the node.

The virtual DOM is subsequently reflected into the real DOM created under the *mount root*.

Suppose that the *mount root* have no children (the case of the *mount root* with children is described in the [subsection 4.2.7 Injecting](#), the node mount process is described in the [algorithm 2](#) **Initial mount**:

Data: The node in the virtual DOM and a HTML element, to mount the node to (mountRoot)

Result: Mounted the node into the element

if *the component of the node is DomComponent* **then**

 create an element representing the component;

 add the created element to the mountRoot;

 save relations between the created element and the node;

if *the component is a pair component* **then**

for *a child in children of the node* **do**

 mount the child node into the newly created element as a mountRoot;

end

end

else

if *the component is a text component* **then**

 create a HTML text node with the text from the component;

 add the text node to the mountRoot;

 save relations between the created text node and the node;

else

for *a child in children of the node* **do**

 mount the child node into the mountRoot;

end

end

end

Algorithm 2: Mount the component into the real DOM.

The algorithm is recursive and skips custom components. It creates relationships between created elements and nodes. These relations is used to easy reflect the change in the virtual DOM into the real one when the virtual DOM is changed.

The DOM structure created by the [algorithm 2 Initial mount](#) reflects the virtual DOM structure with removed nodes with the custom component.

Update As time goes, the situation, that the virtual DOM need to be updated, occurs. The need of the update is realized by the method **redraw** of the component, which cause marking the node as dirty. The framework recognize this situation and perform update of this node and the structure under it. The update produces a list of changes in the virtual DOM needed to be applied to the real DOM.

Updates are processed by their type. For every type the information about the HTML element representing a node is needed. This information is stored in the relation $\text{Node} \rightarrow \text{Element}$, created in the mount of the node. It is stored by **Map<Node,**

Element>.

What happened when we want to apply the node change into the real DOM structure differs by the type of the change:

CREATED

A newly created node is mounted into the DOM. The adjusted [algorithm 2 Initial mount](#) is used for this purpose.

UPDATED

If the node with a DOM component in it was updated, the element, related with it, is updated with the props of the node. If the node with a custom component was updated, no action is needed, because this node is not directly reflected to the DOM.

MOVED

The DOM representation of the moved node is moved to the new position.

DELETED

The DOM representation of the removed node is removed from the real DOM.

4.2.6 Events

In the library which create own virtual DOM structure, it is natural to think about event in this virtual DOM. As in the real one, it is natural to create event bubbling trough virtual DOM. This synthetic event bubbling represents real event bubbling of the event trough the real DOM elements associated with nodes in the virtual DOM.

Synthetic event bubbling is useful for the user of the library to catch events in the DOM and react on them by update of the state and triggering of the redraw of the component, if needed.

The **Tiles** library listen to DOM events on *mount root* of the virtual DOM. When an event occur, the **Tiles** catch it on it's bubbling route up to he **body** element, recognize, which node represents the target element of the event and start synthetic bubbling of the event from that node.

To store the relation **Target Element** \rightarrow **Node** is used **Map<Element, Node>**.

The **Tiles** library listen to every type of the event only once in the *mount root* of the virtual DOM for each event listened by components in it.

It is important from the performance point of view, because we can add only one event listener for each event type in the whole virtual DOM. We will discuss this later in [chapter 5 Performance](#).

Synthetic event bubbling through the virtual DOM enable to listen to the bubbled event even on custom component.

Synthetic bubbling

When the component is mounting, we store the relation between the HTML element, and the node, which contains this component. Then we check a presence of the event listeners in the descriptions of the component. We add event listener for every type of the event present in the virtual DOM to the *mount root* HTML element. The event listener in the *mount root* is only one for each event type in the virtual DOM.

When the event type listened by *mount root* occurs in the DOM subtree representing our virtual DOM, it will bubble up to the *mount root*, where is caught by the event listener created by the **Tiles** library. The **Tiles** library will recognize the target HTML element of the caught event and simulate event bubbling from the node representing the target element in the virtual DOM.

The synthetic bubbling process starts from the target node and check event listeners associated with it. If a listener for a type of the current event is presented, it is called with the event and the component as arguments.

The component is passed because the listener is not created by this component itself, but by the component above it, which should be informed, on which component the listener was called.

There is no official way of getting an information if the `stopPropagation` was called on the event. Therefore the listener, which want to stop a propagation, should return the `false` boolean. If the listener returns something else, bubbling continues with the parent of the current node. When the root node is achieved, the bubbling stops.

4.2.7 Injecting

The **Tiles** library implement a built in server side rendering.

It is important as for the SEO, so for the user experience. User don't have to wait until the source code of the application is loaded into the browser and see the content of the page. If our browser package replaces the structure generated on server with a new DOM structure generated from virtual DOM, the part of the page is re-drawn from scratch, what cause disappearance of the content for the small amount of the time.

To prevent this behavior, we created an injecting system, which will "inject" the existing DOM structure and rebuild it to represent the virtual DOM.

When the `ComponentDescription` is mounting, basic implementation can erase

whole content of element to which is the description mounted to. Instead, we will reuse the existing structure by iterating through the virtual DOM and reusing every element, which match the virtual DOM. The **Tiles** library iterate through the virtual DOM and the real DOM simultaneously and compare them.

When the iteration through the virtual DOM process a node with a DOM component and the HTML element, their tag names are compared. If tag names match, the HTML element is associated with the node, element's attributes are adapted to the props of the node's component and is used to mount/inject children of the node recursively.

When the tag name don't match, a new HTML element is inserted before the currently iterated element and paired with the node. Subsequent DOM component at the same layer of the DOM component tree ⁴ can reuse this not matching element.

When injecting of children finishes for a node with a DOM component in it, not used(injected) child HTML elements of the associated element are removed.

This implementation of injecting enable a full reuse of the DOM structure created on the server. It also create relations important for future updates of the virtual DOM.

Moreover, if there exist a similar structure, not generated from same components on the server, our library will reuse as much as possible of it.

4.3 API

One product of this work is an open source UI library called **Tiles**. To use this library, user need to know the application programming interface(API) of it.

In the **Tiles** library, the user interface consist of components, represented by the class **Component**. The **Component** class is used to create a functional logic of a part of the UI.

To use it, mount it and add it to the other component as a child is necessary to create a **ComponentFactory** and a **ComponentDescriptionFactory**, which creates a **ComponentDescription**.

The **ComponentFactory** is a function with two positional optional arguments: **props** and **children**. It return an instance of the **Component**. The **ComponentFactory** is used as a comparable type of the component. The easiest and most common way to create **ComponentFactory** is shown in next example:

```
class MyComponent extends Component {
  MyComponent(props, [children]): super(props, children) {}
```

⁴ By the DOM component tree we mean a derived tree from the virtual DOM tree, which can be constructed by removing nodes with a custom component and connecting nodes with DOM components to the closest ancestor in the virtual DOM tree with a DOM component in it.

```
}
```

```
ComponentFactory factory =  
  ([dynamic props, dynamic children]) => new MyComponent(props, children);
```

ComponentDescription is a description of a component. It is used as argument to mounting functions and as a result of the **render** method. Most of the time is used indirectly by usage of **ComponentDescriptionFactory**.

ComponentDescriptionFactory is a function, which create a **ComponentDescription**. It has 7 optional named arguments:

props passed to the component factory,

children also passed to the component factory,

key used to optimize reorder of children and

listeners used to attach event listeners to the component created by the description.

The **ComponentDescriptionFactory** can be created by programmer himself, but the most often way is to create it by the **registerComponent** function, which accept one argument of **ComponentFacotry** type. This method creates **ComponentDescriptionFactory**, which returns a description with passed **ComponentFacotry**. A usage is shown on the following example:

```
ComponentDescriptionFactory myComponent = registerComponent(factory);
```

The **ComponentDescriptionFactory** is used to create a component description or directly in the mounting into HTML element:

```
Element mountRoot = querySelector("#container");
```

```
var props = {};
```

```
var children = [];
```

```
ComponentFacotry myComponentDescription = myComponent(props, childre);
```

```
// Or directly in mountComponent
```

```
mountComponent(myComponent(props, children), mountRoot);
```

4.3.1 Component

The `Component` class is the main class of the API of the **Tiles** library. Every custom component should extend or implement it.

The `Component` class contains a constructor, life-cycle methods, the `render` and a `redraw` method, props, children and offer a `needUpdate` stream. The default `Component` implementation is following:

```
class Component {
  dynamic props;
  List<ComponentDescription> children;
  final StreamController _needUpdateController;
  Stream<bool> get needUpdate => _needUpdateController.stream;

  /**
   * Life cycle
   */
  Component(this.props, [this.children]):
    this._needUpdateController = new StreamController<bool>() {}
  didMount() {}
  willReceiveProps(dynamic newProps) {}
  shouldUpdate(dynamic newProps, dynamic oldProps) => true;
  List<ComponentDescription> render() {}
  didUpdate() {}
  willUnmount() {}

  redraw([bool now = false]) {
    _needUpdateController.add(now);
  }
}
```

The easier way to create own component is by extending the `Component` class:

```
class MyComponent extends Component {
  MyComponent(props, [children]): super(props, children) {}

  render() {
    return div(null, [
      span(null, "This is simple component")
    ]);
  }
}
```

```

    }
}

```

The `MyComponent` only renders `ComponentDescription` of a `div` DOM component which contain children with a span description.

4.3.2 DOM component API

The `Tiles` library contain prepared `ComponentDescriptionFactory` functions for the each of standard HTML elements, with standard arguments. They creates new `ComponentDescription`, describing `DomComponent` with passed props and children.

Usage of the factories are shown on the previous example.

4.3.3 Browser specific API

There are 3 main additions when the virtual DOM is rendered in the browser.

- Mounting
- References
- Event listeners

The **mounting** is implemented by the `mountComponent` method with 2 arguments, `ComponentDescription` `description` and `Element` `mountRoot`. It will mount the described component into the `mountRoot` element.

```
mountComponent(myComponent(props, children), mountRoot);
```

References are part of the props. If component have in `props["ref"]` instance of an internal class `_Ref`, which is only a function returning `void` with one `Component` argument, then when this component is created and mounted, this reference method is called with it.

It is useful, when a custom component want to have a reference to the element associated to one of it's descendant. Example of the usage is in the following example:

```

class MyComponent extends Component {
  /* ... */
  Element input;

  render() {

```

```

    return input({"ref": (component){
        this.input = getElementForComponent(component);
    }});
}
}

```

Event listeners are represented in a form of a Map, stored at the node. They should be instances of a `EventListener` class, which is a function with 2 arguments, an event and a component, and returns boolean. The map containing event listeners is passed to the `ComponentDescriptionFactory` as a named parameter `listeners`.

```

input(listeners: {"onClick": (event, component){
    print("Input clicked.");
}});

```

4.3.4 Server specific API

There is only one thing we want to do on the server. Create markup for a `ComponentDescription`. We created a method `mountComponentToString`, which require one `ComponentDescription` argument and returns the markup string representing the virtual DOM created from the description of a component.

```

String markup = mountComponentToString(
    span(props: {"class": "my-span"}, "Text in the span")
);
markup == '<span class="my-span">Text in the span</span>' // is true

```

Chapter 5

Performance

The performance is an important aspect of the UI library¹. If the UI library is slow, the user experience is unsatisfactory and application is considered as worse as it can be.

The question about performance of the UI library can be divided into the initial render, the update of the UI and the continuous consumption of computational resources.

The complexity of each task of the UI can be dependent on data size and resulting structure size. These two aspect are generally connected.

We will take into account both of them. Data are a partial input of all tasks. The resulting structure combine the input data and the complexity of the UI. Therefore "How long it takes to process the structure of a size N" is in some cases a good question.

The complexity of the application logic not reflected into the complexity and a size of the UI structure is not relevant for as, because it is independent from the UI library. Therefore we will suppose, that it is constant.

5.1 Initial render

The initial render creates the whole structure of the UI, therefor it is not possible to perform it in less then a linear time from the size of the resulting structure. It should be at least constructed.

The performance is also at least linear from the size of the read data(they must be at least read).

The **Tiles** library constructs virtual DOM representing the UI with a tree of the nodes. Each node contain the constant size of the information used by the **Tiles** library.

¹as almost everywhere

The creation of the virtual DOM will process each node in the virtual DOM tree constant number of times. We assume a constant complexity of the application logic, so every life-cycle method called by our library will take a constant time. Therefore the creation of the virtual DOM is linear from the size of it.

The initial render will process the virtual DOM by the depth-first search and creates the DOM structure in the process of the search without any looping and repeating. So the projection from the virtual DOM into the real one is performed in linear time from the size of the virtual DOM.

At the summary, the whole initial mount consist of the creation of the virtual DOM (linear from the size of the virtual DOM), and the projection of the virtual DOM into the real one (also linear from the size of the virtual DOM). The complexity of the initial render is therefore linear from the size of the virtual DOM.

The last question is, what relation is between the size of the virtual and the real DOM.

The subtree without any DOM component in it is useless from the UI point of view, so we assume, that every leaf² of the virtual DOM is a DOM component, which will be linearly reflected to the HTML element.

A long line of not DOM nodes without any branching is also useless (it can be replaced by one virtual DOM).

From these facts is obvious, that in the most cases, the ratio of DOM components in the virtual DOM has the constant lower bound.

So the complexity of the initial render is linear from the size of the virtual DOM, which is generally linear from the size of the resulting structure.

5.2 Update of the UI

The performance of the update process is reflected into the agility of the UI of the application. We assume, that the library is used wisely by a user of it. We also suppose no usage and user optimization by `shouldUpdate` method.

In the **Tiles** library, when data are changed and the redraw method was called, the competent node is marked as dirty. The update process will redraw the whole virtual DOM structure under the dirty node. The list of changes is produced and subsequently applied to the real DOM.

The complexity of the update process is therefore asymptotically equal to the size of the subtree of the virtual DOM with the root in the dirty node. This is in the worse case the whole virtual DOM.

²A leaf is also a subtree

This complexity can be decreased by proper usage of the `shouldUpdate` function.

In general scenario, parts of the UI dependent on a data change are not huge, which decreases the time needed for the update.

5.3 Continuous resource consumption

As we perform an asynchronous updating of the virtual DOM, the only one repeated task is to check, if there is something to be updated. This is not a performance bug, it will be discussed in the [section 5.4 Performance optimizations](#).

The check of the update need is performed on each mounted virtual DOM in the constant time(`root node isDirty` or `hasDirtyDescendant`). When we have the whole application UI represented in one virtual DOM, there is only constant time to check the update need.

In general, the complexity of the task, which is performed periodically in the time, is linear from the number of virtual DOMs. Virtual DOMs should not be initialized dynamically from the size of a data, for this purpose should be used the virtual DOM itself. Therefore we assume the constant number of virtual DOMs in the application, so the constant complexity of the task performed in each animation frame.

5.4 Performance optimizations

The performance of the pure algorithm can be enhanced by an optimization of selected parts. For this purpose we used batched updates, the up-down update (from the root tree to the leafs of the virtual DOM) and many small refinements like `shouldUpdate` method.

5.4.1 Batched updates

Every triggered redraw of the component will cause an event in the `needUpdate` stream. The event is caught by the node containing the component, which mark itself as dirty. The synchronized process of redraw ends here.

If more than one redraw happened, all of them are batched by marking a components node as dirty. Mark the node as dirty cause adding the flag to the route from the node to the root node of the virtual DOM as *have dirty descendants*.

On every animation frame, each root node is check for the need of the update, and if it is dirty or has dirty descendant, the update is started. The update perform all needed updates in the virtual DOM in one batch. The update is perform only when

the real DOM will be drawn to the user ³.

5.4.2 Up-down update process

We can perform the update process of batched updates from the root to leafs of the virtual DOM tree. Therefore when the node is deleted from the virtual DOM, none of its descendants will be updated, even if they wanted to. The update of removed node would be useless, because its change will be never reflected into the real DOM.

The up-down update also saves the performance from repeating the update of the same node. For example, we have 4 nodes A, B, C and D, when the A is parent of the B, the B of the C, etc. If they trigger the redraw from the bottom to the root, the order is D, C, B and A. If we perform updates in the order in which the redraw was triggered, we will perform following updates:

1. update D,
2. update C, and also D, as it got a new props,
3. update B, so update the C and D too and
4. update A, which cause the update of the B, the C and the D too.

When we perform update process of the batched updates from the root to leafs, every node will be updated only once.

1. update A,
2. update B,
3. update C and
4. update D.

³The information about redrawing the UI by the browser is implemented by the *animation frame* event

Chapter 6

Benchmarks

Conclusion

Here will be conclusion of wholw thesis

Bibliography

- [col] A Facebook & Instagram collaboration. *React - A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES*.
<http://facebook.github.io/react/>.
- [Goo] Google. *AngularJS - HTML enhanced for web apps!*
<http://angularjs.org/>.
- [INC] TILDE INC. *ember - A framework for creating ambitious web applications*.
<http://emberjs.com/>.
- [Kea] Tyler Keating. *The Run Loop*.
<http://blog.sproutcore.com/the-run-loop-part-1/> and <http://blog.sproutcore.com/the-run-loop-part-2/>.
- [wika] Flex SDK wiki. *Two-way Data Binding - Functional and Design Specification*.
<http://sourceforge.net/adobe/flexsdk/wiki/Two-way%20Data%20Binding/>.
- [Wikb] Wikipedia. *Data binding*.
http://en.wikipedia.org/wiki/Data_binding.