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:	FMFI.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 Rovan, 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 Rovan, 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 slovencine.

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

Contents

List of Figures

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 - why databinding

The first question, as always should be, is the motivation of this work. What is the motivation to create another library, that will handle databinding in dart?

We will start with small introduction to history of how websites and later web-applications was created. Then we define a set of features required for **tiles**.

1.1 History

1.1.1 Plain documents

1.1.2 Simple PHP

1.1.3 Server side frameworks

1.1.4 Simple JavaScript/jQuery

1.1.5 JavaScript MVC frameworks

1.2 Objectives

From previous overview of "history" we can produce set of features, which should be contained in **tiles**.

1.2.1 Server-side rendering

1.2.2 Programmer friendly API

1.2.3 Easy concept

1.2.4 Two way databinding

Chapter 2

Databinding

In this chapter we will introduce problematics of databinding more deeply then in introduction.

2.1 One way databinding

Discuss one way databinding.

2.2 Two way databinding

Discuss two way databinding.

Chapter 3

Existing solutions

3.1 Template driven

Discuss databinding based on filling some type of template with model. This approach is used in standard MVC frameworks like AngularJS, Ember or UI libraries like Polymer.dart.

3.2 Component driven

Discuss databinding based on component approach used for example in React from facebook or our library.

Chapter 4

Our solution

In this chapter we will introduce and deeply describe our Dart library `Tiles`.

4.1 Requirements

In this section we write down a list of requirements on our library.

4.2 Architecture

In this section we describe our architecture from couple points of view like High level idea, Structure, Core, Life-cycle, Events, Rendering and Injecting.

We will focus on good understanding of how library works. We will not discuss API a lot, this is the focus of next section.

But, of cause we add some examples, so wee will show some parts of api in this section too, but they don't will be so much described as in next section.

4.2.1 High level idea

Our high level idea inherit from facebook react library. We created api, whose main class is `Component`, which represents construct very similar to react's `Component`. This component is mounted to some element, where it renders itself. This relationship is shown on figure Idea.

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

There can be event listeners attached to these components. **Events** [1] are then

[1] We work at Dart, which create browser compatibility for us, so we don't have to create synthetic

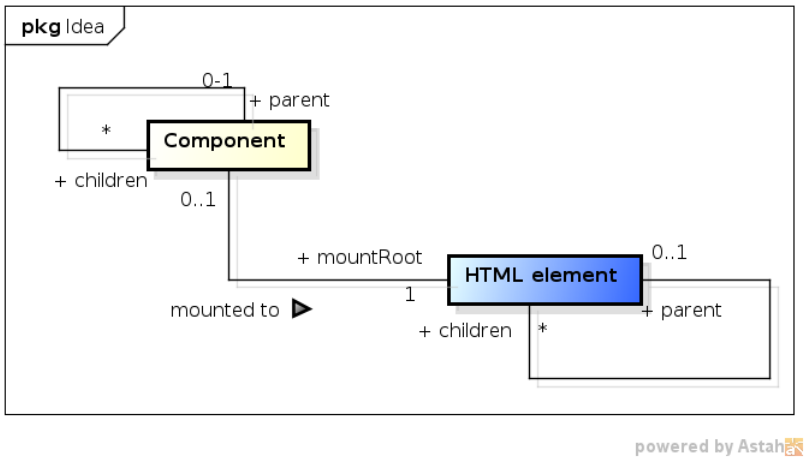

Figure 4.1: Idea

bubbled trough virtual DOM, instead real one. By this there can be listener attached to custom component, which don't have element representing it in real DOM.

As we work in Dart language, it is natural to try to reuse most of code on both, client and server. So other important part of idea is **server-side rendering**, which is meant to easy rendering the same content on server as on client's browser. It is very important for SEO purposes and smooth user experience.

4.2.2 Structure

We split our library to 3 partially dependent packages.

Tiles

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

Tiles Browser

These package is used for mounting components to real HTML elements. It maintain relationships between elements and components, simulate events bubbling and keep real DOM in sync with virtual one.

Tiles Server

Tiles Server maintain server-side rendering. It offers api to render component structure to string with markup based on DOM components.

events like react.

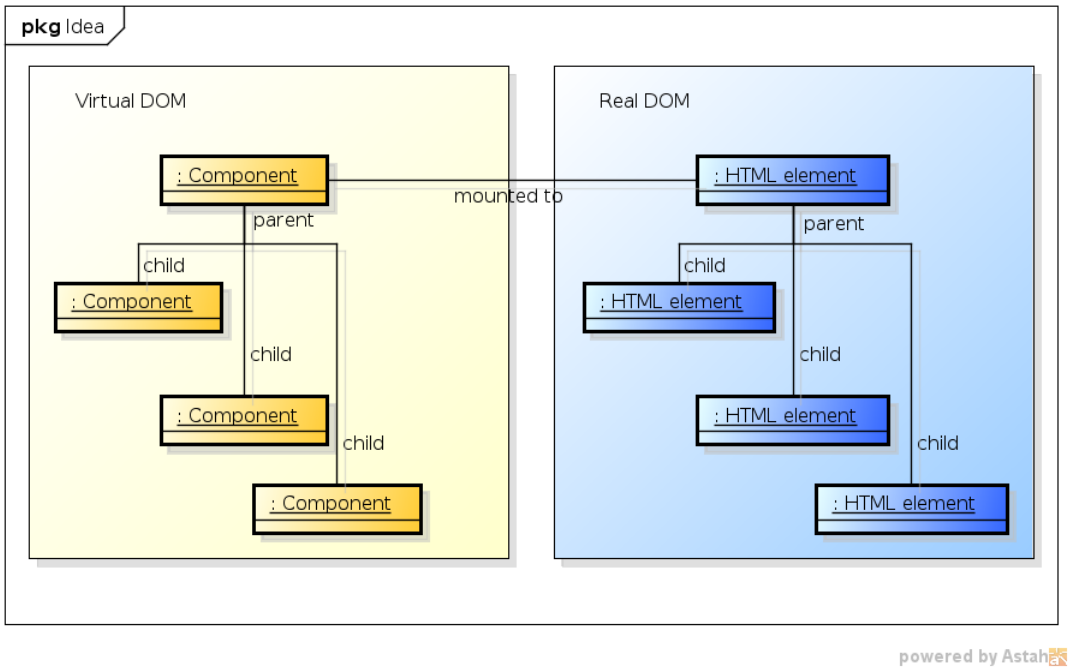

Figure 4.2: Virtual DOM

From this it is quiet obvious what are dependences between these packages. **`Tiles`** is independent, and both of **`Tiles Browser`** and **`Tiles Server`** are dependent on **`Tiles`**. These dependences are shown on figure Packages.

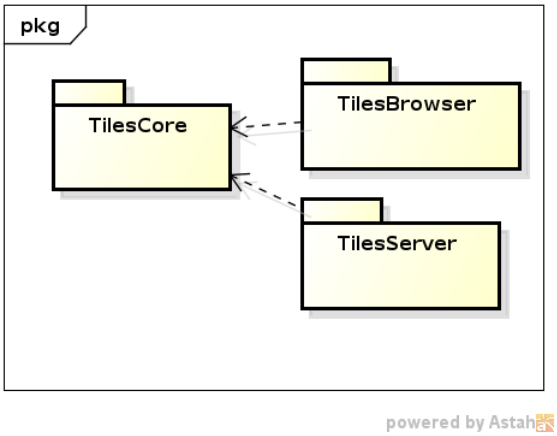

Figure 4.3: Packages

4.2.3 Core

We have 4 main classes in core of the library.

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

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

ComponentDescription is, as it sound like, description of a component. It is returned from component to describe it's children. We discuss this principle later.

NodeChange represents one change in virtual DOM, which should be applied into real DOM. By this, we achieve changing real DOM by smallest difference in virtual DOM. Types of change are `CREATED, UPDATED, MOVED, DELETED`

In contrast with facebook react `Component`, our component do nothing else than api offered to programmer. This class is the main class for programmer which use our library, he don't need to use any other class created by our library. Just some methods.

We also got inspiration from react with idea of **virtual DOM**. This virtual DOM is not represented by tree of components, but to separate functionality, it's vertices are constructed by instances of class `Node`.

Every node contain instance of `Component`. For this `Component` instance this node is something like *representer of me in virtual DOM*.

The whole image of relationships is shown on figure Core of the library. Now we will focus on the main classes separately.

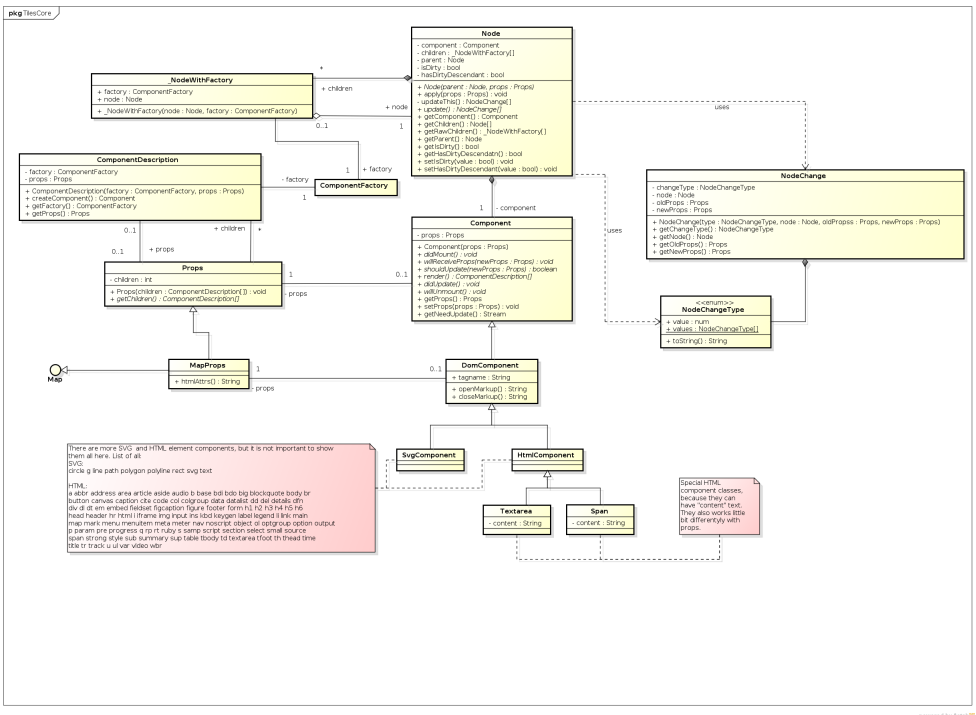

Figure 4.4: Core of the library

Component

Component, as in react, is the main building brick of application(library). It offers api to the programmer with life-cycles, props and so on.

Life-cycle methods will be discuss later, lets focus on the role of component in the whole library.

Component is a class, which represents functionality of certain part of UI[2] in an application. It is created with some props and children and it is upon it to do what ever it wants with it. But main purpose of it is to create some structure below[3] it, add some event listeners and update itself sometimes, e.g. on some event occur.

The main method of the `Component` is `List<ComponentDescription> render()`. By this method component creates it's substructure. It will return list of children of this component, represented by instance of `ComponentDescription`. `Node`, which owns this component (and called it's render method) will take care of the rest. Basically, it will return something 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 first 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 factory)` is also very helpful method. By passing it it gets `ComponentFactory` as an argument and from it, create `ComponentDescriptionFactory`. This is mainly created for easy use of the library, we will show why these factories exists and how are they used in section 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 have attributes saved in `Map`, `render` method, which return `children` member variable and `svg` and `pair` flags.

Specific HTML elements are then created as with different `ComponentFactory` and also different `ComponentDescriptionFactory`, which is used to easily create `ComponentDescriptions` of `DomComponent` in custom component render method.

[2] UI = User Interface

[3] From the virtual DOM tree point of view

ComponentDescription

`ComponentDescription` is description of a component. It describes what component should be rendered.

For this purpose, it's need 4 types of information:

- **Type of the component**
  To create instance of some 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, to be new component created with them.

- **Children**
  Children of described component. This is useful mainly when programmer want to render more complex structure of `DOMComponent`s.

- **Key**
  Key is an identifier of a child. If component's function render returns list of children with keys, and after update it returns the same children but in different positions, tiles only reorder this children and not remove, add them or change their properties.

It has all these information as final. Description is once created, with all informations and then these information can't be changed. All these information is setted up bu constructor.

`ComponentDescription` have 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 creates virtual DOM tree, maintain creating and updating of it based on results of component's `render` method, listen to component's `needUpdate` stream and mark self as *"dirty"* when it's component need update and handle process of updating which rearrange children of the `Node`.

Children are stored trough class `NodeChild`, which represents all information about child (node, component's factory and key). By this encapsulation, when update is triggered, we can compare factory of a component description and factory of a child and decide, if we need to replace this child or it is enough to update it.

As the key is also stored in `NodeChild`, we also compare old children with next one when doing update, and if there exists child in old children with the same key as in the next one, this child "step out of the line" and only move around without replacing.

Node have two important flags: `isDirty` and `hasDirtyDescendant`. These flags represents information, if some node need to be redrawn. If `isDirty` is true, this node need to be updated, because component of this node called `redraw`. If `hasDirtyDescendant` is true, that means, that there exist a descendant of this node, which want to be updated. When `hasDirtyDescendant` is true and `isDirty` is false, component don't have to update itself, it is enough to call update on child nodes.

Method `List<NodeChange> update()` is doing this update process. It returns list of changes, which was needed to put node in new state. This update is mainly called by browser part of a library and this list is used to translate changes in virtual DOM to real one. Method consists of several main steps.

1. check, if update is needed by flags `isDirty` and `hasDirtyDescendant`, if no, return,
2. if component of this node need update (`isDirty == true`), update this node with rearrangement of children,
3. add results of update calls on children to result,
4. set this node as not dirty and not have dirty descendant and return.

Rearrangement of children by calling `render` method of this component and adapting node's children to returned descriptions has not so difficult as complex algorithm, which we will not describe here. It is fully documented in the source code related to this work, extracted to specific method private to the library which is in own file: https://github.com/cleandart/tiles/blob/master/lib/src/core/node_update_children.dart

NodeChange

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

When some node in the virtual DOM is updated by method `update`, list of changes is returned. This list is then processed by browser part of a library, which mirror these changes to real DOM.

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

Type is stored as instance of `NodeChangeType` enum and can be one of `CREATED`, `UPDATED`, `MOVED` and `DELETED`, whom meaning is obvious. When type is `UPDATED`, old props and new pros take effect.

4.2.4 Life-cycle

Every instance of `Component` have own life-cycle. As every object, first it is created. Then, when component is mounting or rendering into text, it is rendered, and then it is mounted. Then it lives it's own life.

When something "higher" want to update it, it will first receive props, then it is asked, if it should be updated, and if yes, then it is rendered. After that, it was updated, of course.

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

At the end of component's life, component should be notified about that it will be unmounted(e.g. from DOM), to be able, to do some modifications to it's refs, destroy timers and so on.

This whole life-cycle is shown on the figure Life cycle of a `Component`.

Create

Create part of life-cycle is implemented by constructor of `Component`. It will receive props and optionally children as arguments and it should do whatever it needs to prepare whole state of object to live.

An trivial example of constructor of `Component` is

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

which only call's constructor of super class `Component`

Example of more complex constructor should be e.g. component which maintain example `Todo` instance.

```
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

`Component` life-cycle **Did mount** is implemented by method `didMount`. It is called after component is mounted to DOM.

This is the correct place to initialize for example timers, stream listeners and so on.

For example, in our `MyTodoComponent` we should listen for change of todo on server, and if it was changed, we can redraw component.

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

  // ...

  didMount() {
    this.subscription = this.todo.changedOnServer.listen((change) {
      this.redraw();
    });
  }

  // ...

}
```

Will receive props

Will receive props life-cycle method is `willReceiveProps`. It is called every time, when component will recevie new `props`, except first time, when these `props` are passed to constructor.

This is place, where old props and new props can be compared, so this is right place to make changes based on difference in old and new props.

Example of `willReceiveProps` in our `MyTodoComponent` should compare `todo` of old and new props and there are not equal, it can 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

Should update is partly lifecycle, partly not. It is a question, if component should update on this props-change.

This "life-cycle" is implemented by method `shouldUpdate`. This method is used mainly for speed up performance. By default it returns true, so if it is not implemented in custom component, it will update always.

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

Example in `MyTodoComponent` should look like this:

```
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 method `render`, which have no attributes. 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 life-cycle method `didUpdate`, by which is implemented this life-cycle event, is triggered, component, and programmer, can be sure, that component is mounted and there exist elements in DOM for each DomComponent descendant.

Will unmount

This event is implemented by method `willUnmount`, which contain no arguments.

It is called right before it is unmounted from dom.

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 our library is rendering of a content. By `Component` and `Node` we can create virtual DOM tree, but now we want to render this structure into something independent from our library.

As our work is in the scope of web applications, we want to render it into DOM (browser rendering), and also into textual representation of a DOM(server rendering).

As we described earlier in subsection 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, because of that we want to render our virtual DOM structure into a string, which user of this library can return as a response to browser request.

When we have string, which represents markup of DOM created from virtual DOM, created by our library, we can do server-side databinding, which can be reuse in the browser to smooth user experience.

Our target, render virtual DOM into markup string is quiet easy. From the programmers point of view, he will use `ComponentDescription` to describe, what component he want to render. Our server-side package of library will get this description, creates component from it, puts it into node and perform update of node. By this, virtual DOM is created.

Now, as we have virtual DOM, we can render it's markup by depth-first search of it's tree. In the search, when we came into node, we will do something like this algorithm (pseudo-code):

Data: node in virtual DOM
Result: String with markup of subtree of virtual DOM with root in setted node
if *component is DomComponent* **then**
  if *component is not pair* **then**
    write markup with attributes from props;
  else
    write open markup;
    write markup for all children recursively;
    write close markup;
  end
else
  write markup for all children recursively;
end

Algorithm 1: Write node into string.

In browser

Rendering in browser is quiet more difficult than rendering to string. We can use same render to string method, but we will need some connections between nodes and elements, so we can't do it this simply.

Initial mount First the user of the library need to do is to mount component to the HTML element. Of cause, he will mount component description, not component

directly. When component is mounting, it is created, placed into the node and after this, node is "updated". It is initial update which creates virtual DOM.

When virtual DOM is created, we need to construct real DOM under the root element (element, which was component mounted to) from "virtual image".

For now, we describe case, that root element is empty (has no child element or node). Case, when it is not empty we discussed in the subsection Injecting. The mount is easily described by next algorithm:

```
Data: node in virtual DOM and HTML element, to mount node to
      (mountRoot)
Result: Mounted node into element
if node.component is DomComponent then
    if node.component is not pair then
        create element representing component;
        add created element to mountRoot;
    else
        create element representing component;
        add created element to mountRoot;
        for child in node.children do
            run recursively with created element as mountRoot and child as node;
        end
    end
    save relations between created element and node;
else
    if node.component is tex component then
        create HTML text node with text from component;
        add text node to mountRoot;
        save relations between created text node and node;
    else
        for child in node.children do
            run recursively with mountRoot and child as node;
        end
    end
end
```

Algorithm 2: Write node into string.

As we can see, algorithm is recursive and skips custom components. Also it creates relations between created elements and nodes. Which are these relations we discuss later, when we need them.

By this algorithm, it is obvious, that we have real DOM with the same structure,

if we can obtain from virtual DOM by removing nodes with custom components and connect their children with their parent.

Update Later, there can be situation, that virtual DOM want to be updated. This is when some node was marked as *dirty*. Then framework perform update of this node, which triggers update of the subtrees with roots in dirty nodes. This updates return lists of changes in virtual DOM, which should be applied to browser element structure.

These updates should be processed by it's type. But for every type we need the information about which HTML element represents some node. This is first relation, which we need to remember, when we initialize mounted relation, relation `Node` → `Element`. This relation is stored by map `Map<Node, Element>`.

But what happened when we want to apply node change into real DOM structure? For each type of change something different of cause:

CREATED when new node is created, it should be mounted into the DOM. If it has `DomComponent` inside, HTML element will be created and placed at the correct place. If it has some custom component, this change will be ignored.

UPDATED If node was updated, then if it has `DomComponent`, it's element is updated with setted props.

MOVED Node or it's children(if it is node with custom component) is moved to new position.

DELETED Element of node or elements of its descendants(if it is node with custom component) are removed from DOM.

4.2.6 Events

As we were created dart library which creates virtual DOM, composed from nodes, which contains components, it is obvious that we can "simulate" event bubbling trough this virtual DOM.

Add more content.

This is fully in domain browser part of a library.

4.2.7 Injecting

4.3 API

Documentation of offered API of our library.

4.3.1 Component

4.3.2 Browser specific API

4.3.3 Server specific API

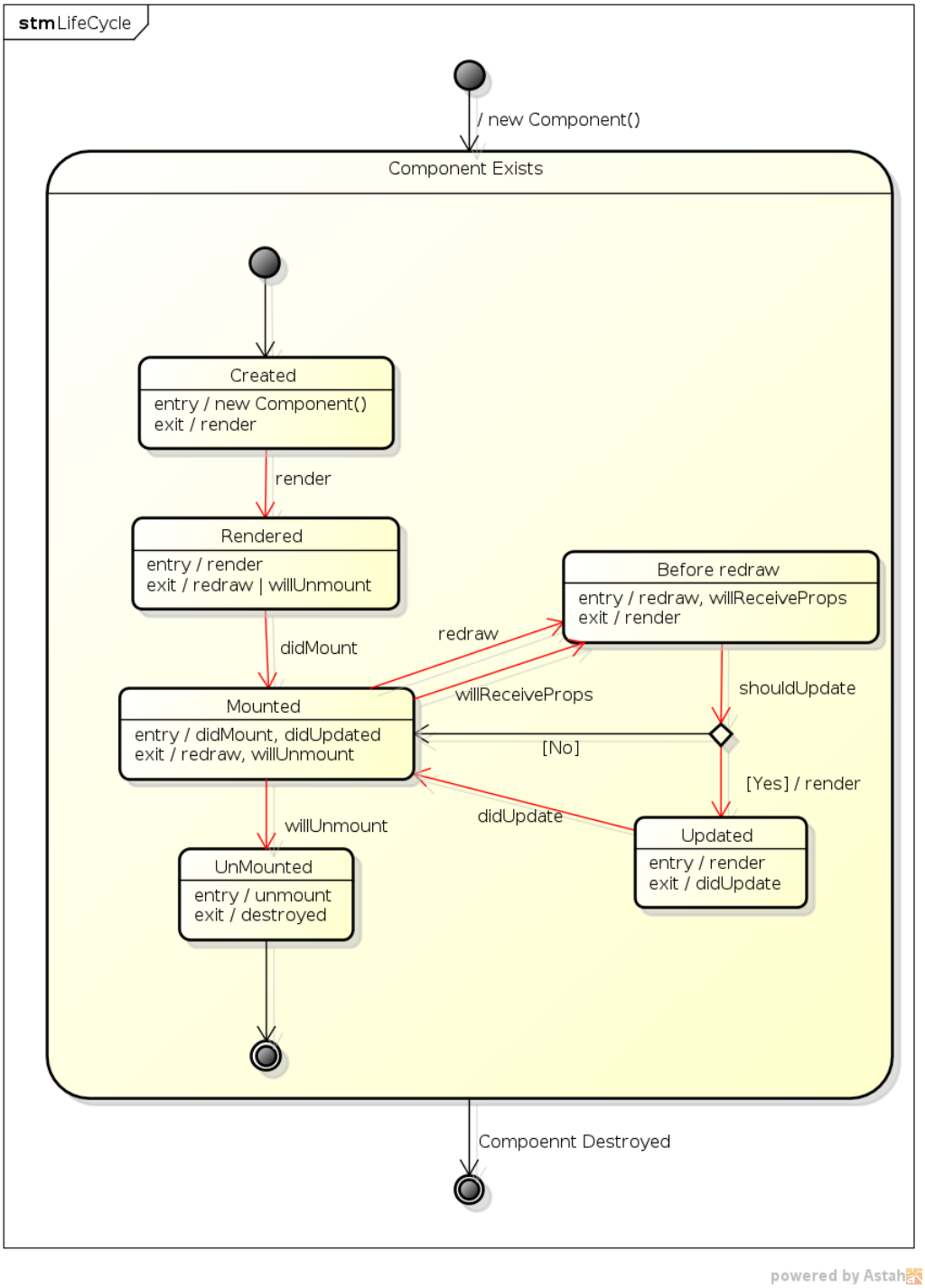

Figure 4.5: Life cycle of a `Component`

Chapter 5

Performance

Chapter 6

Benchmarks

Conclusion

Here will be conclusion of wholw thesis

Bibliography

[Aja10] AjaxPatterns.org Wiki. *RESTful Service*, 2010.
http://ajaxpatterns.org/RESTful_Service.

[jav12] *Java web frameworks discussed*, 2012.
http://entjavastuff.blogspot.com/2012/01/java-web-frameworks-discussed.html.

[JQU12] JQUERY FOUNDATION AND THE JQUERY UI TEAM. *jQueryUI Demos & Documentation*, 2012.
http://jqueryui.com/demos/.

[Mic] Microsoft Developer Network. *Model-View-Controller*.
http://msdn.microsoft.com/en-us/library/ff649643.aspx.

[Nie03] Jakob Nielsen. *Usability 101: Introduction to Usability*, 2003.
http://www.useit.com/alertbox/20030825.html.

[Ste07] Stefan Tilkov. *A Brief Introduction to REST*, 2007.
http://www.infoq.com/articles/rest-introduction.

[Sun02] Sun Microsystems, Inc. All Rights Reserved. *Java BluePrints: Model-View-Controller*, 2002.
http://java.sun.com/blueprints/patterns/MVC-detailed.html.

[TS] Jupiter Consulting JavaScriptMVC Training and Support. *JavaScriptMVC Documentation*.
http://javascriptmvc.com/docs.html.

[zen] zenexity & Typesafe. *Play 2.0 documentation*.
http://www.playframework.org/documentation/2.0.1/Home.