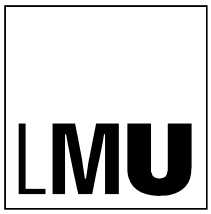
Institut für Software & Systems Engineering
Universitätsstraße 6a D-86159 Augsburg

Holistic Approach for a Separable, Reactive, Model-Agnostic View Data Binding

Johannes Rummel

Master's Thesis in the Elite Graduate Program: Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program



Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Holistic Approach for a Separable, Reactive, Model-Agnostic View Data Binding

Matriculation number:	1277688
Started:	May 15th, 2014
Finished:	November 15th, 2014
First assessor:	Prof. Dr. Alexander Knapp
Second assessor:	Prof. Dr. Bernhard Bauer
Supervisors:	Dipl.-Inf. Univ. Ralf S. Engelschall Jochen Hoertreiter

SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

I, hereby certify that this thesis has been written by me, that it is the record of work carried out by me and that I have not used anything else but the indicated sources and tools.

München, den 24. Oktober 2014 Johannes Rummel

Abstract

CONTEXT User interfaces in rich clients today contain complex logic and need to be performant, while offering powerful functionality. Their development becomes more complex and there is a great need to simplify recurring tasks that require more and more boilerplate code. One such task is synchronizing values between the view mask and a presentation model. It arises in popular architectural patterns like Model-View-ViewModel or Model-View-Controller.

MOTIVATION The initial construction of the view mask using the presentation model is tedious and produces unmaintainable code. Writing logic that reflects every possible change in the presentation model or the view mask is even worse and error-prone. A common inefficient workaround is to rerender the whole view mask again after every change to the presentation model. Solutions addressing this problem lack flexibility, efficiency or architectural cleanliness. Some do not concentrate on view data binding only.

APPROACH We define the smallest set of concepts and their semantics for achieving a powerful view data binding. Our binding is agnostic to the presentation model implementation and follows the reactive programming paradigm. That means, it is scalable, event-driven and responsive. To encourage the important architecture principle of Separation of Concerns, we allow to define the binding separately. We further permit both declarative and imperative specification of the binding inside the view mask itself or in its corresponding rendering code. We define a Domain-Specific Language and use it for all approaches. To take this holistic approach even further, we design our architecture to be extensible and pluggable.

CHALLENGE Especially in the context of web applications, we need to find efficient algorithms and data structures that realize view data binding with surgical Document Object Model (DOM) updates. The data and event binding need to hide their complexity from the developer to reduce the amount of boilerplate code. We want a powerful syntax that is easy to understand and that is, in spite of its compactness, still intuitive.

SOLUTION We compare existing solutions to our approach and present an architecture that focuses on modularity and adaptability. We test whether our concepts can be implemented by presenting BindingJS[1], a JavaScript library. We evaluate it by comparing it with existing solutions and by incorporating it into a real world web application.

[1] http://www.bindingjs.org/

Contents

1. Big Picture **1**
1.1. Evolution of the Web . . . 1
1.2. Rich Web Clients . . . 2

2. The View Data Binding Problem **5**
2.1. Architectural Patterns in Web Applications . . . 5
2.1.1. Model-View-Controller (MVC) . . . 6
2.1.2. Model-View-ViewModel (MVVM) . . . 8
2.1.3. Model-View-Controller / Component Tree (MVC/CT) . . . 9
2.2. Manual View Data Binding . . . 11
2.2.1. TodoMVC . . . 12
2.2.2. msg TimeSheet . . . 14
2.3. Reactive Programming . . . 16
2.4. Motivation . . . 18

3. Existing Solutions for View Data Binding **21**
3.1. Solutions targeting Desktop and Thin Web Clients . . . 21
3.1.1. Eclipse Rich Client Platform (RCP) . . . 21
3.1.2. JavaServer Faces (JSF) . . . 22
3.1.3. Windows Presentation Foundation (WPF) . . . 24
3.2. Libraries for Rich Web Clients . . . 26
3.2.1. Facebook React . . . 27
3.2.2. Ractive.js . . . 29
3.2.3. KnockoutJS . . . 30
3.2.4. Other Libraries . . . 31
3.3. Comparison of Rich Web Client Libraries . . . 33
3.3.1. Methodology . . . 33
3.3.2. Results and Conclusion . . . 34

4. Concept and View Data Binding Ontology **35**

5. View Data Binding Concepts **39**
5.1. Core Binding Concepts . . . 39
5.1.1. Selection . . . 39
5.1.2. Binding . . . 42
5.1.2.1. Adapter . . . 44
5.1.2.2. Connector . . . 46
5.1.2.3. Binding Scope . . . 54
5.1.3. Iteration . . . 56

5.2. Core Structure Concepts . . . 63
5.2.1. Identification . . . 63
5.2.2. Insertion . . . 64
5.3. Convenience Binding Concepts . . . 66
5.3.1. Two-Way Binding . . . 66
5.3.2. One-Time Binding . . . 67
5.3.3. Resource Sequence . . . 69
5.3.4. Initiator . . . 72
5.3.5. Parameter . . . 74
5.3.6. Expression . . . 77
5.4. Convenience Structure Concepts . . . 83
5.4.1. Template . . . 83
5.5. Domain Specific Language . . . 84

6. Implementation, BindingJS 87
6.1. Architecture . . . 87
6.2. Application User Interface . . . 92
6.2.1. Public API . . . 92
6.2.2. Binding API . . . 94
6.2.3. Socket API . . . 99
6.2.4. Plugin API . . . 101
6.3. Algorithms . . . 105
6.3.1. Repositories . . . 105
6.3.2. Binding Scope . . . 105
6.3.3. Parser . . . 106
6.3.4. Preprocessor . . . 108
6.3.5. Iterator . . . 121
6.3.6. Propagator . . . 126
6.3.7. Sockets . . . 130

7. Evaluation, BindingJS 131
7.1. Comparison with other Rich Web Client Libraries . . . 131
7.2. Employment in Applications . . . 137
7.2.1. TodoMVC . . . 138
7.2.2. msg TimeSheet . . . 142

8. Conclusion and Future Work 143

A. Appendix 147
A.1. Comparison of Rich Web Client Libraries . . . 147
A.1.1. Criteria Catalog . . . 147
A.1.2. Evaluation . . . 153
A.1.3. List of Libraries . . . 160
A.2. Code Listings . . . 162
A.3. Figures . . . 178

B. Glossary 181

C. Bibliography 185

List of Figures

1.1. Teasing Example — evolutionoftheweb.com . . . 1
1.2. 7 Layer Architecture (Excerpt) [Enga] . . . 3

2.1. Typical Collaboration of MVC Components . . . 5
2.2. View Data Binding with Knockout . . . 9
2.3. Artificial Example of Reusing a Login Component in ComponentJS . . . 10
2.4. TodoMVC Implementation . . . 13
2.5. msg TimeSheet, Main View . . . 15
2.6. Separation of Concerns Introduced by Cascading Style Sheets . . . 18

3.1. Eclipse IDE, an RCP Application . . . 22
3.2. HTML, Rendered by JSF . . . 24
3.3. View Data Binding Flows in Windows Presentation Foundation . . . 25
3.4. Facebook React Comment Box — Component Structure . . . 27
3.5. Ractive.js Item List . . . 29

4.1. Concept Ontology . . . 35
4.2. View Data Binding Ontology . . . 37

5.1. Typical Data Flow through Adapter and Connectors . . . 50
5.4. Annotated Binding Specification . . . 86

6.1. BindingJS, Context Diagram . . . 88
6.2. BindingJS, Component Diagram . . . 89
6.3. BindingJS Engine, Component Diagram . . . 89
6.4. BindingJS API . . . 93
6.5. State Diagram, View Data Binding . . . 96
6.6. Reference, Class Diagram . . . 104
6.7. Repositories, Class Diagram . . . 105
6.8. Binding Scope, Class Diagram . . . 106
6.9. Preprocessor, Overview . . . 108
6.10. Plain Iteration, Class Diagram . . . 113
6.11. Plain Iteration Tree, Object Diagram . . . 116
6.12. Expanded Iteration, Class Diagram . . . 117
6.13. Plain together with Expanded Iteration Tree . . . 117
6.14. Expanded Iteration Tree, Object Diagram . . . 120
6.15. Iteration Instance, Class Diagram . . . 121
6.16. Iteration Components, Simplified . . . 124
6.17. Influences of Bindings as Graph . . . 129

A.1. WPF Application Demonstrating Data Binding . . . 178
A.2. Selenium IDE Test Case Execution . . . 179

1. Big Picture

1.1. Evolution of the Web

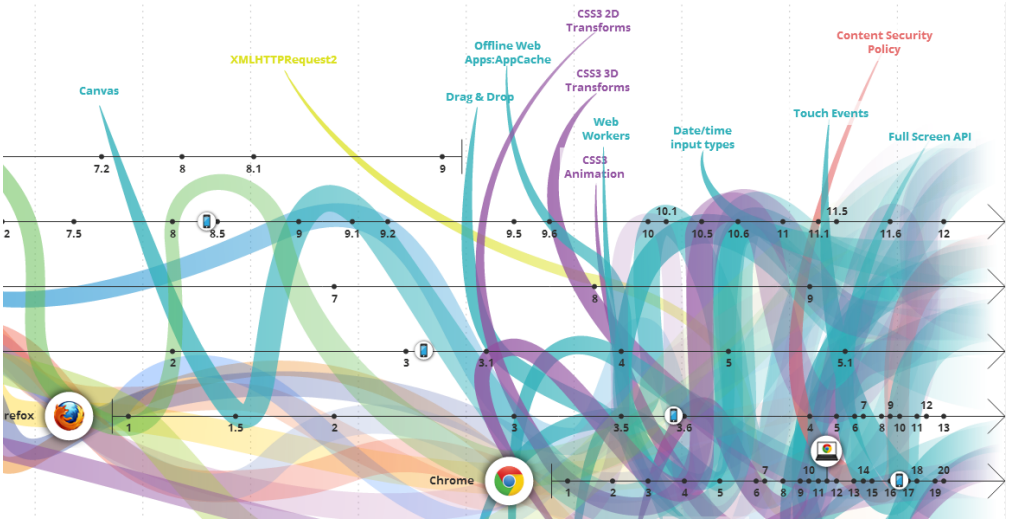


Figure 1.1.: Teasing Example — evolutionoftheweb.com

In spite of its short history, the Internet has come a long way. Around twenty years ago, static web pages were dominant. To provide a more interactive experience, developers later augmented them with pieces of JavaScript code. More ornate components however required users to manually install plug-ins like Adobe Flash or Microsoft Silverlight. Fortunately their most used features like embedded videos are now slowly replaced by HTML 5 [Har]. To make user interfaces more appealing, Cascading Style Sheets (CSS) became an important part of every web developer's toolbox. A more detailed overview can be found at evolutionoftheweb.com, which is shown in figure 1.1 and in turn is a very good example of what is currently possible.

These fundamentals were then adopted to build sophisticated widget and JavaScript libraries, e.g.,dojo or jQuery. Nowadays we see powerful frameworks such as AngularJS, Backbone.js and even whole programming languages like Dart. They all address the inability of native web technologies to support extensive application logic. JavaScript alone also does not encourage using proven architectural patterns, which become more and more important with increasing amounts of code.

The current trend is for web browsers to run powerful applications instead of only displaying content retrieved from a server. Some examples of this include popular web sites such as facebook, YouTube or the various google products like Google Maps, Google Docs or Google Hangouts. Google even goes so far as to market an operating

system, called Chrome OS, whose main purpose is just to run a web browser [Pic].

The idea of putting different amounts of the application logic onto the client is not new. In the era of mainframes around 1980, many terminals that only displayed data accessed the central system as a client. This way it was easy to update the application and security could be handled easily. When personal computers became increasingly popular it was no longer reasonable to leave their processing power unused. This led to applications that executed almost all the application logic and needed the server only as a database. In contrast to this, a web browser again is like a terminal that displays data received from a web server [Sha]. There seems to be a cycle and what we identified as a trend to more powerful web applications earlier implies the beginning of the next step in that cycle.

There are several reasons why it makes sense to create web pages that do more than just displaying HTML from the server. About one third of the world's population goes online every day [Webi]. A client that performs parts of the work reduces load on the server and therefore operational cost. Also such a client can in many cases be fully functional even when lacking a constant Internet connection [Wikia]. This especially helps the increasing number of people who access the Internet from mobile devices [Webb].

1.2. Rich Web Clients

The terminology describing a web client who does more than just view rendering is very unclear and ambiguous. Adjectives like *heavy*, *fat*, *thick* or *rich* are often falsely used interchangeably. By identifying different responsibilities of a web client, we clarify what type of client each term defines.

There are numerous classifications of layer based architectures including Nilsson [Nil02], Fowler [FR10] and Brown Layers [Bro+03]. We focus on the layer architecture presented by Martens [Mar10], who also illustrates how to map other layer classifications onto his definition. The goal of this architecture is to find the maximum number of layers so that every web library can be assigned to exactly one layer. As a result, Martens presents the seven layers shown in figure 1.2 and proves that if their number were greater there would be libraries that fell into more than one layer.

Martens identifies three different roles, *Interaction*, *Domain* and *Data*. Layers that appear in the role of *Interaction* are responsible for interacting with the user of the application. They display and render the GUI and accept user-triggered events. On the other hand, layers having the *Domain* role cover business specific functionality. The *Data* role is not relevant for distinguishing the different types of web clients and is therefore not further considered here.

The three *Interaction* layers are *Technical*, *Abstract* and *Domain Interaction*. The *Technical Interaction* layer is responsible for displaying and rendering the view mask. In a web context this layer is usually implemented by the HTML rendering engine of the web browser and requires no further development efforts. *Abstract Interaction* deals with view components, so-called *Widgets*, and makes building user interfaces easier. jQuery UI or dojo are examples of libraries on this layer in web development. Finally, the *Domain Interaction* layer links *Interaction* and *Domain* by observing and controlling the *Abstract Interaction* layer. ComponentJS offers an implementation for this layer in web applications.

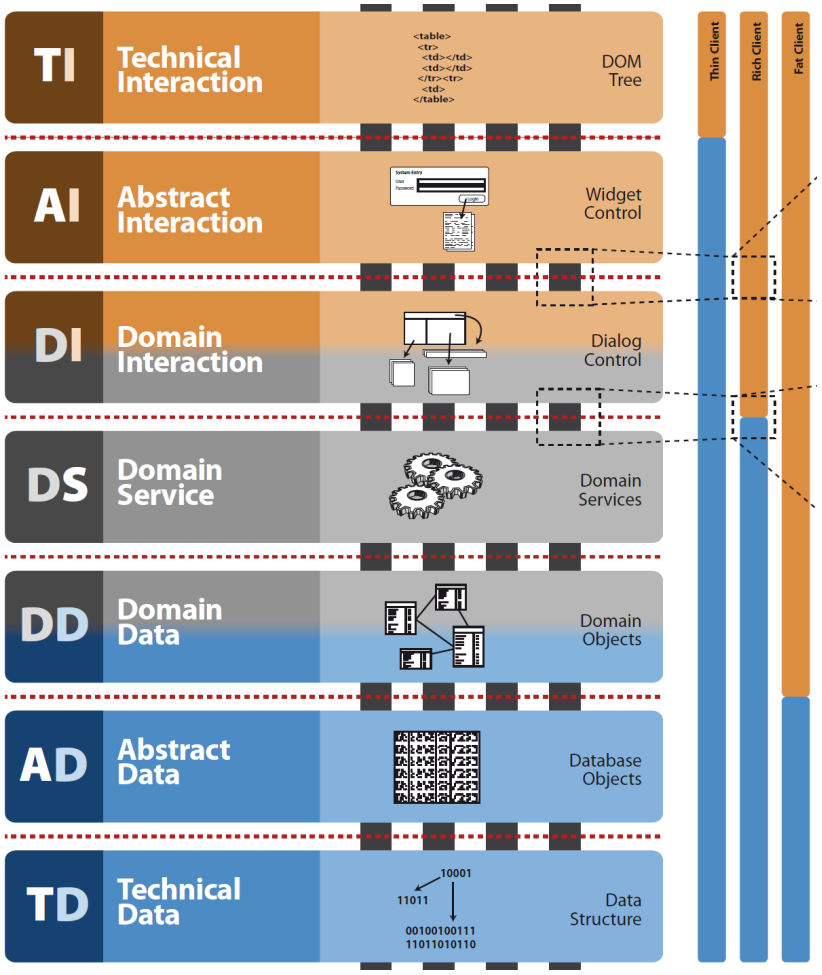


Figure 1.2.: 7 Layer Architecture (Excerpt) [Enga]

Beneath these resides the layer *Domain Service*, which uses and stores data in the *Domain Data* layer. These two layers make an application unique and contain the specific functionality required to fulfill its purpose. Of course, these layers cannot be covered by libraries; a high level of abstraction is desired elsewhere and the most time for implementing functionality should be spent here.

Having an abstract understanding of how a web application is structured, it is now clear how to define what the various types of web clients are. They are defined by the distribution of layers to the client and the server. A *Thin Client* only consists of the *Technical Interaction* layer, while a *Rich Client* provides the whole *Interaction* role including the *Abstract* and *Domain Interaction* layer. A *Fat*, *Heavy*, or *Thick Client* on the other hand includes even more business logic and comes with all *Domain* role layers.

2. The View Data Binding Problem

The topic of this thesis is particularly specific and mostly invisible to someone not deeply involved in web development. View data binding generally means that two properties are synchronized, so that they always have the same value. While this theoretically is possible in any context, it is most important in architectural patterns, where presentation of data is separated from storage of that data. To get a better understanding of the structure of applications that our approach mainly addresses we start this chapter with an introduction to such architectural patterns.

These patterns require that the same or similar data is held in multiple locations. We show what problems arise when the binding that synchronizes these different data sources is implemented manually, without automation. A desirable way of dealing with view data binding resembles the ideas of *Reactive Programming*. We therefore explain the meaning of this before summarizing our motivation to work on this topic and what problems we want to solve with our approach.

2.1. Architectural Patterns in Web Applications

It is important to know about the most popular patterns of the Model-View-* family [SW] in order to understand why view data binding is an issue. There are many frameworks implementing these patterns, and recognizing how we are compatible to them first requires the knowledge presented in this section. There are many different interpretations and definitions of Model-View-ViewModel and especially Model-View-Controller. We focus on explaining their most important concepts, while leaving out details, which may vary.

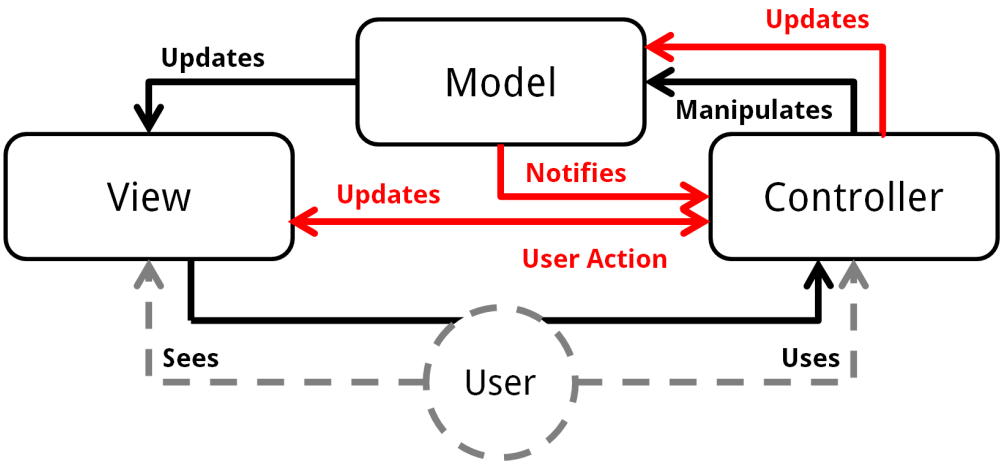


Figure 2.1.: Typical Collaboration of MVC Components

2.1.1. Model-View-Controller (MVC)

Model-View-Controller (MVC) is an architectural pattern that is used to implement user interfaces. When it was invented is unclear, but it must have been around the 1970s and approximately ten years prior to the publication of the first significant paper about MVC in 1988 [Mod]. Although this makes the pattern seem antique, it played an important role throughout the history of application development and continues to have a big part in modern web development. MVC offers a way to separate the responsibilities of presenting and managing domain information by decoupling them into three components.

Model Encapsulates the internal representation of information, related to a specific problem domain. It may also contain business rules and logic on how to manipulate and access its data [Webr].

View Is responsible for presenting model data to the user. It knows how to draw itself and is able to respond to user actions [Webs].

Controller Acts as an interface to the application for the user [Grea]. It may also perform setup and coordinating tasks for an application and manage the life cycles of other objects [Webs].

Figure 2.1 illustrates two different interpretations of the interaction between these three components. The arrows drawn in black indicate that the model is responsible for updating the view [Wikic]. The variant with arrows in red completely decouples view and model and routes all change notifications through the controller [Webs].

In web environments, the implementations vary in the distribution of the components to server and client. In thin client applications, such as a Java EE web application, all components run on the server. When a call to the controller is triggered, for instance by a click of the user on a Hyperlink, the request is executed remotely and a rendered view mask is returned. In contrast, a rich or fat client architecture moves more components onto the client. To illustrate how this is implemented we want to use Backbone.js[1], a JavaScript framework, to present a simple example.

There is an ongoing discussion as to whether Backbone.js qualifies as an MVC framework, since it has no characteristics of a dedicated controller. Some parts of the controller's functionality are placed in the view and some in a component called router, which is not explained here. It therefore seems clear that the framework is not a pure implementation of MVC. It should rather be classified as a member of a more general *Model-View-** family, since it adopts principles from MVC, MVP and MVVM (see section 2.1.2) [Bai]. Nevertheless, Backbone.js is close to MVC and it serves as a useful example in this section, especially considering its popularity.

Backbone.js provides an event mechanism that notifies view and model instances about changes to their associated counterparts. Models are defined by inheriting from a class provided by the framework as in listing 2.1. In addition, they are allowed to contain additional functionality for representing business logic like the `prettyPrint` function in line 6.

[1] http://backbonejs.org/

```
1  var Thesis = Backbone.Model.extend({
2    defaults: {
3      author: '',
4      year: 1970
5    }
6    prettyPrint: function () {
7      return this.get('author') + ' in ' + this.get('year');
8    }
9  });
10
11 var holisticApproach = new Thesis({
12   author: 'Rummel, Johannes',
13   year: 2014
14 });
```

Listing 2.1: Defining and Instantiating a Model Component in Backbone.js

A key benefit of favoring the framework over defining models with plain JSON objects is that these managed components support the observer pattern. Listing 2.2 shows how easy it is to register changes by using this technique.

Whenever the `author` attribute of `holisticApproach` is changed, Backbone.js will notify the callback function passed in as the second argument. Within this callback the change is usually reflected to the view component to update the displayed `author` accordingly. This already creates a simple one-way view data binding.

```
1 holisticApproach.on('change:author', function () {
2     updateView();
3 });
```

Listing 2.2: Register Change Listener in Backbone.js

Listing 2.3 shows how view components are defined in a similar fashion by extending a framework class. A related change listener as in listing 2.2 is registered in line 4 as soon as the view is initialized. Any time a change to the associated model takes place, `render` is executed. It is common but inefficient to render the view again after every change, although it may not seem relevant in this minimal example.

```
1  var ThesisView = Backbone.View.extend({
2    tagName: 'div',
3    initialize: function () {
4      this.listenTo(this.model, 'change', this.render);
5    },
6    render: function () {
7      // Another library like jQuery is typically used here
8      this.$el.html(this.model.prettyprint());
9    }
10 });
```

Listing 2.3: Defining a View Component in Backbone.js

2.1.2. Model-View-ViewModel (MVVM)

> *"GUIs consist of widgets that contain the state of the GUI screen. Leaving the state of the GUI in widgets makes it harder to get at this state, since that involves manipulating widget APIs, and also encourages putting presentation behavior in the view class."* — Martin Fowler [Fow]

Model-View-ViewModel (MVVM) is a variant of MVC and a specialization of the design pattern *presentation model* [Fow]. MVVM abstracts view attributes into a view model, which leads to better maintainability, especially if the view needs to be changed [Webw]. Keeping values synchronized between view and view model, however, requires view data binding. Although it could be created manually, a binding library is usually employed to facilitate development [MSDN] [Lik].

Model Consists of data that are stored by the application and independent of the user interface [Webq]. In MVVM, however, the model should contain the whole business logic [Webt].

View Contains all graphical elements that represent the state of the view model. It also passes user input to the view model and requires only a small amount of code if a suitable binding library is used [Webq].

ViewModel Comprises logic to convert data into a format that is suitable to be displayed to the user [Smi]. It mediates between view and model, but it doesn't have to know about the view. This makes it possible to drastically change the view if necessary [Webq].

This time we use Knockout[2], a standalone JavaScript implementation of MVVM, to demonstrate, how MVVM is used in web development.

```
1 <p>First name: <input data-bind="value: firstName" /></p>
2 <p>Last name: <input data-bind="value: lastName" /></p>
3 <h2>Hello, <span data-bind="text: fullName"></span>!</h2>
```

Listing 2.4: Defining a View Component in Knockout.js [Wikib]

The view component in listing 2.4 is defined with HTML. The `input` fields for the first and last name are annotated with HTML 5 custom data attributes [Bew], specifying a binding declaratively. In lines 1 and 2 it signifies that the `input`'s value and the model attributes `firstName` or `lastName` respectively should automatically be synchronized.

```
1 function ViewModel() {
2   this.firstName = ko.observable("Johannes")
3   this.lastName = ko.observable("Rummel")
4   this.fullName = ko.computed(function () {
5     return this.firstName() + " " + this.lastName()
6 }, this)}
7 ko.applyBindings(new ViewModel())
```

Listing 2.5: Defining a ViewModel Component in Knockout.js [Wikib]

[2] http://knockoutjs.com/

In this simple example there is no separate model along with the viewmodel from listing 2.5. Apart from storing first and last name in lines 2 and 3, it also concatenates them to provide an additional attribute `fullName` in line 4. `fullName` is updated as soon as either `firstName` or `lastName` changes. Since `fullName` is bound to the `span`'s text from listing 2.4 in line 3, this application will instantly display the concatenated name to the user as soon as he or she types it in. To get a better idea of what this looks like, figure 2.2 shows the code rendered in a web browser.

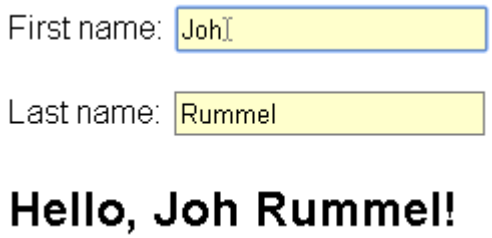


Figure 2.2.: View Data Binding with Knockout

2.1.3. Model-View-Controller / Component Tree (MVC/CT)

Model-View-Controller / Component Tree (MVC/CT) [Engb] is an architectural pattern that is related to Presentation-Abstraction-Control (PAC) [Cou87] and combines aspects of MVC and MVVM. It is similar to PAC, since both resemble the inherent hierarchical nature of user interfaces by organizing their components into a tree structure. Every component that is a node of that tree is structured using MVC, but still differentiates between a business and presentation model as in MVVM.

A reference architecture exists for MVC/CT [Engc] called *user interface component architecture*. It shows which aspects can be and as part of what component,i.e. Model, View or Controller, they have to be implemented. It also compares this distribution to other architectural approaches and thereby clarifies the difference between MVC/CT and PAC. PAC places no logic into the view component apart from mask rendering, while MVC/CT is intended to put all view data binding related code there, which in contrast is implemented as part of the controller in PAC.

MVC/CT uses a tree of user interface elements. It serves as a communication infrastructure between those elements and has the benefit of a simple event propagation mechanism [Vaa13]. Among many other advantages, splitting user interfaces into manageable chunks also fosters reusability and maintainability of single components. Figure 2.3 shows a rather artificial example of this by instantiating a login widget multiple times inside one parent dialog[3].

[3] http://componentjs.com/demo/

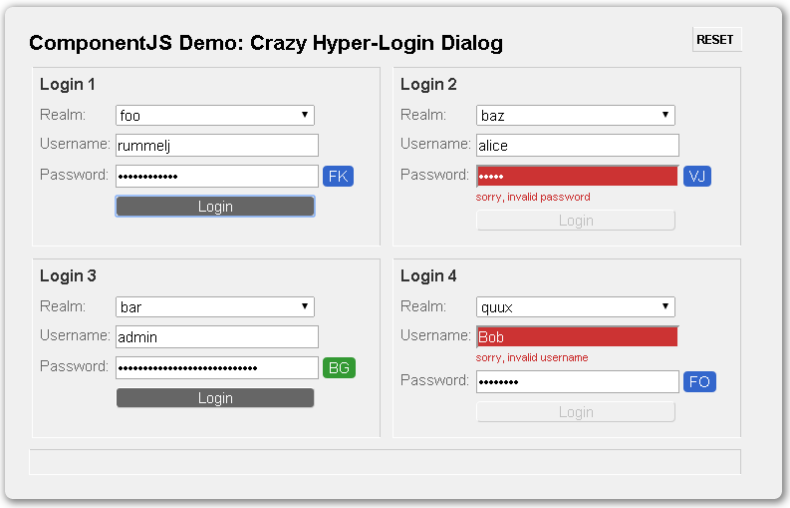


Figure 2.3.: Artificial Example of Reusing a Login Component in ComponentJS

Types of View Data Binding

It is of special interest for this work that the reference architecture differentiates between four types of view data binding [Engc].

Data Binding Synchronizes data that are meant to be displayed to the user. The binding may be two-directional, so that the data value can be either changed by the user or programmatically. In figure 2.3 all realm, username and password values are *data bound* to the model.

State Binding Handles binding of states, meaning data that have finitely many values, including Boolean or enumeration values. Examples of state binding include whether buttons are active or validation results. State bindings in figure 2.3 include the login button state or the validity of the password.

Event Binding An event in this context means an action triggered by the user. Binding to an event in ComponentJS sets a Boolean model value, a so-called event value, to `true` whenever the event is fired. Resetting this value to `false` is then handled by the framework. Examples of this in figure 2.3 are clicking the login button or selecting another realm.

Command Binding Commands are specific events whose directions are opposite to those described previously. Commands propagate from model to view. A command binding will trigger a change to the view whenever a Boolean model attribute, a so-called command value, is set to `true`. The view, which observes this command value through the command binding reacts to the change and it is set to `false` by the framework. In figure 2.3 an example of this is clicking the reset button on the top right. This will first trigger an event to the model. The controller then sends a command back to the view that it should reset its input fields to their defaults.

The differentiation helped us to find many use cases for view data binding. It is also a good practice to denote the type of a model value as a prefix. This improves the understandability and maintainability of code, since how the model is used is directly visible.

ComponentJS

ComponentJS is a JavaScript implementation of MVC/CT and its associated reference architecture. Since it is widely used at msg systems AG, one of our main goals was to be compatible with it. Although we want to ensure that our concepts and implementation are flexible enough to work well with any kind of framework or pattern, being compatible with ComponentJS shows that our approach is capable of dealing with hierarchical models. In chapter 3 we show that almost all available solutions lack this feature.

2.2. Manual View Data Binding

In this section we present one of the problems we want to solve with our approach. Starting off with a very basic example we move on to more complex problems. We demonstrate, where view data binding occurs and how difficult its implementation without using automation is. We are using a JavaScript like pseudo code and HTML throughout these examples, but our approach is not limited to web development and as you will see in chapter 3 and 5, the problem domain is much larger.

A most basic example, where view data binding comes into play, is a form where the user is supposed to enter data. To keep it even more simple, there is only a single text box for the username.

```
1 <div id="foo">
2   <input id="username" type="text" />
3 </div>
```

Listing 2.6: HTML Markup — An Introductory Example

In addition to the markup, defined in listing 2.6, a model is required. It stores the `username` and is defined with a plain JSON object in listing 2.7.

```
1 var model = new Model({
2   user: { name: "Johannes" }
3 }
```

Listing 2.7: Introductory Example: Model

Now a binding between the value of the `input` text box and the model attribute `user.name` needs to be established. We want this binding to be two-directional. First, if the user enters his or her name, the model attribute is automatically updated. Second, if the model attribute is changed programmatically, the value of the text box is also updated. This means that both values are always equal, no matter which end is changed. To acquire that functionality without using a binding library, the code shown in listing 2.8 has to be added.

```
1  // Executed when page loads
2  function init() {
3    // Updates view if model changes
4    model.user.name.observe(
5      function (newName, oldName) {
6        jQuery("#username").val(newName)
7    })
8
9    // Updates model if view changes
10   jQuery("#username").change(
11     function () {
12       model.user.name = jQuery(this).val()
13     }
14   );
15 }
```

Listing 2.8: Introductory Example: Binding

Lines 4 to 7 register a callback function that is notified whenever `user.name` in `model` changes. It then updates the value displayed to the user in line 6. Analogously, lines 10 to 14 realize the other direction of the binding. They register a callback that sets the model attribute, whenever the user changes the value inside the text box.

Even though this is a very basic example, the amount of code required is already significant. If now more `input` fields were added, the number of lines required to achieve the same view data binding would increase drastically with a lot of repetition.

2.2.1. TodoMVC

TodoMVC[4] is a project whose goal it is to help developers select one of the many JavaScript frameworks that implement a Model-View-* pattern. Its functionality is simple, but diverse enough, to cover many typical requirements of web applications. Since many aspects of TodoMVC require view data binding, it serves as a good example to demonstrate the difficulties of implementing them. Figure 2.4 shows one implementation of TodoMVC. Looking at selected requirements in the specification [Weba] illustrates where view data binding comes into play.

- Adding or filtering items updates the list of items and the counter of items remaining. This means that there must be code that keeps a collection of items from the model synchronized with the displayed list of items in the view. An efficient implementation for this not only requires being able to construct one list item, but also needs to surgically remove or add items at specific positions without touching unchanged items.
- Items can be completed or open. This state has to be reflected in the model and has multiple implications inside the markup. The tick in front of the item and its text change their color and the text gets struck through. To realize this efficiently the item that was completed or opened needs to be identified within the collection of items to prevent manipulating other items unnecessarily.

[4] http://todomvc.com/

- Item texts are editable and their label needs to be updated in the model accordingly. As explained before, this requires a similar logic to identify specific items, but has to be rewritten again.
- If no items are present, the footer should be hidden. This means that parts of the mask appear or disappear based on a condition. Thus, the condition needs to be observed for changes. If the footer is truly removed and not only made invisible, its markup needs to be stored internally, since otherwise it would be impossible to display it again later.

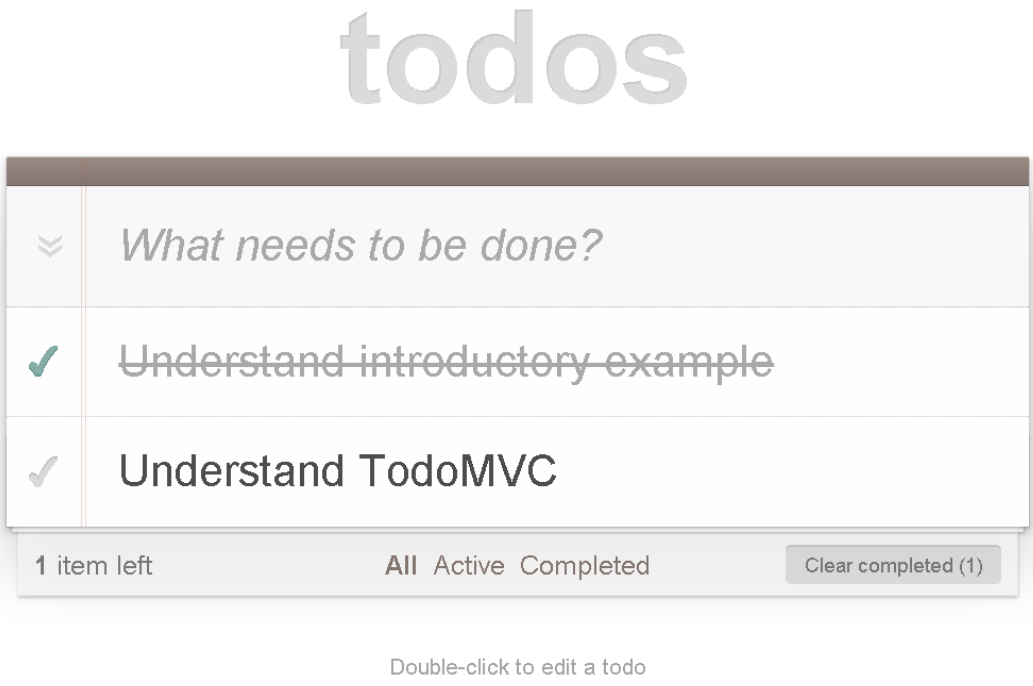


Figure 2.4.: TodoMVC Implementation

There is an implementation of TodoMVC, using JavaScript without additional libraries or frameworks. By the time of writing, this solution comprises more than 1000 lines of code. However this huge amount of code does only implement an inefficient solution without the aforementioned surgical updates. Listing A.1 in the appendix shows a simplified version of code that is responsible for displaying the list of todo items. It is called whenever an item is added, changed or removed. It shows clearly that not only the amount of code required is unproportional, but also the responsiveness and usability of the application is severely endangered if the number of items increases.

2.2.2. msg TimeSheet

The last and most complex example is a single page application used at msg systems AG for tracking working hours. It offers a seamless user interface for desktop and mobile clients by exploiting the potential of MVC/CT (see section 2.1.3). Only view modules are exchanged, while reusing all other components for both modes. The application is of special importance, since it is used in chapter 7 to evaluate our approach.

Since msg TimeSheet is implemented with ComponentJS, its user interface (UI) is structured into composites and widgets. These terms both describe UI fragments and follow a strict ontology.

Composite A Composite is a high-level UI fragment. It can either be a *Panel*, which orchestrates multiple contained UI fragments, or a *Dialog*, which interacts with the user through contained *Widgets* [Engd].

Widget A Widget is a mid-level UI fragment. It can either be a *Container* that mainly logically groups other UI fragments, a user interacting *Control* or a non-interacting *Visual*, which just displays textual or graphical content [Engd].

Figure 2.5 shows the main view of msg TimeSheet with a spreadsheet of time bookings. We select one of the many widgets and examine it more closely. The *list* widget is used in various places of the application. For example it is included in figure 2.5 on the right as a list of the most used entries, making them easier accessible.

```
1  var listModel = new Model({
2    "data:entryList" : [],
3    "command:render" : false,
4    "event:entrySelected" : false,
5    "event:entryDeleted" : false
6  });
7
8  var entryModel = new Model({
9    "data:id": 0,
10   "data:content": "",
11   "state:disabled": false,
12 });
```

Listing 2.9: Model of *list* Widget in msg TimeSheet (Simplified)

Listing 2.9 shows a simplified version of the *list* model. All attributes are prefixed with their corresponding type as explained in section 2.1.3.

The apposite markup is shown in listing 2.10. It uses jQuery Markup to ease its initial rendering. Places like in lines 4 or 5, where data are plugged in on initialization, are marked with a double mustache notation. There is also one type of binding that is invisible inside the markup. The existence of the `div` element from lines 6 to 8 is dependent on the value of `state:disabled` in `entryModel`.

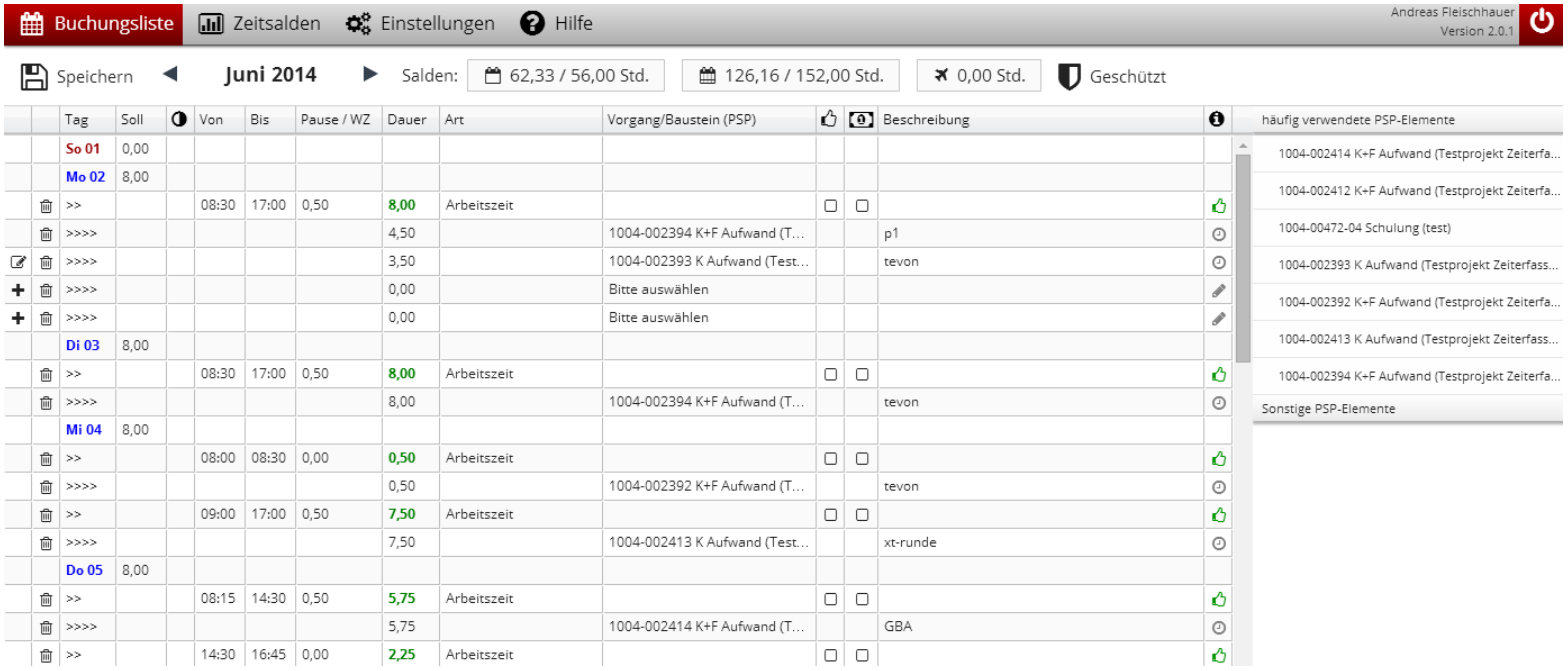


Figure 2.5.: msg TimeSheet, Main View

```
1  <markup id="widget-list">
2    <ul class="widget-list">
3      <markup id="entry">
4        <li class="{{class}}" data-id="{{id}}" title="{{content}}">
5          {{content}}
6          <div class="stateDisabled">
7            <div class="button locked fa fa-lock"></div>
8          </div>
9        </li>
10     </markup>
11   </ul>
12 </markup>
```

Listing 2.10: View Mask of *list* Widget in msg TimeSheet (Simplified)

Although jQuery Markup is used, the code for initially rendering the view component requires more than 50 lines of code. However, this does not include any updates similar to what we have shown in section 2.2.1. Change operations on the list are realized in an inefficient way by rendering the whole list again after every change.

msg TimeSheet is a rather big business application, but with around ten composites and widgets each, its size is not exceptional. Writing the code necessary to equip 20 such components with reactive view binding support is not only daunting due to the amount of repetition, but also error-prone and inefficient.

2.3. Reactive Programming

We strive for a solution that realizes view data binding in a reactive fashion. To see what that means, we explain Reactive Programming, which benefits it offers and which features a reactive application requires.

Data Flows versus Imperative Assignments

At first, Reactive Programming is a paradigm that focuses on expressing static or dynamic data flows, while hiding the underlying execution model, which automatically propagates changes to the data [Wikid]. Data flows have different semantics than assignments that are used in imperative programming. It is mostly difficult to express reactive data flows in imperative languages without the help of additional libraries.

Consider the simple example from listing 2.11 [Sto]. It keeps track of the temperature in both degrees Celsius and Fahrenheit. In imperative programming updating `celsius` in line 5 has no effect on the value of `fahrenheit`. If the assignment in line 2 however was interpreted as an expression, it would create a relationship between `celsius` and `fahrenheit`, resulting in an automatic update to `fahrenheit`. Another example for such a relationship is well known from Microsoft Excel formulas that update as soon as any referenced cell changes [We].

```
1  var celsius = 10;
2  var fahrenheit = (9/5) * celsius + 32; // = 50
3
4  // Sudden rise in temperature
5  celsius = 30;
6
7  if (fahrenheit == 50) {
8    // fahrenheit did not update, which
9    // is a typical interpretation of line 2
10   // in imperative programming
11 } else if (fahrenheit == 86) {
12   // fahrenheit was updated along
13   // with celsius, which is a reactive
14   // interpretation of line 2
15 }
```

Listing 2.11: Use Case for Reactive Programming

The Reactive Manifesto

The reactive manifesto[5] is a document that wants to raise awareness for the need to adopt the ideas of reactive programming. It does so by explaining the characteristics of a reactive application and how to achieve them.

Event-driven Updates need to be propagated asynchronously using events to avoid blocking the application. Since this implies loose coupling, the implementation is easier to extend, evolve and maintain [Rm].

Scalable Asynchronous message-passing and the event-driven approach are the prerequisites for a scalable application that allows to easily add or remove nodes [Rm].

Resilient Key to an architecture that is able to recover from errors is the isolation of components (e.g. by using the bulkhead pattern [McC]) to prevent cascading failures [Nyg]. This is ensured by the aforementioned event-driven design [Rm].

Responsive This feature is most important to the user. Responsive means that the application is always ready to handle user input and provide an interactive experience, while eliminating unnecessary wait times. This is fostered by the event-driven approach and can be ensured even under the presence of failure by being resilient [Rm].

[5] http://www.reactivemanifesto.org/

2.4. Motivation

All previous sections constitute vital prerequisites for understanding what problems we deal with and how we want to solve them.

Simplifying View Data Binding

We have shown in section 1.2 that most time for development should be spent as close as possible to the domain service layer and not on recurring tasks like view data binding. By demonstrating the various problems that arise if view data binding is implemented manually we have shown that this is currently not the case and a solution automating view data binding is needed.

In the various examples we gave an impression of how much lengthy boilerplate code is required to carry out typical view data binding requirements of an application. Long and repetitive code with a lot of redundancy not only requires an unproportional amount of time to write, but also poses maintenance difficulties and the danger of implementation errors.

One especially troublesome programming error involves observers that are used in view data binding. If they are not cleaned up correctly, memory leaks occur [Hua].

We used TodoMVC and the list widget in msg TimeSheet to show that if collections are bound, new challenges in complexity arise. It is then often unreasonable to implement view data binding in such a way that the view is only surgically changed as much as necessary. Mostly inefficient solutions that do not avoid big updates of the view are preferred. We want to reduce the amount of code required to implement view data binding. Also we need to cut down the opportunities to produce errors to a minimum. In addition, we want to implicitly prevent memory leaks and offer an efficient binding of collections with surgical updates.

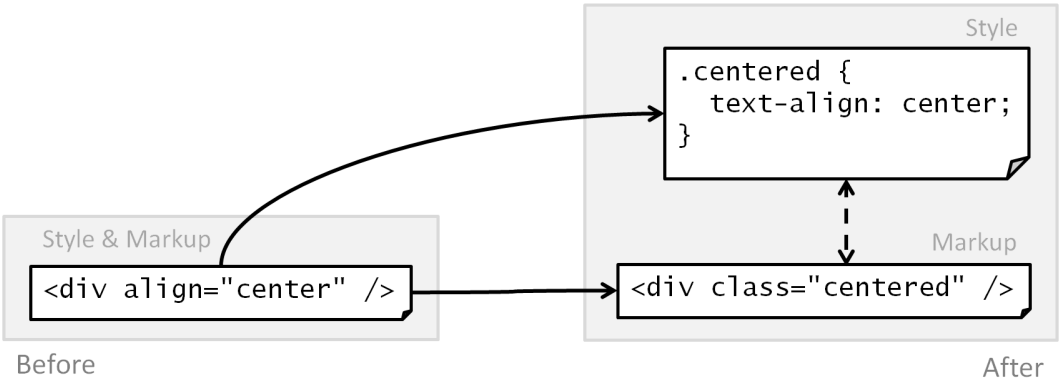


Figure 2.6.: Separation of Concerns Introduced by Cascading Style Sheets

Promoting Separation of Concerns

Separation of Concerns is one of the most, if not the most important design principle in computer science [Mak] [Cor] [Greb]. View data binding is an individual concern, but currently has no designated habitat. Therefore we want to provide a way to specify view data binding separately.

Comparable to that are Cascading Style Sheets (CSS), which were invented in the early years of the internet around 1994 [Lie94]. Prior to them it was common to mix style and markup information inside HTML, which is now strongly discouraged and all styling information is put in a separate CSS file as shown in figure 2.6. Similarly bindings and markup are currently mixed together and that is a violation of Separation of Concerns. By restoring this principle we further want to increase maintainability [Greb].

Enabling Flexibility

We presented MVC, MVVM and MVC/CT in sections 2.1.1, 2.1.2 and 2.1.3, which are all widely used in web development and implemented by many different frameworks like Backbone.js, Knockout or ComponentJS. We want to be open to and compatible with all kinds of model implementations, including the hierarchical MVC/CT.

As we show in chapter 3, many frameworks available do not concentrate only on view data binding and require to structure a whole application according to the framework. We, however, do not wish to impose such requirements and want to focus on view data binding, while being as adaptable as possible to existing solutions. Our goal is that in any solution our approach can replace just the view data binding part of the code, while having minimal implications for the remaining application.

Offering a Reactive User Experience

Section 2.3 introduced reactive programming and the benefits it offers. We want users of our approach to be able to make their applications reactive. The internal implementation of our concepts needs to be asynchronous where possible and must reduce unnecessary waiting times to yield a responsive application.

3. Existing Solutions for View Data Binding

This thesis is about presenting an approach for solving the view data binding problem illustrated in the previous chapter. With such a relevant topic we apparently cannot be the first to solve it. Therefore various existing solutions are examined and demonstrated with examples. We conclude this chapter with a comparison of libraries that are most similar to our implementation and point out what we learned from them and how our approach is different.

3.1. Solutions targeting Desktop and Thin Web Clients

View data binding is not limited to rich web clients and was already an important issue when desktop applications and thin web clients were popular. This is not surprising, since as demonstrated in section 1.2 the three differ only in their distribution of layers to the server or the client. An extreme certainly is a desktop application, where one machine fills both the server and client role and executes all layers.

3.1.1. Eclipse Rich Client Platform (RCP)

The Eclipse Rich Client Platform (RCP) is an open tools platform that is used primarily to build integrated development environments (IDEs). In theory it can be used to build any kind of client application [Webv]. Probably the best-known RCP application is the Eclipse IDE (see figure 3.1). It can be used to develop programs in Java and there are many adaptions for other programming languages. Apart from the Eclipse IDE, there are numerous products which have been built on top of RCP including reporting software, customer relationship management and even tools for e-learning [Webc].

```
1 public void setUpBinding(IObservableValue<Boolean> modelValue,
2                          ObservableValue<String> targetValue,
3                          IBidiConverter<Boolean, String> converter,
4                          IValidator<String> validator) validator) {
5   Bind.twoWay(modelValue)
6       .convert(converter)
7       .validate(validator)
8       .to(targetValue);
9 }
```

Listing 3.1: JFace Binding API

In RCP the JFace library is commonly used to simplify view data binding [Vog]. In figure 3.1 a use case for JFace can be seen. The Eclipse IDE user interface (UI) supports developers with navigating code by presenting an outline of all available methods in the current file on the right. If an entry in the outline is clicked, the code in the center scrolls to the correct position and highlights the corresponding method. The same applies to the other direction, so that moving the cursor to a method declaration in the code will simultaneously select the appropriate entry from the outline list. Assuming that this is realized by using a model attribute storing the currently selected method, this information needs to be bound to those two locations in the UI mentioned earlier.

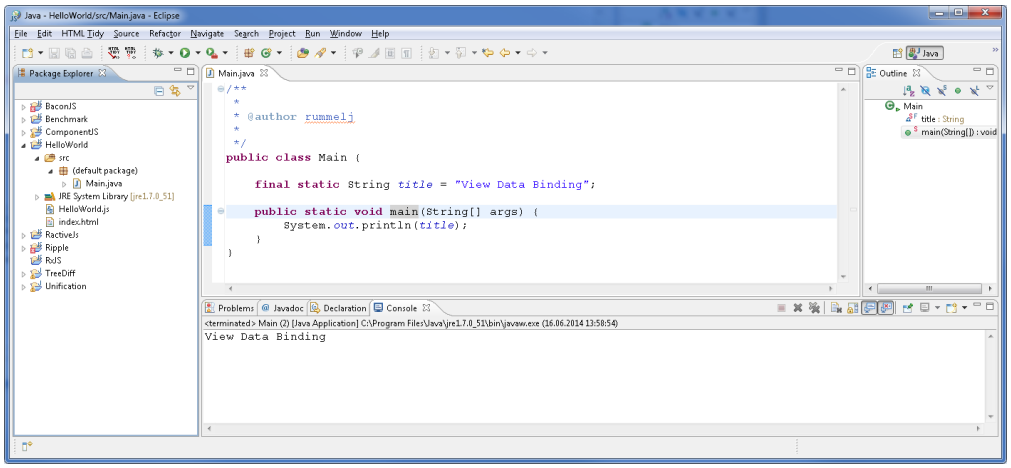


Figure 3.1.: Eclipse IDE, an RCP Application

JFace requires that all attributes that participate in view data binding implement an interface so that they are turned into observable values. With that it is easy to define bindings using a legible, imperative syntax [Webl] as shown in listing 3.1. The example shows a two-way binding between a Boolean model attribute and a string value inside the view. A validation is chained into any propagation of `targetValue` to `modelValue` in line 7. Since `targetValue` is of type `String`, its validator also needs to be an instance of the generic `IValidator` interface working on `Strings` (line 4). The converter in lines 3 and 6 converts both `modelValue` and `targetValue`. It therefore needs to implement the two methods `modelToTarget` and `targetToModel`. The first produces a `String` by consuming a `Boolean` and the second does the exact opposite.

3.1.2. JavaServer Faces (JSF)

JavaServer Faces (JSF) is a technology for building server-side user interfaces [Webk]. It is typically used as the view component in an MVC architecture (see section 2.1.1) together with managed beans as the model [Webaa]. Managed beans store application data and may also perform validation, event handling and routing related tasks [Weby]. Their attributes are referenced with an expression language inside the view component [Webz]. One task that is taken over by JSF is to keep the displayed values of those referenced attributes synchronized with model attributes, no matter if they are changed in the view on the client or the model on the server.

In such a thin client architecture, after a change to the model the view cannot be updated without reloading it. However, this is usually undesirable, since it destroys the interactive feeling of an application. Fortunately, JSF offers a solution to this by employing AJAX to perform asynchronous communication with the server.

```
1  <html xmlns:f="http://java.sun.com/jsf/core"
2        xmlns:h="http://java.sun.com/jsf/html">
3    <h:body>
4      <h:form>
5        <h:inputText id="name" value="#{model.name}" />
6        <h:commandButton value="Display name">
7          <f:ajax execute="name" render="output" />
8        </h:commandButton>
9
10        <h:outputText id="output" value="#{model.name}" />
11     </h:form>
12   </h:body>
13 </html>
```

Listing 3.2: JSF, View Component

Listing 3.2 shows JSF markup that renders a view including a text box (line 5), a button (lines 6 to 8) and a label (line 10). By using the expression language of JSF a model attribute `name` is both bound to the value of the text box and the caption of the label (lines 5 and 10). A click on the button causes the view data binding mechanism in JSF to synchronize the attribute `name` and to render the element with id `output` again (lines 6 to 8).

```
1  @javax.faces.bean.ManagedBean
2  @javax.faces.bean.SessionScoped
3  public class Model implements java.io.Serializable {
4
5          private static final long serialVersionUID = 1L;
6
7          private String name;
8
9          public String getName() {
10            return name;
11         }
12
13         public void setName(String name) {
14            this.name = name;
15         }
16 }
```

Listing 3.3: JSF, Model Component

A model on the server containing an attribute `name` is required to complete the example (see listing 3.3). Since JSF needs a managed bean, an annotation is added to a standard Java class (line 1). If in addition the class follows the JavaBean conventions, this is already sufficient. Those conventions demand that the class is serializable (lines 3 and 5) and every attribute is accessible through a getter (lines 9 to 11) and a setter (lines 13 to 15). By convention `#model.name` (see listing 3.2, line 5 or 10) then resolves to `model.getName()`.

Figure 3.2.: HTML, Rendered by JSF

Figure 3.2 shows the example in action after deploying it onto a suitable web application server. If the user enters something into the text box and clicks the button, the label to the right of the button updates after the asynchronous AJAX call to the server returns. Although invisible to the user, not only the label, but also the model attribute on the server then contains the value entered into the text box.

3.1.3. Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) is a technology for rendering user interfaces in windows-based applications [Webj]. It uses XAML, an extension of XML to describe user interfaces and to define view data binding declaratively.

Concepts

Each binding in WPF consists of four components. The MSDN explains them with the example of binding a text box to an employee's name [Webd].

Binding Target Object
: The element targeted by the binding. In this case the text box.

Target Property
: The attribute of the binding target object that is bound. This is the text of the text box.

Binding Source
: The counterpart of the binding target object in the model. Here it is an `Employee` object.

Path to Value
: The address to the bound attribute inside the Binding Source analogous to the target property. In this case this would typically be the `Name` property of the `Employee`.

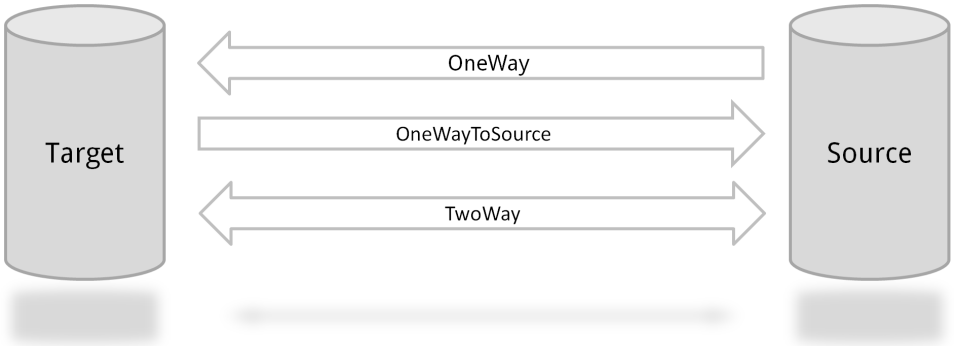


Figure 3.3.: View Data Binding Flows in Windows Presentation Foundation

In addition, WPF differentiates between four types of data binding. They differ in the direction of the data flow and in how long they remain active (see figure 3.3) [Webd].

OneWay
Data flows from source to target only. If the source is changed, the target will be updated, but the source remains unchanged if the target is changed. This type of binding is useful if non-editable model attributes are displayed to the user.

OneWayToSource
Opposite of OneWay. The source is updated if the target changes, but there is no flow from source to target. This is typically used when user input needs to be stored in the model without being displayed again.

TwoWay
Combination of the OneWay and OneWayToSource types. No matter which end of the binding is changed, the other end will be updated accordingly. A regular use case for this is editable data that are changed either in the model or in the view by the user. For example, a change to the model happens if there is a change to the underlying database.

OneTime
OneWay binding that is only executed once. The target is initialized with the source value only one time and subsequent changes to the source have no effect on the target.

Another interesting concept is binding triggers. Both *TwoWay* and *OneWayToSource* have a property called `UpdateSourceTrigger`. It configures which event triggered by the target causes an update to the source. A text box, for example, can be configured so that updates are propagated only when it loses focus or after each key stroke [Webd]. However, the concept lacks a pendant like `UpdateTargetTrigger` for the opposite direction [Webab] and does not allow registering multiple binding triggers for one binding.

Features and Example

WPF allows a declarative and imperative definition of binding and plugging in value converters or components for validation. In addition, it is possible to bind collections which can be sorted, filtered or grouped. To debug a binding it is possible to adapt a log level, so that useful information about the internal execution becomes visible. These features are demonstrated with a bidding platform (see figure A.1 in the appendix) [Webd]. It lists items for sale and allows adding and bidding on items.

The list of items is bound to a collection in the model. Its elements are individually styled and grouped. Grouping, Filtering and Sorting can be dynamically turned on or off by using three check boxes. If an item is selected, the corresponding attributes are bound to captions of labels in a description section.

Finally, the use of a converter is demonstrated with the attribute that stores how long a user already has been a member. This attribute is stored as a Unix Timestamp in the model, which is illegible to the user. Therefore, it is converted to a human readable format before being displayed in the view.

```
1  <Application>
2    <Application.Resources>
3      <src:DateConverter x:Key="dateConverter"/>
4      <DataTemplate DataType="{x:Type src:AuctionItem}">
5        <!-- Binding is defined here -->
6        <TextBlock
7            Text="{Binding Path=StartDate,
8                   Converter={StaticResource dateConverter}}"/>
9      </DataTemplate>
10   </Application.Resources>
11 </Application>
```

Listing 3.4: Binding a Text Box with a Converter using simplified XAML in WPF

Listing 3.4 shows how to define such a binding in WPF. In line 3 the aforementioned converter is registered under the name `dateConverter`. In line 4 the *Binding Source* is configured to be `AuctionItem`. Most relevant are lines 6 to 8, where the binding is established with an expression inside the `Text` (*Target Property*) attribute of a `TextBlock` (*Target Object*). Finally, the expression defines the missing *Path to Value* in `AuctionItem` to be `StartDate` and plugs in the `dateConverter` converter.

3.2. Libraries for Rich Web Clients

This section moves the focus towards solutions for rich web clients that are more similar to our implementation than those for desktop and thin clients. With Facebook React, KnockoutJS and Ractive.js we are examining three of the most popular libraries currently available for view data binding. To avoid giving the impression that there are no other solutions available, we conclude this section by presenting five more libraries briefly and make clear that even more can be easily found.

3.2.1. Facebook React

Facebook React is a JavaScript library that was developed and made publicly available by the social network facebook. It aims to ease the definition of modular and composable components in web pages by offering its own markup syntax called JSX. The reason why this library deserves a closer look is its popularity, its implicit approach on view data binding and the fact that it has similar goals in mind, such as reducing boilerplate code and making web pages more performant by employing surgical updates [Webf]. A component in Facebook React usually consists of two things, a template and an internal state. Whenever the model which is represented by the internal state changes, the library decides how to update the DOM that is an instantiation of the template. To simplify the composition of components, each of them can be defined in a separate file [Webg].

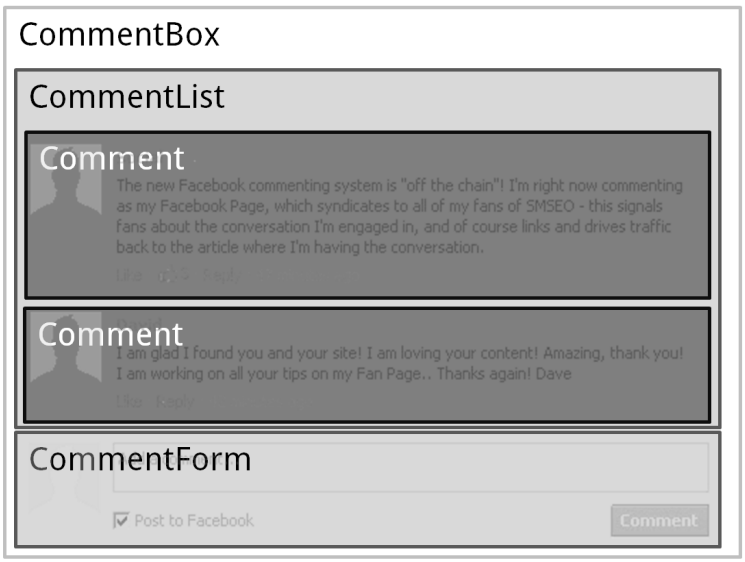


Figure 3.4.: Facebook React Comment Box — Component Structure

Facebook React attempts to increase the performance of web pages by automatically applying event delegation and by using an algorithm called reconciliation. The idea of event delegation is to reduce the number of event listeners by attaching a single event listener to an ancestor element instead of attaching multiple listeners directly to all elements. The easiest example of this technique is a list of elements, where each element executes a certain action when it is clicked. Instead of attaching an event listener to each element of the list, only one is added to the whole list to collect all events. On this level it is usually still possible to determine which element was clicked. One advantage of event delegation is that it increases maintainability. In the list example, the code for change operations on the list becomes a lot simpler. It needs only to modify the list itself and does not have to deal with updating event listeners, too [Wal]. Also, the technique provides looser coupling between the DOM and its associated code. Most importantly, event delegation increases performance, because it requires less memory to manage event handlers [Zak].

The second ingredient for performance in Facebook React is the reconciliation algorithm. It tries to minimize the number of operations to transform one template instantiation into another. Such a transformation is caused by an update to the underlying model [Webe], usually a list.

To demonstrate how these concepts of Facebook React can be used, we want to go over a simplified version of a comment box example adapted from a tutorial that is part of the official documentation [Webh]. The comment box is broken down into four components: a comment box acting as a container, a list of comments, a single comment, and a comment form that allows to add a comment (see figure 3.4).

```
1  var CommentBox = React.createClass({
2    getInitialState: function () { return {data: []}; },
3    handleCommentSubmit: function(comment) {
4      this.state.data.push(comment);
5      this.setState({data: this.state.data});
6    },
7
8    render: function () {
9      return (
10       <div>
11         <CommentList data={this.state.data} />
12         <CommentForm onCommentSubmit={this.handleCommentSubmit} />
13       </div>
14     );
15   }
16 });
17
18 var CommentList = React.createClass({/*...*/});
19 var CommentForm = React.createClass({/* ... */});
20 var Comment = React.createClass({/* ... */});
21
22 React.renderComponent(<CommentBox />, $('#container'));
```

Listing 3.5: Comment Box with Facebook React

Listing 3.5 shows how each component is defined by using the library. In line 1 the surrounding comment box is defined. Lines 18 to 20 are there only to indicate that all other components would be defined similarly. One might expect that not the `CommentBox` component but the `CommentList` should manage the list of comments. However, this would require that `CommentForm` manipulated a sibling in the component tree. Nevertheless, `CommentList` needs the list of comments, and this is resolved by explicitly binding it as a parameter to `CommentList` in line 11. Lines 10 to 13 are written in JSX and implicitly establish a binding whenever curly braces are used.

The `CommentForm` component in line 12, on the other hand, receives a reference to the `handleCommentSubmit` function and will use it as a callback whenever a comment is added. `CommentBox` in turn adds the comment to its internal state and Facebook React propagates the change to `CommentList`. Line 22, finally, plugs the `CommentBox` into an element from the DOM with an id of `container`.

3.2.2. Ractive.js

Ractive.js is a template-driven user interface library written in JavaScript [Webu]. It offers a broad set of functionality including two-way binding, an expression language for calculations or formatting and even animations. It was originally developed for the web presence of *The Guardian*[1] newspaper. Therefore, its primary goals were that it be easy to develop, work reliably across multiple browsers and perform well on mobile devices [Webu]. Ractive.js uses a double mustache notation to define a binding between a template and a model. Although the syntax is very simple, it is possible to express conditions and iterate over collections.

```
1 <div id="template" />
2   <ul>
3     {{#items}}
4       <li>{{itemLabel}}</li>
5     {{/items}}
6   </ul>
7   <input type="text" value="{{newLabel}}" />
8   <button onclick="addItem">Add Item</button>
9 </div>
```

Listing 3.6: Ractive.js Example Template

To put Ractive.js into action, a template like the one in listing 3.6 is required. It displays an unordered list of items (lines 2 to 6). Every item consists of a label `itemLabel` (line 4). New items can be added by entering their label in a text box (line 7) and clicking a button (line 8). The example executed in a web browser is illustrated in figure 3.5.

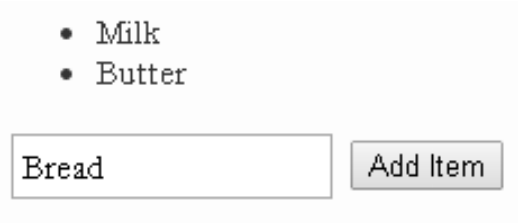


Figure 3.5.: Ractive.js Item List

The code bringing this template to life is shown in listing 3.7. It first declares the model comprising only the list of items (line 1), which is then passed into the library (line 6). To set up Ractive.js it needs a reference to the template (line 5) and the element, where the result should be appended to (line 4). `items` was declared globally to be also accessible when handling the click on the button by the user (lines 9 to 11). By overwriting the built-in JavaScript methods for collection modification including `push` (line 10), Ractive.js recognizes that the model has changed and updates the output accordingly.

[1] http://www.theguardian.com/uk

```
1  var items = []
2
3  ractive = new Ractive({
4    el: $("#output"),
5    template: $("#template"),
6    data: { items: items }
7  });
8
9  ractive.on("addItem", function (event) {
10   items.push({ itemLabel: ractive.get("newLabel") })
11 });
```

Listing 3.7: Ractive.js Example Code

3.2.3. KnockoutJS

In section 2.1.2 we have already shown how KnockoutJS can be used to implement simple view data binding in the Model-View-ViewModel pattern. In this section, we want to go into more detail about the view data binding support of KnockoutJS.

The library uses its own binding syntax, which adds attributes to the template in a declarative manner. A binding consists of a name and a value, which are separated by a colon. The name has to match a registered binding handler or has to be a parameter [Webn]. Built-in binding handlers include `text`, `html` and `attr` to bind to an elements text, inner HTML and attributes. In addition, there are special handlers `css` and `style` for adding or removing style classes or attributes. To serve non-standard use cases KnockoutJS offers a mechanism to extend those handlers with custom ones [Webm].

The binding value can be a reference to a model attribute or almost any valid JavaScript expression [Webn]. This also includes JavaScript functions that can be bound to event handlers. For example, a callback function can be bound to the click event of a button. KnockoutJS furthermore supports binding to collections and uses surgical updates to efficiently refresh the view following changes to the model [Webo].

The library assigns a context to each node in the DOM. These contexts may be manually switched with a command to zoom into the model. This way the context may reference only a certain part of the model instead of the whole model. Conceptually, this is more interesting when looking at control flow structures involving iteration. The context then automatically contains attributes that allow accessing the current element of the collection that is iterated and its index [Webp]. To demonstrate the syntax of view data binding in KnockoutJS we use the same example of an extendable list of items as in section 3.2.2 (see figure 3.5).

```
1 <div id="template">
2   <ul data-bind="foreach: items">
3     <li data-bind="text: itemLabel" />
4   </ul>
5   <input type="text" data-bind="value: newLabel" />
6   <button data-bind="click: addItem">Add Item</button>
7 </div>
```

Listing 3.8: KnockoutJS Example Template

Listing 3.8 displays the template for the item list. In contrast to Ractive, Knockout uses `data-bind` attributes to specify the binding, which does not result in invalid HTML for the template [Webx]. The binding handlers used in this example are `text` (line 3), `value` (line 5) and `click` (line 6). Their corresponding binding values are `itemLabel` (line 3), `newLabel` (line 5) and the function `addItem` (line 6). Moreover, it can be seen that the context within the iteration (lines 2 to 4) switches from the whole model to the current item, so that its label can be referenced directly.

```
1  function ItemList() {
2    this.items = ko.observableArray([]);
3    this.newLabel = "defaultLabel";
4
5    var self = this;
6    this.addItem = function() {
7      // 'this' would be the 'addItem' function here
8      // instead of the 'ItemList' function
9      self.items.push({itemLabel: self.newLabel});
10   }
11 }
12
13 ko.applyBindings(new ItemList());
```

Listing 3.9: Knockout Example Code

The model for this template is presented in listing 3.9. An empty array initializes the items in the `ItemList` model (line 2). In Ractive.js the callback function for the button click did not need to reside inside the model. KnockoutJS, on the other hand, can bind only to elements that are inside the model (lines 6 to 10).

As we will see in section 3.3, Facebook React, Reactive and also KnockoutJS share the property of being tied to their model implementation. In the case of KnockoutJS this has led to the development of Knockback[2], which tries to combine both KnockoutJS and Backbone.js[3].

3.2.4. Other Libraries

Analyzing and presenting every framework that offers some sort of view data binding support would certainly exceed the scope of this chapter. We investigated and counted more than 30 libraries. The complete list can be found in section A.1.3 of the appendix. Because of the relevance of the topic, the number of libraries is rapidly increasing. Nevertheless, we want to provide an overview by introducing some additional libraries in a shorter format.

[2] http://kmalakoff.github.io/knockback/
[3] http://backbone.js.org/

RivetJS http://rivetsjs.com/
RivetJS is a very lightweight solution for view data binding. Its source code after minification is only around 20 kB uncompressed and less than 4 kB compressed with gzip. Similar to KnockoutJS, it uses DOM attributes to define the binding in a declarative manner. The library is special in that it is one of the very few that is truly agnostic to the model implementation. The model is always accessed through a component called adapter, and to support new model implementations only an appropriate adapter has to be added.

RivetJS goes even further and allows chaining different adapters. This way, it is possible to reference a backbone model that is nested inside a plain JavaScript object. Our implementation, however, does not support such an adapter chaining, because we think this would introduce too much coupling between the view and the structure of the model.

Ripple https://github.com/ripplejs/ripple
Ripple is a simple view data binding library written in JavaScript. It allows composing views similar to Facebook React but encapsulates an individual set of plug-ins into each component. Those plug-ins, for example, include iteration, event handling or computed properties.

Vue http://vuejs.org/
Vue focuses on simplicity while offering a broad set of powerful functionality. In contrast to KnockoutJS, which consistently uses the `data-bind` attribute, the library adopts different attributes to express the type of binding like `v-text` or `v-attr`. Besides, Vue supports defining bindings with a double mustache notation comparable to Ractive.js. Similar to Facebook React, it is component-oriented and follows the same idea as web components. This means that Vue.js bundles a template, its binding and a ViewModel together, which can then be referenced by its name and plugged into any other location.

MontageJS https://github.com/montagejs/frb
MontageJS is a framework for building web applications and includes much more than view data binding. A project that is part of it is Functional Reactive Bindings (FRB). FRB is unique in the sense that it makes it possible to define a binding imperatively. This means that not DOM elements are enriched with annotations; rather code statements set up a binding. With this it is the only library that theoretically allows definition of view data binding separated from the view. However, this is almost impractical, because it requires such great effort.

Additionally, not only does FRB cover binding view to model; it also offers rich functionality including higher-order functions like map or filter to bind any two values. These bindings, however, are designed to solve a more abstract problem and are almost impractical for use in rich clients.

Reactive https://github.com/component/reactive
Reactive has an API similar to Ripple but lacks its ability to compose views. Roughly speaking, Reactive is the predecessor of Ripple.

3.3. Comparison of Rich Web Client Libraries

In this section the eight solutions for view data binding in rich web clients previously presented are compared. The primary goal of this comparison is to determine whether one library exists that already satisfies all needs. If this is not the case, we can learn which aspects need to be improved in a novel approach to contribute relevant work to the topic.

3.3.1. Methodology

We examined and combined more than 30 aspects including the predominant features that were named in the documentation of the libraries. After adapting this criteria catalog to cover all relevant ingredients, we broke it down into five different categories.

Design
How many options are there to define the binding? Is it necessary to violate architectural best practices when applying the solution? Is the library performant, and is its level of abstraction appropriate for the task at hand?

Compatibility
Does the solution play well with various model implementations and browsers?

Features
How rich is the set of available functionality?

Usability
How understandable and maintainable is the code for the binding? How much effort is necessary to apply the library to an existing solution?

Support
Is it easy to get information about and assistance with the library?

Reliability
How risky is it to use the solution? How much care did the developers take to provide a stable solution?

Each aspect was graded with a positive score that increases, or a negative score that decreases the overall rating by one point. There are neutral scores that do not affect the result, and in some rare cases there are aspects that can yield two points. An in-depth description of every aspect and how it was evaluated can be found in section A.1.1 of the appendix.

While there are easily assessable aspects like the size of the library in kB, there are others where the correct rating is not obvious. These scores should be understood as qualitative. However, this does not affect the results of the comparison, since it depends more on the overall quality of a library than on every detail.

3.3.2. Results and Conclusion

The accumulated scores in table A.7 suggest that KnockoutJS, followed by Ractive.js and Facebook React currently offer the best available solutions for view data binding in rich web clients. But the table also shows that there is room for improvement, especially when it comes to design (see table A.1) and compatibility (see table A.2).

Results

With their excessive expression language, both KnockoutJS and Facebook React allow putting too much business logic into the view. Specifying the binding separated from the view, which would be a more preferable design choice, is not fully supported in any framework. In addition, no solution is flexible enough to allow both an imperative and a declarative definition of the binding.

A big drawback of KnockoutJS, Facebook React and Ractive.js is that they do not focus on view data binding only and impose the use of a specific model implementation. This means that they are incompatible with various model implementations including hierarchical models. Only RivetJS receives good scores in terms of compatibility, but it is too lightweight to offer a rich set of features as other libraries (see table A.3).

In general, we liked the usability of the frameworks (see table A.4). Ractive.js and RivetJS scored the most and MontageJS the least points in this category. However, the problem of compatibility pertains to usability, because it makes it difficult to integrate the frameworks into existing solutions.

Those libraries that offer good support are more popular (see table A.5). This connection is easily explained, since new users require a good documentation to get started and as the number of users increases, it is easier to find help in communities.

Some aspects, such as maturity in the reliability section (see table A.6), cannot be influenced directly by the authors of a library. Many solutions like Ractive.js or MontageJS, however, did not convince us with their architectural and implementational cleanliness.

Conclusion

While almost all libraries we examined received a good rating, there are differences in quality and we identified room for improvement. No library offers a binding that can be defined separately and is truly model agnostic while still offering a rich set of features. Also, no solution fully addresses our idea of holisticity regarding a high degree of freedom in defining the binding.

4. Concept and View Data Binding Ontology

Before introducing the *Concepts* for realizing *View Data Binding* we give an overview of relevant terms and their relationships. This ontology will then allow us to consistently explain our *Concepts* throughout the next chapter. Whenever we refer to a term from the ontology we indicate it by putting it in *italics*.

Concept Ontology

Concepts can be either divided into *Core* and *Convenience Concepts* or into *Binding* and *Structure Concepts*. Therefore, we have four different types of *Concepts* which are *Core Binding*, *Core Structure*, *Convenience Binding* or *Convenience Structure Concepts*. The relations of the terms is illustrated in figure 4.1.

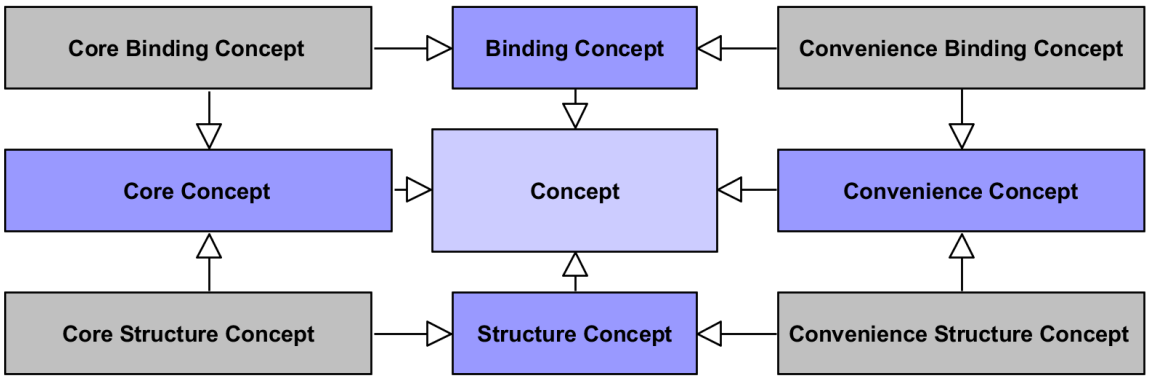


Figure 4.1.: Concept Ontology

Core versus Convenience Concepts

Core Concepts are the essential building blocks of our *View Data Binding*. They would cause a loss of expressiveness if missing. It follows naturally that a *Core Concept* can not be replaced by a combination of other *Core Concepts* and that the set of *Core Concepts* is minimal.

Convenience Concepts on the other hand are solely introduced to satisfy nonfunctional requirements like usability, readability and maintainability. They do not increase expressiveness and could be seen as *Syntactic Sugar*. We will therefore later demonstrate how each of our *Convenience Concepts* is derived from *Core Concepts*.

An intuitive example of a *Convenience Binding Concept* is *Two-Way Binding* (see section 5.3.1). Even without any knowledge about semantics it should be clear that it can be transformed into two *One-Way Bindings* (see section 5.1.2). This, however, doubles the amount of code and justifies the existence of *Two-Way Binding*.

Binding versus Structure Concepts

Concepts regarding *View Data Binding* are called *Binding Concepts*. On the other hand, *Structure Concepts* provide tools for organizing an application. Structural needs only emerge from best practices that demand component oriented applications (see sections 2.1.1, 2.1.2 and 2.1.3) and would not be relevant in a Big Ball of Mud architecture.

View Data Binding Ontology

The goal of this ontology is to give certain entities unique names. This builds a solid foundation for communication. Although we will sometimes give illustrative examples, the focus of this section does not lie on semantics, but on the terms itself and their relationships. The complex diagram of figure 4.2, which we want to elucidate with some accompanying notes, covers most of the information concerning this section.

Blue boxes represent entities and arrows or plain connections show relationships between them. The latter have multiplicities if useful and hold the semantics known from Unified Modeling Language class diagrams. As a reminder of their meanings we included a legend showing how everyday terms can be related to them. The root of the ontology, which is – apart from few exceptions – a tree, is *View Data Binding*, the main topic of this thesis.

Data Targets

The task of *View Data Binding* is to connect and synchronize *Data Targets*. In particular, these are *Presentation Model* and *View Mask*. In addition, we identified a third *Data Target* called *Binding Scope* (see section 5.1.2.3).

The *View Mask* comprises a *Template DOM Fragment*, which is – together with the *Presentation Model* and *Binding Specification* – processed to produce the *DOM Fragment*. The latter is mounted into the *View Mask* at a designated spot called *Mount Point*. The term *View Mask* typically describes an HTML page in a web environment, but could be any kind of user interface, such as the one of a desktop application.

Binding Specification

The *Binding Specification* is the syntactical representation of our *Concepts* and every entity of its subtree will later be defined with a grammar. It comprises an arbitrary number of *Blocks* which are either a *Group* (see section 5.2.1) or a *Scope* (see section 5.1.1). Each *Block* may have child *Blocks* which creates a tree of *Groups* and *Scopes*.

Every *Scope* must have at least one *Selector*, while the attributes *Iteration*, *Template*, *Insertion* are optional.

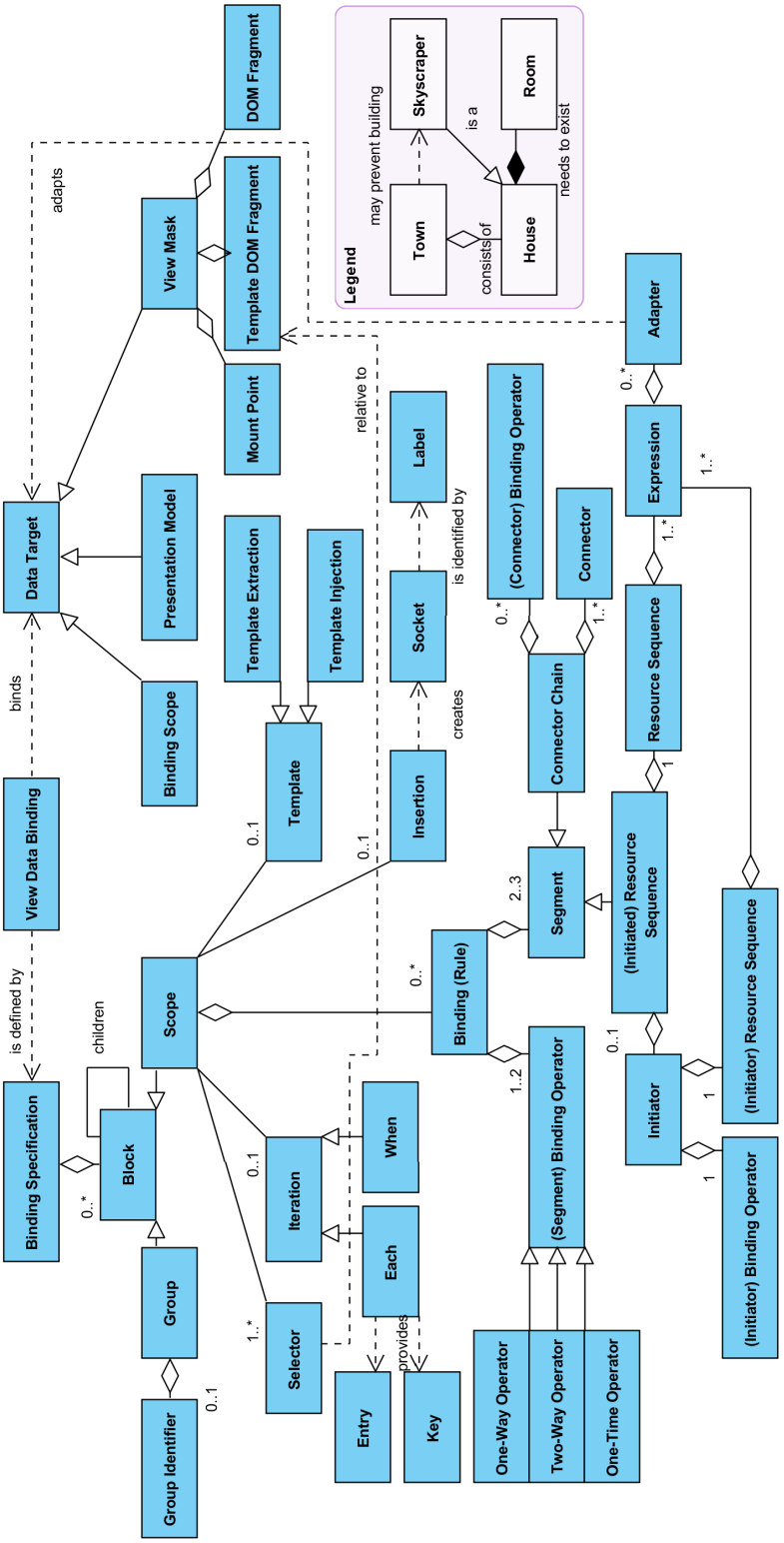


Figure 4.2.: View Data Binding Ontology

With *Iteration* elements are either repeated or conditionally shown and hidden. We call the first case *Each* and the latter *When*. In both cases *Iteration* provides the *Entry* that is currently iterated and the *Key* of that *Entry*. If *Iteration* occurs as *When*, this information is typically not required (see section 5.1.3). The realizations of *Template* are *Template Extraction* and *Injection* (see section 5.4.1). With the *Concept* of *Insertion* it is possible to create *Sockets* that are identified by a *Label* (see section 5.2.2).

Binding Rules

The actual instructions for *View Data Binding* are defined by the *Binding Rules* of a *Scope*. *Bindings* are formed of two or three *Segments*, which are separated by *Segment Binding Operators* (see section 5.1.2).In turn, *Segments* are realized by either *Initiated Resource Sequences* or *Connector Chains* which comprise at least one *Connector*. *Connectors* are separated by *Connector Binding Operators*.

An *Initiated Resource Sequences* is, as its name suggests, a combination of an optional *Initiator* and a *Resource Sequence* (see section 5.3.3). An *Initiator* is a combination of two *Resource Sequences* where one of them acts as the *Initiator* and the other is *Initiated*. The two are combined with the *Initiator Binding Operator* (see section 5.3.4).

Obviously, the shorthand names for *Segment*, *Connector* and *Initiator Binding Operator* as well as those for *Initiated*, *Initiator Resource Sequence* and *Resource Sequence* itself are overlapping. This is intentional, since it allows to talk about *Binding Operators* or *Resource Sequences* in an abstract way.

All *Resource Sequences* consist of *Expressions* (see section 5.3.6). Every *Expression* may comprise an arbitrary number of Adapter including no *Adapter* at all. The semantics of an *Adapter* are explained in section 5.1.2.1. Please note that the plural of *Adapter* is also *Adapter*.

5. View Data Binding Concepts

In this chapter we define and explain the *Concepts* which are necessary to realize *View Data Binding*. They are designed to cover the use cases we found while examining existing solutions (see chapter 3) and applications as much as possible. For each *Concept* we introduce its syntax and semantics, or, in other words, how it is denoted and what it means. In case of *Convenience Concepts* we define their semantics by explaining how they are derived from *Core Concepts*.

The syntax of our *Domain Specific Language* for *View Data Binding* is defined with Parsing Expression Grammars [For04]. These grammars are written as the quadruple $G = (V_N, V_T, R, e_S)$ of V_N, the set of non-terminal symbols, V_T, the set of terminal symbols, R, the set of rules and e_S, the start expression. To keep their representations compact we denote them just by their rules R, implicitly specifying V_N and V_T. All elements from V_N are always capitalized in R, while members of V_T are enclosed in quotation marks or denoted as regular expression. These expressions, such as `[a-z]` or `[@$%]`, refer to equivalence classes and should be interpreted as the set of strings that are matched by the given regular expression. Further, we implicitly declare the start expression e_S by coloring it in gray.

5.1. Core Binding Concepts

In this section the *Concepts* of *Selection*, *Binding* and *Iteration* are introduced. In short, *Selection* determines which elements of the *Template DOM Fragment* to bind, while *Binding* defines how these elements are bound. Last, *Iteration* allows to repeat parts of the *Template DOM Fragment*, or to conditionally show or hide them.

5.1.1. Selection

The goal of *View Data Binding* is to bind elements from the two *Data Targets*, *Template DOM Fragment* and *Presentation Model*. Therefore, we need a way to specify which element of a *Data Target* should be bound. The information that allows locating certain elements of a *Data Target* is called a *Path*. In order to succeed in this we identified five different ways. To explain them vividly the HTML from listing 5.1 serves as a *Template DOM Fragment* and the JSON object from listing 5.2 as a *Presentation Model*. The objective of the example is to bind the two `span` elements (Listing 5.1, lines 3 and 5) to the text `Lorem ipsum` (Listing 5.2, line 4).

```
<div id="template">
  <div class="foo">
    <span class="bar"></span>
  </div>
  <span class="bar"></span>
</div>
```

Listing 5.1: Template DOM Fragment

```
var model = {
  foo: {
    bar: {
      baz: "Lorem ipsum"
}}}
```

Listing 5.2: Presentation Model

The five options, where *2b* is the inverse of *2a* and *3b* the opposite of *3a*, are as follows

1. Fully qualifying both *Paths* in each *Binding*.

2a. Specifying the *Path* for the *Presentation Model* dedicated to a set of *Binding*s, so that each *Binding* only needs to specify the *Path* for the *Template DOM Fragment*.

2b. Specifying the *Path* for the *Template DOM Fragment* dedicated to a set of *Binding*s, so that each *Binding* only needs to specify the *Path* inside the *Presentation Model*.

3a. Implicitly specifying the *Path* for the *Template DOM Fragment* by adding the *Binding* to it.

3b. Implicitly specifying the *Path* for the *Presentation Model* by adding the *Binding* to it.

Listing 5.3 shows options *1*, *2a* and *2b*. Option *3a* is well known from many existing frameworks, such as KnockoutJS (See listing 3.8). We do not visualize option *3b* because it is only of theoretical character.

```
// 1.  Qualifying both paths
bind("span").in("TDF").to("foo.bar.baz").in("PM")

// 2a.  Presentation Model first
foo.bar.baz {
  bind.to("span")
}

// 2b.  Template DOM Fragment first
span {
  bind.to("foo.bar.baz")
}
```

Listing 5.3: Three Possibilities for Selection

For the sake of *Separation of Concerns* we drop both options *3a* and *3b*. They do not allow to specify a *View Data Binding* separately and define it in a place of a different concern. Option *1* is too verbose, since there is no convention about the place where each of the *Paths* is written. Thus, this information has to be attached each time.

With the remaining options *2a* and *2b* left we decided to use *2b*. That is because due to the huge popularity of CSS, which operates in a similar fashion, this option feels most natural.

Selection Syntax

The syntax for *Selection* is inspired by *less*[1], which is a CSS preprocessor, primarily increasing readability and maintainability of CSS code. Its biggest advantage over CSS is that it allows to hierarchically structure *Selectors*. From an ontology perspective we define the syntax of a *Scope* with grammar 5.1.

```
Scope ← Selectors "{" ScopeBody* "}"
Selectors ← Selector ("," Selector)*
ScopeBody ← Scope | Binding
```

Grammar 5.1: Syntax of *Scope*

The non-terminal symbols `Binding` and `Selector` remain open. `Binding` will be defined in section 5.1.2 and `Selector` stands for any valid CSS selector[2]. For reasons of shortness, we decided not to include their grammar.

Listing 5.4 shows a valid document that is produced by this grammar. All `Selector` elements are highlighted in blue.

```
1  #wrapper {
2    <Binding-1>
3    div > .input, span {
4      <Binding-2>
5    }
6    <Binding-3>
7    div {
8      div + p {
9        <Binding-4>
10     }
11     .empty {}
12   }
13   <Binding-5>
14 }
```

Listing 5.4: Selection Example

[1] http://lesscss.org/
[2] http://www.w3schools.com/cssref/css_selectors.asp

Selection Semantics

Each *Scope* has a subset of elements of the *Template DOM Fragment* which is defined by its *Selector*. We call a *Scope* that has no parent a *Root Scope*. All *Root Scopes* apply their *Selector* to the whole *Template DOM Fragment*. The matched set becomes the subset of elements of that *Root Scope*. Any other *Scope* uses the descendants of elements from its parent's *Scope* as its context for applying the *Selector*. Listing 5.6 shows pseudo JavaScript code that demonstrates how the set of matched elements for each *Scope* is determined.

We want to explain this algorithm with an example, use the *Scope* definition from listing 5.4 and add the missing *Template DOM Fragment* with listing 5.5.

```
1 <div id="wrapper">
2   <span class="intro"></span>
3   <span class="welcomeText"></span>
4   <div id="data">
5     <input id="name" type="text" />
6   </div>
7   <input type="button" value="Submit" />
8 </div>
```

Listing 5.5: *Template DOM Fragment* Example

Here every line, except lines 6 and 8, contains exactly one element of the *Template DOM Fragment*. All elements from lines 2 to 7 are descendants of the element in line 1, and the one from line 5 is a descendant of both elements from lines 1 and 4. The example from listing 5.4 contains one root *Scope* with the *Selector* `#wrapper`. This means that it matches all elements having an `id` attribute that is equal to `wrapper`. The only element satisfying this criterion is in line 1 and therefore becomes the only item in the set of matched elements for this *Scope*.

The *Scope* with the *Selectors* `div > input` and `span` now retrieves the descendants of all elements matched by its parent. These are exactly the elements from lines 2 to 7. Within these all are selected which either are of type `span` or `input`, having a `div` as their parent. This obviously applies to the elements from lines 2, 3 and 5, but not the `input` from line 7. Although it has a `div` as its parent this element is not part of the context.

Another approach in clarifying the semantics is that it is always possible to eliminate the nesting of *Scopes*. Listing 5.7 shows how this can be done to the example from listing 5.4. Also, it shows clearly why it is advantageous to use nested *Scopes*. The *Selectors* of all ancestor *Scopes*, such as `#wrapper`, have to be written over and over again for each new *Scope*.

5.1.2. Binding

The goal of a *Binding* is to keep values from *Data Targets* synchronized. The typical use case is to bind a value of the *View Mask* to an attribute of the *Presentation Model*, so that if any of the two is modified, the other changes accordingly. To introduce our understanding of a *Binding* we need to define the two components *Adapter* and *Connector* first.

```
global templateDomFragment = ...

foreach (rootScope in Scopes) {
  match(rootScope)
}

function match(scope) {
  var matchedElements = [] /* Set */
  var selectors = scope.getSelectors()

  if (!scope.hasParent()) {
    foreach (selector in selectors)
        matchedElements.addAll(templateDomFragment.apply(selector))
  } else {
    var parentMatchedElements = scope.getParent().getMatchedElements()
    var descendants = getAllDescendants(parentMatchedElements)

    foreach (selector in selectors)
        matchedElements.addAll(descendants.apply(selector))
  }
  scope.setMatchedElements(matchedElements)
  // Recursion
  foreach (childScope in scope.getChildren()) {
    match(childScope)
  }
}
```

Listing 5.6: Pseudo Code Setting Set of Matched Elements for *Scopes*

```
#wrapper {
  <Binding-1>
  <Binding-3>
  <Binding-5>
}
#wrapper div > .input {
  <Binding-2>
}
#wrapper span {
  <Binding-2>
}
#wrapper div div + p {
  <Binding-4>
}
#wrapper div .empty {}
```

Listing 5.7: *Selection* Example Flattened

5.1.2.1. Adapter

Our definition of an *Adapter* is not fundamentally different from the classical design pattern in the famous book *Design Patterns* by the *Gang of Four*. There its intent is described as follows.

> *"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."* — Adapter (Intent), Design Patterns [Gam+95]

Instead of converting class interfaces we want to make access to various *Data Target* implementations consistent. There are many different ways to realize the *Presentation Model*. It can be a plain JSON object, a *Backbone* model (see section 2.1.1) or even a hierarchical structure as in *ComponentJS* (see 2.1.3). Although the implementation and the many examples we present assume that the *View Mask* is always realized with HTML, an *Adapter* could abstract from any other user interface technology. This especially means that our *Concepts* are also applicable to desktop or mobile applications.

The primary goal of the *Adapter* is to make our *Concepts* model agnostic. It converts the interfaces of *Data Targets* into a consistent representation and allows to treat them all alike. Any operation concerning a *Data Target* is never made directly, but always through an *Adapter*. This way the *Presentation Model* implementation can be exchanged effortlessly by only replacing the *Adapter*.

Adapter Syntax

An *Adapter* is formed by concatenating a *Prefix* and an optional *Qualifier*. The *Prefix* is the unique name of the *Adapter* and its *Qualifier* a static parameter. A *View Mask Adapter* that modifies attributes of HTML elements could have the *Prefix* `attr`. Its *Qualifier* would express the name of the attribute modified. Instances of this *Adapter* could be `attr:id`, `attr:class` or `attr:style`.

The *Qualifier* is defined to be static to increase readability. After introducing the *Parameter* concept in section 5.3.5 it will be possible to use multiple and even dynamic instructions.

The syntax of an *Adapter* is defined by grammar 5.2. Its *Prefix* may be a single special character, such as `%`, `@`, `$`, or an arbitrary string. In the first case the *Qualifier* is written directly behind the *Prefix* and in the latter it is separated by a colon. If no *Qualifier* is used, the colon may be dropped. If the *Prefix* is a special character, the *Qualifier* must not be empty. These rules ensure both readability and shortness.

```
Adapter ← [@$#%&] [a-zA-Z_] ([a-zA-Z0-9_-])* |
          [a-zA-Z_] ([a-zA-Z0-9_])* (":" ([a-zA-Z_] [a-zA-Z0-9_-])*)?
```

Grammar 5.2: Syntax of *Adapter*

Examples of syntactically valid *Adapter* are `$name`, `@temp`, `value`, `text`, `attr:data-name`, `model:intro`. Edge cases are `_` or `_:_`.

For the sake of shortness, we recommend to assign a single, special character as *Prefix* to each of the most common *Adapter*, and we will apply this recommendation throughout the rest of our examples. We denote any *Presentation Model Adapter* with the `$` *Prefix* and any *Binding Scope Adapter* with the `@` *Prefix*.

Adapter Semantics

An *Adapter* is associated to a certain *Data Target* and it is the only component that knows how to read from, or write values to it. To be model agnostic, the internal data representation of a *Data Target* cannot be considered inside a *Binding*. Therefore, we introduce the idea of a *Path* which reduces the information about the location of a certain data item to a list of identifiers. The *Adapter* can then decide how to map any *Path* to its underlying data structure.

```
1 var model = {
2   people: [
3     { name: "Alice", age: 35 },
4     { name: "Bob", age: 37 },
5     { name: "Martha", age: 67 }
6   ]
7 }
```

Listing 5.8: *Presentation Model* Example defined with JSON

We want to illustrate the idea and why we need *Paths* with an example. Consider a *Presentation Model* which comprises a list of people who are all equipped with information about their name, age et cetera. It could be implemented with JSON as in listing 5.8. If the *Adapter* is asked for all people, it cannot just return the data in the format of its *Data Target* because this would mean that the *Binding* would also need to have knowledge about its implementation. In fact, the only values that a *Binding* should deal with are instances of primitive data types. This means that an *Adapter* may only produce one or more *Paths* that all are *References* to primitive values.

- `["people", 0, "name"]`
- `["people", 0, "age"]`
- `["people", 1, "name"]`
- `["people", 1, "age"]`
- `["people", 2, "name"]`
- `["people", 2, "age"]`
- ...

If the name of the second person should be bound to a text box, the *Presentation Model Adapter* can be simply asked for the correct string which is then placed in the text box. If, however, the *Binding* should work aswell in the other direction so that

a user can change the value of the text box, there needs to be way of instructing an *Adapter* to update a certain part of its *Data Target*. The *Path* exactly serves this purpose. To summarize this, an *Adapter* needs to provide the functionality of getting the primitive value or a list of subordinate *Paths* for a given *Path* and of setting the information at a given *Path* to a certain primitive value.

According to the three *Data Targets*, we differentiate three types of *Adapter* namely the *View Mask*, *Binding Scope* and *Presentation Model Adapter*. We assume that each *Adapter* is able to provide information about its type. The type of an *Adapter* determines what an empty *Path* stands for. In case of the *Presentation Model* and *Binding Scope Adapter* it just means the whole *Data Target*. The *View Mask Adapter*, on the other hand, is never associated to the whole *Template DOM Fragment*, but always to the element matched by its enclosing *Scope*. Another difference is that, since the *Template DOM Fragment* is a static input to the *View Data Binding*, we assume that any *View Mask Adapter* never modifies the structure of its *Data Target*, but only the attributes of its elements.

The *Qualifier* of an *Adapter* is nothing but a possibility to syntactically specify the first element of the *Path*. The semantics of the symbol referencing an *Adapter* is the functionality of getting and setting values for given *Paths*. If a *Qualifier* is present, it is pushed in front of any incoming *Paths* when setting values and removed from any outgoing *Paths* when getting values.

We further assume that an *Adapter* is observable. This means that if a change listener was previously registered for the *Path* specified by the *Adapter*, this listener must always be notified whenever the value referenced by the *Path* changes.

5.1.2.2. Connector

Apart from *Adapter*, a *Binding* may consist of *Connectors* or, more generally, of a *Connector Chain*. In contrast to the *Adapter*, a *Connector* does not deal with *Data Targets*, but processes, transforms or modifies JSON data that is propagated through a *Binding*. It has no knowledge about its surroundings and especially has no direct access to any *Data Target*. In other words, *Connectors* are free of side effects.

A *Connector* should intuitively be seen as a function that consumes input parameters and produces an output. To illustrate how useful they are we present the various types of *Connectors* that we identified in table 5.1.

All *Connectors* from table 5.1 have in common that they produce an output that is unrelated to their input. There are, however, other types of *Connectors* shown in table 5.2 that only rearrange instead of modify their input.

Connector Syntax

Grammar 5.3 for *Connectors* is very simple, since they are represented by just strings.

```
Connector ← [a-zA-Z_] ([a-zA-Z0-9_])*
```

Grammar 5.3: Syntax of *Connector*

Name	Description	Examples
Converter	Most abstract form of a *Connector*. It may perform any kind of conversion.	Formatters for dates or currencies, replacing i18n identifiers with their translations or any typical use of the higher order function *map*
Validator	Typically information that is entered by a user is validated. This type of *Connector* produces an indicator, whether the validation was successful, and may additionally include the original value.	String length and character set or email and postcode validation
Aborter	Causes that the *Propagation* of data through a *Binding* stops.	Two *Bindings*, but only one of them active at the same time based on a condition
Aggregator	Operates on collections and produces one value or a collection that has fewer elements than its input.	Count, sum, average, grouping

Table 5.1.: Value modifying *Connectors*

Name	Description	Examples
Filter	Looks at every element of a collection and decides if it remains in or leaves the collection.	List of people, filtered by gender, maximum age, place of birth etc.
Sorter	Modifies the order of elements in a collection.	List of people, sorted by name, age etc.
Generator	In theory multiplies certain or all elements of a collection and produces an output that is larger than its input.	

Table 5.2.: Collection modifying *Connectors*

Connector Semantics The semantics of the symbol referencing a *Connector* are defined by a function consuming and producing a JSON object. In addition, the output of a *Connector* can be a special abort symbol.

Binding Syntax

The syntax of a *Binding* is defined by grammar 5.4. In defining the non-terminal *Binding* it adds to grammar 5.1 for *Scopes*. For the sake of readability *Segment Binding Operator* is abbreviated to *Sbo* and *Connector Binding Operator* to *Cbo*. Listing 5.9 shows different examples of syntactically valid *Bindings*.

```
Binding ← Adapter Sbo ConnectorChain Sbo Adapter |
          Adapter Sbo Adapter
ConnectorChain ← Connector (Cbo Connector)*
Sbo ← "<-" | "->"
Cbo ← "<-" | "->"
```

Grammar 5.4: Syntax of *Binding*

```
1 Adapter <- Adapter
2 Adapter -> Adapter
3 Adapter <- Connector <- Adapter
4 Adapter -> Connector -> Connector -> Adapter
5 Adapter <- Connector -> Connector <- Adapter
```

Listing 5.9: *Binding* Examples

Line 5 defines a *Binding* which is semantically wrong. To keep the grammar as simple as possible, we decided against preventing such ill-formed *Bindings* on the grammar level.

The symbols `<-` and `->` are used for both *Segment* and *Connector Binding Operators* because the terminological difference is neither intuitive nor important from a developer perspective. The symbols itself are intuitive, short and can not be easily confused with the relational operators < or >.

By combining grammar 5.4 and 5.2 for *Adapter*, it is now possible to express simple *Bindings* as shown in listing 5.10. We generally prefer to write *View Mask Adapter* on the left and *Presentation Model Adapter* on the right.

```
1 div {
2   attr:class <- $class
3 }
```

Listing 5.10: *Adapter* Example

Depending on the semantics of the participating *Adapter*, this *Binding* could express that the attribute from the *Presentation Model* named `class` is bound to all `class` attributes of any `div` elements inside the `DOM Fragment`.

Listing 5.11 shows a more complex example that also makes use of the *Connectors* defined by grammar 5.3.

```
1  div.wrapper {
2    span.intro {
3      text <- i18n <- $introI18nId
4    }
5
6    input#zip {
7      value -> validateZip -> $zipValid
8    }
9
10   @text <- doOnce <- $defaultText
11   textarea#text {
12     value -> @text
13   }
14
15   span.charsWithoutWhitespaces {
16     text <- length <- join <- split <- @text
17   }
18 }
```

Listing 5.11: *Connector* Examples

The names of the *Connectors* hint at their semantics. They could be a *Converter* in line 3, a *Validator* in line 7 and an *Aborter* in line 10. Line 16 shows a *Connector Chain*, which could calculate the number of non blanks in `@text` depending on the meaning of `split`, `join` and `length`.

Binding Semantics

Figure 5.1 shows the typical flow of data realized by *Bindings*. It visualizes how *Adapter* abstract away their underlying *Data Targets* and that typically *Presentation Model* are bound to *View Mask Adapter* with optional intermediate *Connector Chains*. Data may also be stored temporarily inside the *Binding Scope*.

A *Binding* is semantically valid if it is recognized by grammar 5.4 and uses only one of the available *Binding Operators*. It is therefore either read from left to right if it uses `->` only or read from right to left if it uses `<-` only. With this definition of *direction* it intuitively makes sense that *Adapter* and *Connectors* inside a *Binding* have a *Successor* and a *Predecessor*. We call the *Adapter* without a preceding element the *Source* and the one lacking a *Successor* the *Sink Adapter* of a *Binding*.

Semantically it is irrelevant in which direction the *Binding* is written. This means that it may be mirrored without changing its meaning. Also, the order of *Bindings* inside a *Scope* is irrelevant to their semantics.

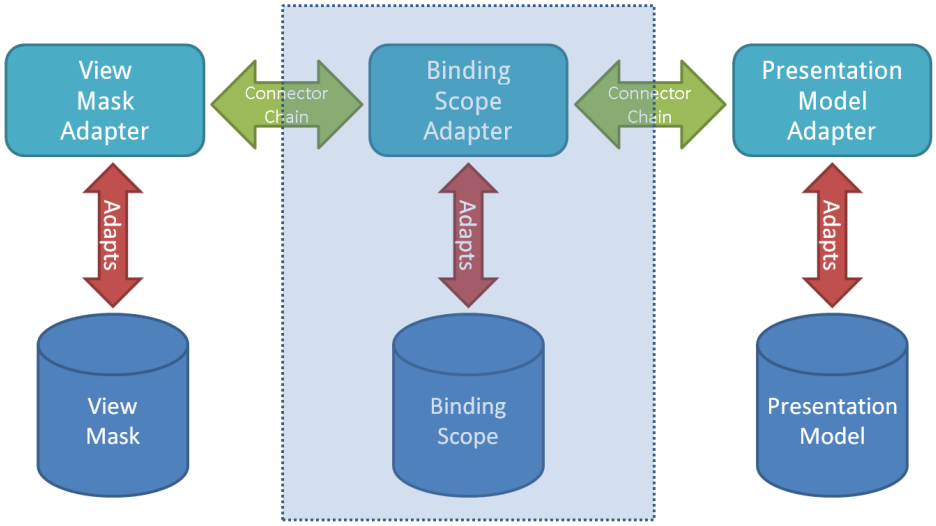


Figure 5.1.: Typical Data Flow through Adapter and Connectors

Every *Connector Chain* inside a *Binding* can be expressed by a single *Connector* which consecutively executes their underlying functions. Consider the following *Connector Chain*.

```
@sink <- foo <- bar <- baz <- @source
```

It can be expressed by a single *Connector* `quux`, which first processes the input in the same way as `baz` and then uses that output as the input for the part representing `bar`. The result becomes the input for the logic replacing `foo` producing the overall output.

This means that it is sufficient to define the semantics of the following two *Bindings*, one without and one with a *Connector*.

```
Adapter <- Adapter
```

```
Adapter <- Connector <- Adapter
```

Any *Binding* operates as soon as the value of its *Source Adapter* changes. We call the algorithm that handles that change *Propagation*. When it starts, the *Source Adapter* is asked to produce a set of *Paths* from the *Path* that is defined through its *Qualifier*. This set of *Paths* is converted into structured JSON. Each *Path* is also wrapped into a *Reference* which is nothing but a *Path* in combination with the *Adapter* responsible for it. This means that it can always be converted to its underlying primitive value. To illustrate this, consider the following examples denoted with JSON shown in table 5.3.

The successor of the *Source* is either a *Connector* or an *Adapter*. If it is an *Adapter* other than the *Binding Scope Adapter* all *References* are converted into values and the resulting object is passed to the *Adapter*, which writes it to its associated *Data Target* at the *Path* that is defined by its *Qualifier*. If it is a *Connector*, it is called with the converted object as its input and *Propagation* continues with the result of the

```
// Initialization
foreach (binding in bindingSpecification) {
  binding.getSource().observe(
    propagate(binding.getSource(), binding.getSink())
)}
// Propagation
function propagate(source, sink) {
  // Read
  var pathToRead = []
  if (source.hasQualifier()) {
    pathToRead.add(source.getQualifier())
  }
  var paths = source.read(pathToRead)
  // Convert
  var value = convert(paths)
  // Evaluate Connector
  if (source.getNext().getType() == "Connector") {
    var connector = source.getNext()
    value = connector.process(value)
    if (value == abort) {
      return
  }}
  // Write
  if (sink.getType() == "BindingScopeAdapter") {
    var currentValue = bindingScope.get(sink)
    if (!currentValue ||
          (!isReference(currentValue) && !isReference(value))) {
      bindingScope.write(sink, value)
    } else if (!isReference(currentValue) && isReference(value)) {
      bindingScope.write(sink, value)
    } else if (isReference(currentValue) && isPrimitive(value)) {
      currentValue.set(value)
      bindingScope.notifyChanged(sink)
    } else if (isReference(currentValue) && isReference(value)) {
      if (currentValue.getType() == value.getType()) {
        bindingScope.write(sink, value)
      } else {
        currentValue.set(value.getValue())
        bindingScope.notifyChanged(sink)
      }
    } else {
      bindingScope.write(sink, value)
  }}} else {
    value = resolveReferences(value)
    sink.write(value)
}}
```

Listing 5.12: Propagation

Source Adapter	Paths Returned	Converted Object
$name	`[[]]`	`{Adapter: $, Path: [name]}`
$people	`[[0], [1], [2]]`	`[{Adapter: $, Path: ["people", 0]}, {Adapter: $, Path: ["people", 1]}, {Adapter: $, Path: ["people", 2]}]`
$data	`[["foo"], ["foo", "bar"], ["foo", "bar", 0]`	`{foo: [{Adapter: $, Path: ["foo", "bar", 0]}]}`

Table 5.3.: Conversion to *References* when reading from *Adapter*

Connector. If it, however, produced the *abort* symbol, *Propagation* stops. The process is shown with pseudo JavaScript code in listing 5.12.

If the *Data Sink Adapter* of the *Binding* is the *Binding Scope Adapter*, it is handled differently, since it is the only *Adapter* that stores *References*. With *element* in the following explanation we mean the data entry in the *Binding Scope* referenced by the *Qualifier* used together with the *Binding Scope Adapter*

1. If the element did never receive a value or if neither its current value nor the value that should be written into the element are *References*, the new value is written as it is (line 28). This case occurs whenever *Connectors* that replace *References* by their values are used in a *Binding*.

2. This is also done if the new value is a *Reference*, while the old value of the element was not (line 30). A situation when this case applies could be a *Connector* that only sometimes produces *References*.

```
1 input {
2   @temp <- $value
3   value -> @temp
4 }
```

Listing 5.13: *Binding Scope* Aliasing

3. If, however, the current value of the element is a *Reference* and the new value is a primitive value, instead of overwriting the element with the new value, the *Reference* is asked to write the new value to its underlying *Adapter*. (line 32). This case is necessary to realize the typical aliasing functionality of the *Binding Scope*. The *Iteration* concept (see section 5.1.3) makes heavy use of this to backtrack the *Entry* and *Key Adapter* values to their original *Source Adapter*. Another example is shown in listing 5.13. Here `$value` is aliased with `@temp`. When `@temp` is written by `value` it already stores a *Reference* to `$value`.

4. If both the current and new value of the element are *References*, we need to further decide whether those two *References* point to *Adapter* of the same type (line 35).

```
1 @temp <- abortIfCondition <- $foo
2 @temp <- abortIfNotCondition <- $bar
3 ... -> @temp
```

Listing 5.14: *References* in *Binding Scope* Overwriting Each Other

a) If that is the case, the old *Reference* is overwritten by the new *Reference* (line 36). A use case where this condition applies is shown in 5.14. Here, based on a condition, *References* to different attributes of the *Presentation Model* are written into the same *Binding Scope Adapter*.

```
1 @foo <- $value
2 @bar <- value
3
4 @bar <- @foo
```

Listing 5.15: Aliasing two *Adapter*

b) If the two types, however, are different, the element is not overwritten, but the underlying value of the new *Reference* is written into the underlying *Adapter* of the current *Reference* (line 38). An example why this behavior is necessary is shown in listing 5.15. Here different *Adapter* are aliased.

5. The last case is where we only know that `currentValue` is a *Reference* and that the new value neither is primitive nor a *Reference* (line 42). There is at least one situation where this can happen. If the element acts as the source of an *Iteration* and that *Iteration* initially iterates an empty collection, the *Adapter* stores a single *Reference* pointing to that collection. If now elements are added to the collection, the new value is no longer a *Reference*, but a list of *References* of the same type. If more complex objects are iterated, there might be an arbitrary structure of *References*.

5.1.2.3. Binding Scope

The *Binding Scope* is an artificial third *Data Target* that should intuitively be seen between *Presentation Model* and *View Mask*. It is a hierarchical data structure that exactly follows the system of *Scopes*. When the *Binding Scope Adapter* that is always written together with a *Qualifier* is used within the *Binding* of a *Scope* that *Qualifier* becomes visible to all descendant *Scopes*.

```
1  div#first {
2    @foo <- ...
3    div#second {
4      ... -> @foo
5      @bar <- ...
6    }
7    div#third {
8      ... -> @foo
9      @bar <- ...
10 }}
```

Listing 5.16: *Binding Scope* Example

In listing 5.16 there are three *Scopes* from lines 1 to 10, 3 to 6 and 7 to 10. We call these *Scopes* the first, second and third *Scope*. The *Binding Scope Adapter*, which we denote with the *Prefix* `@`, is used multiple times, particularly with the two *Qualifiers* `foo` and `bar`. Since `foo` was used in line 2 inside the first *Scope* that has the second and third *Scope* as its children, the references to `foo` in line 4 and 8 all point to the same attribute of the *Binding Scope*. The references in line 5 and 9, on the other hand, reference different attributes of the *Binding Scope*, although they use the same *Qualifier* `bar`. This is because they both introduce a new *Qualifier* that was never used in any of their ancestor *Scopes*.

There is a simple recursive top-down algorithm that renames *Qualifiers* of *Binding Scope Adapter* so that same *Qualifiers* always reference the same attribute of the *Binding Scope*. This algorithm demonstrates how the visibility of *Binding Scope Adapter Qualifiers* is resolved. It is shown in listing 5.17 with pseudo JavaScript code.

```
1  global count = 1
2  disambiguate(scope, {})
3
4  function disambiguate(scope, assignMap) {
5    for (var qualifier in scope.getBindingScopeQualifiers()) {
6      if(qualifier not in assignMap) {
7        assignMap[qualifier] = "t" + count++
8      }
9      rename(qualifier).to(assignMap[qualifier])
10   }
11
12   for (child in scope.getChildren()) {
13     disambiguate(child, assignMap.clone())
14 }}
```

Listing 5.17: *Binding Scope* Disambiguation Algorithm

To show how the algorithm operates on the example from listing 5.16 we first visualize it as a tree of *Scopes* in figure 5.2. Every *Scope* in the tree contains a set of *Binding Scope Adapter Qualifiers* used inside that *Scope*.

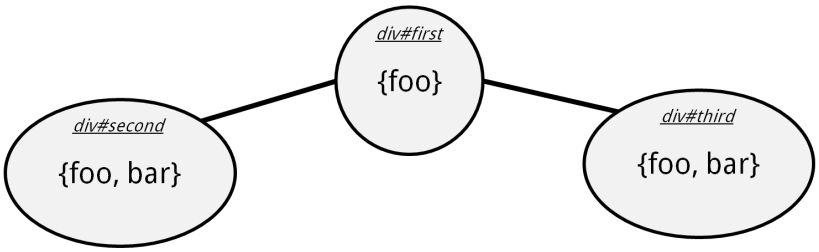


Figure 5.2.: Scope Tree from Listing 5.16

`Disambiguate` is initially called with `div#first` and an empty `assignMap`. In the first for loop from lines 5 to 10 the entry `{"foo", "t1"}` is added to `assignMap`, `count` increased by one and `foo` inside `div#first` renamed to `t1`. The second for loop from lines 12 to 14 makes two recursive calls for `div#second` and `div#third`, each time with a copy of `assignMap`. In both calls `foo` is again renamed to `t1`, since it is already in `assignMap` and `bar` is renamed to `t2` and `t3` respectively. Figure 5.3 shows what the tree looks like after the algorithm is finished.

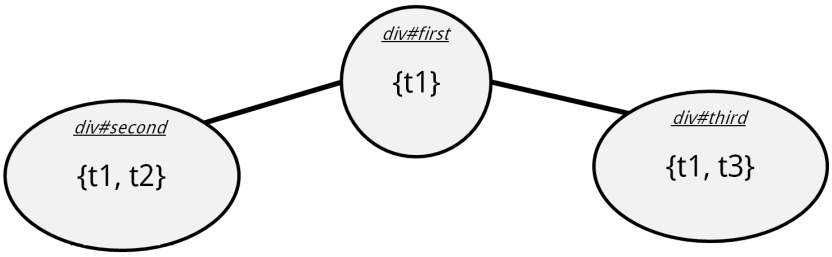


Figure 5.3.: Scope Tree from Listing 5.16 After Disambiguating Qualifiers

We introduced the *Binding Scope* for three reasons.

1. It allows to structure the *Binding Specification* more flexibly. By factoring out recurring *Connector Chains* into a parent *Scope,* less code has to be written. Also it allows breaking long *Bindings* into multiple shorter ones.
2. Without it would be only possible to realize binding different view elements with artificial attributes in the *Presentation Model*. Since the associated element of a *View Mask Adapter* is defined by its *Scope,* it is impossible to cross-reference another element of the *View Mask* there. If for instance a button should be active only if a text box is filled, the information whether the text box is empty cannot be retrieved inside the *Scope* that selects the button. Listing A.2 in the appendix shows how this can be solved by utilizing the *Binding Scope*.
3. It nicely fits together with the concept of *Iteration* (see section 5.1.3). There *Entry* and *Key* need to be provided somehow and they can both easily be stored inside the *Binding Scope* without introducing a new concept.

5.1.3. Iteration

A common requirement for user interfaces is to repeat certain parts for each element in a collection. We call the concept that realizes this *Iteration*. Hiding or displaying parts of the template can be seen as *Iteration*, too. It just means that one part is repeated one or zero times.

An *Iteration* has a part of the *Template DOM Fragment* and its associated *Binding Specification* as its input. The *Iteration* implicitly defines a *Binding* between the collection that is iterated and their corresponding *DOM Fragments* and *Bindings* that result from iterating the *Template DOM Fragment* and *Binding Specification* respectively. The concept is particularly challenging because it combines many different functionalities. It is the only concept that manipulates both the *Template DOM Fragment* and the *Binding Specification*.

An *Iteration* needs to provide a way for accessing the currently iterated element and its key. In the case of conditionally displaying or hiding parts of the *Template DOM Fragment*, this information is useless, since there is no iterated element that could be referenced. We therefore define the element in such cases to always be `true` and the key to always be `0`.

Iteration Syntax

Since no framework we examined in chapter 3 allows to define *View Data Binding* separately, they all include *Iteration* with special tags in the *Template DOM Fragment*. This, however, is not an option for us, since *Iteration* clearly is not a concern of the *View Mask*, but of the *Binding Specification*. We considered several different options of denoting *Iteration*.

1. As a special *Iteration Adapter*
2. As an *Adapter* that manipulates the *DOM Fragment*
3. As a special *Connector*
4. As an own construct similar to *Scopes*
5. As a Pseudo-Selector attached to *Selectors*
6. As additional information to the *Selector* of a *Scope*

Consider the example of a list of names that is stored in the `names` attribute of the *Presentation Model*. The goal is to bind an unordered HTML list to the list of names. The *Template DOM Fragment* for this task is shown in listing 5.18. Listing 5.19 shows what the various options could look like in a *Binding Specification*.

```
1 <div id="template">
2   <!-- Iteration BEGIN -->
3   <ul id="names">
4     <li class="name"></li>
5   </ul>
6   <!-- Iteration END -->
7 </div>
```

Listing 5.18: Example of a *Template DOM Fragment* for *Iteration*

```
1  // 1. As a special Iteration Adapter
2  li {
3    each(@name) <- $names
4    text <- @name
5  }
6
7  // 2. As an Adapter that manipulates the DOM Fragment
8  ul {
9    html <- each("li") <- $names
10   li {
11     text <- ???
12   }
13 }
14
15 // 3. As a special Connector
16 li {
17   text <- each <- $names
18 }
19
20 // 4. As an own construct similar to Scopes
21 iterate (@name <- $names) {
22   li {
23     text <- @name
24   }
25 }
26
27 // 5. As a Pseudo-Selector attached to Selectors
28 li:each(@name <- $names) {
29   text <- @name
30 }
31
32 // 6. As additional information to the Selector of a Scope
33 li (@name:  $names) {
34   text <- @name
35 }
```

Listing 5.19: Different Options of *Iteration* in *Binding Specification*

One important observation that can be made here is that *Iteration* creates a region very similar to *Scopes*. Therefore the first three options fail to express *Iteration* appropriately because they just try to include it inside *Bindings*. These solutions all have in common that they do not prevent specifying multiple *Iterations* within the same region. For instance there could be another line like

```
html <- each("li") <- $otherNames
```

after line 9. Semantically this does not make sense because it would conflict with the *Iteration* already there. The syntax should intuitively prevent such cases by letting *Iteration* create a syntactical region.

In addition, all options 1 to 3 fail to satisfy the definitions and restrictions for *Adapter* and *Connectors*. In section 5.1.2.1 we defined that an *Adapter* may only modify attributes of its associated element, but not its structure. This would be violated by options 1 and 2. *Connectors*, on the other hand, do by our definition from section 5.1.2.2 not have an associated element and are side effect free. This, in turn, is disregarded by options 2 and 3.

On first sight, options 4 to 6 seem to be very similar. The problem with option 4 is that the iterated element is not attached to the *Iteration*. This means that there could be *Selectors* inside the *Iteration* that refer to unrelated elements of the *Template DOM Fragment*, such as tags nested inside one another or incoherent elements. Option 5 tries to mimic CSS pseudo-classes, but fails the analogy, since in CSS, it is always allowed to add more selectors after a selector.

We therefore decided to employ option 6. It creates an own syntactical region that is directly associated with the iterated element, but is not part of the *Selector*. It is intuitively understood and requires little amounts of code. To include it, we rework grammar 5.1 for *Scopes* into grammar 5.5

```
Scope ← Selectors Iteration? "{" ScopeBody* "}" |
        Selectors Iteration
Selectors ← Selector ("," Selector)*
ScopeBody ← Scope | Binding
```

Grammar 5.5: Syntax for *Scope* with *Iteration*

The differences are that it is now possible to add an *Iteration* after the *Selectors* of a *Scope* and that the curly brackets may be dropped if no *ScopeBody* is present. Finally, we still need to define the non-terminal symbol for *Iteration* which is done by grammar 5.6. It uses the symbols *Entry*, *Key* and *Collection*, which we only added to make it easier to comprehend.

This grammar allows three different ways of writing an *Iteration*, while the first is designed for conditional display (*When*). Each of the three can also be used for repeating elements (*Each*) with designated optional spots for both *Entry* (first) and *Key* (second).

```
Iteration ← "(" (Entry ("," Key)? ":")? Collection ")"
Entry ← Adapter
Key ← Adapter
Collection ← Adapter
```

Grammar 5.6: *Iteration*

1. `(Adapter)`
2. `(Adapter:  Adapter)`
3. `(Adapter, Adapter:  Adapter)`

Listing 5.20 shows a more complex example of how *Iterations* may be used inside the *Binding Specification*. It again uses the *Template DOM Fragment* from listing 5.18. Line 3 would semantically most probably be an example of using only one *Adapter* to realize *When*, while line 5 uses both *Entry* and *Key Adapter*.

```
1 @showNames <- notEmpty <- $names
2 #names (@showNames)
3
4 li (@name, @index:  $names) {
5   text <- @name
6   attr:id <- @index
7 }
```

Listing 5.20: *Iteration* Example

Iteration Semantics

It is assumed that the *Paths* yielded by the last *Adapter* inside an *Iteration* can always be converted to a collection (realizes *Each*) or a Boolean value (expresses *When*). It can be seen as the input of the *Iteration*, and in this context we call it *Source Adapter*. All other *Adapter* need to be *Binding Scope Adapter*. Since the grammar can not know which *Prefix* the *Binding Scope Adapter* uses, every implementation needs to validate this at runtime. We call these two *Entry* and *Key Adapter* respectively.

An *Iteration* expands both the *Template DOM Fragment* and the *Binding Specification*, depending on the values that are retrieved from the *Source Adapter*. For each repetition of the *Binding Specification* the correlating collection items and their keys are injected into the *Entry* and *Key Adapter*. Since the *Binding Scope Adapter* is hierarchical (see section 5.1.2), they can reference different values in each repetition.

In case of *When*, the *Template DOM Fragment* and the *Binding Specification* are modified as follows.

- If the *Source Adapter* yields `true` ...
  - ... the *Template DOM Fragment* is unchanged
  - ... the Iteration is removed from the *Scope*. The *Scope* itself remains unchanged regarding its *Selector*, *Bindings* and child *Scopes*. If the *Entry* or *Key Adapter* were used, their values are initialized with `true` and `0` respectively inside that *Scope*.
- If the *Source Adapter* yields `false` ...
  - ... all elements that are matched by the *Selector* of the iterated *Scope* are removed from the *Template DOM Fragment*.
  - ... the *Scope* is completely removed from the *Binding Specification* including its *Bindings* and children.

In case of *Each*, the *Template DOM Fragment* and the *Binding Specification* are modified as follows.

- The selected elements are removed from the *Template DOM Fragment* and the *Scope* is removed from the *Binding Specification* as if the *Iteration* was a *When* yielding `false`. The parts that were removed and the locations where they were removed need to be remembered for the next steps.
- For each element in the collection retrieved from the *Source Adapter* ...
  - ... insert that part of the *Template DOM Fragment* which was selected after all previously inserted parts or if it is the first part, at the old position.
  - ... create a *Selector* which matches only the part added in the previous step and insert a new *Scope* after all previously inserted *Scopes* or if it is the first one, at the old position. The new *Scope* is equal to the old one except that it has a different *Selector* and no *Iteration*. It is assumed that it is always possible to find a new *Selector*. In HTML for instance this is unproblematic, since CSS includes pseudo-classes that allow to select the n-th child of a tag.

Obviously, the challenge of *Iteration* does not lie in initially modifying both the *Template DOM Fragment* and the *Binding Specification* to eliminate all *Iterations*, but in keeping track of the changes of the *Source Adapter*. If the underlying collection or the truth value changes, the *Template DOM Fragment* and the *Binding Specification* have to be modified to reflect those changes. We will show an efficient way to do this in chapter 6. Although we did not show an example for it, *Iterations* may be nested, which further exacerbates the problem.

To make the semantics of *Iteration* more clear we expand our initial example with its *Template DOM Fragment* from listing 5.18 by assuming that `$names` yields the array `["Alice", "Bob", "John"]`. Listing 5.21 shows the *Template DOM Fragment* after it was expanded by the *Iteration* and listing 5.22 the *Binding Specification* that emerged from lines 33 to 35 of listing 5.19.

```
1 <div id="template">
2   <!-- Iteration BEGIN -->
3   <ul id="names">
4     <li class="name"></li>
5     <li class="name"></li>
6     <li class="name"></li>
7   </ul>
8   <!-- Iteration END -->
9 </div>
```

Listing 5.21: Expanded *Template DOM Fragment*

```
1  li:nth-child(1) {
2    @name <- select0 <- $names
3    text <- @name
4  }
5  li:nth-child(2) {
6    @name <- select1 <- $names
7    text <- @name
8  }
9  li:nth-child(3) {
10   @name <- select2 <- $names
11   text <- @name
12 }
```

Listing 5.22: Expanded *Binding Specification*

In lines 2, 6 and 10 of listing 5.22 *Bindings* are added to illustrate how the collection items are injected into the *Entry Adapter*. This, however, does not need to happen explicitly, since it is specified as part of the *Iteration* concept. It should be noted that `select0`, `select1` and `select2` store a *Reference* inside `@name`. This means that it is possible to use the *Entry Adapter* as the *Data Sink* of a *Binding* (see section 5.1.2).

Although the syntactical representation of *Iteration* allows to easily express complex logic, semantically there are a few caveats.

1. The part of the *Template DOM Fragment* that is repeated by the *Iteration* is defined by the *Selector* of the *Scope* that comprises the *Iteration*. This is important to note, since almost all other frameworks iterate all descendants of the selected elements, while we repeat the descendants *and* the selected element. The first option is easier to implement, since this way there is always a reference element even if there are no elements in the *Iteration*. We however want to provide a more powerful solution that also allows to define *Iterations* that can be placed next to each other without the necessity of wrapper elements.

2. An element from the *Template DOM Fragment* may be matched by the *Selector* of a *Scope* that is not and by another that is iterated. All *Bindings* inside the non-iterated *Scope* are then applied as if they would occur inside the iterated *Scope*, but they do not have access to the *Entry* and *Key Adapter*. Further, this means that conflicting references need to be resolved before merging.

3. If the *Selector* of an iterated *Scope* matches more than one element inside the *Template DOM Fragment*, each of those elements is iterated individually.

4. It is not allowed that the same element inside the *Template DOM Fragment* is matched by the *Selectors* of two or more *Scopes* which all have an *Iteration*. We defined that the order of *Bindings* and *Scopes* is semantically irrelevant, but in such a case a decision about the order of *Iterations* inside the *Template DOM Fragment* would have to be made. The problem, however, can be solved by multiplying the relevant part of the *Template DOM Fragment*, selecting it explicitly through the *Selectors* of the iterated *Scopes* and, thus, making the order of *Iterations* explicit.

5. As we have explained in section 5.1.2, using a *Binding Scope Adapter* makes its *Qualifier* visible to all descendant *Scopes*. Since *Iteration* creates new *Scopes* during the process of expansion, all references to *Entry* and *Key Adapter* are individual within a *Scope* created by expansion. If now the same *Qualifier* as in the *Entry* or *Key Adapter* was used in an ancestor of the *Scope* that is iterated, it would make all references to *Entry* and *Key* point to the same elements. This would inevitable lead to conflicting *Bindings* and we therefore assume that the *Qualifiers* used for the *Entry* and *Key Adapter* never occur in any ancestor of the *Scope* that is iterated.

6. The *Source Adapter* of an *Iteration* may be a *Presentation Model* or *View Adapter*. In the latter case the *Selector* of the *Scope* that encloses the iterated *Scope* defines its associated element. If there is no such *Scope* inside the same *Group* of the *Binding Specification*, the use of a *View Adapter* is not allowed.

5.2. Core Structure Concepts

In this section we introduce the Core Structure Concepts *Identification* and *Insertion*. In short, *Identification* allows to label certain parts of a *Binding Specification* for reference and *Insertion* to plug-in external content not covered by other concepts.

5.2.1. Identification

With *Identification* it is possible to name certain parts of a *Binding Specification*. It is a *Structure Concept* because software is typically structured into components, where one entails the *Template DOM Fragment*, another the *Presentation Model* and yet another the *Binding Specification*. This means that we need to provide a mechanism that allows mapping the *Binding Specification* to its corresponding *Template DOM Fragment* and *Presentation Model*. One could argue that this could be done with the name of the file that comprises the *Binding Specification*. Especially in web environments, however, the number of files is typically reduced to a minimum by compressing all content of a certain type, for example all *Binding Specifications* into one big file.

Identification Syntax

In section 5.1.1 we have shown how *Scopes* build a tree. *Scope* and *Group*, which represents *Identification*, are in fact special *Blocks*. This means that wherever a *Scope* is syntactically allowed, there is as well a *Group*, and vice versa. We first rework the latest version of grammar 5.5 for *Scopes* so that it recursively references *Blocks* instead of *Scopes*. In addition, it makes sense to define *Binding Specification* and *Block* at this point with grammar 5.7. Listing 5.23 shows a *Binding Specification* that makes use of *Identification* several times. It also shows how *Groups* may be nested.

```
BindingSpecification ← Blocks
Blocks ← (Block)*
Block ← Group | Scope
Group ← Identification "{" Blocks"}"
Identification ← "@binding " GroupIdentifier
GroupIdentifier ← [a-zA-Z_]([a-zA-Z0-9_-])*
Scope ← Selectors Iteration? "{" ScopeBody* "}" |
        Selectors Iteration
Selectors ← Selector ("," Selector)*
ScopeBody ← Block | Binding
```

Grammar 5.7: Syntax of *Binding Specification*

```
1  @binding foo {
2    div {
3      ...
4    }
5    @binding bar {
6      span {
7        ...
8      }
9    }
10 }
11 @binding baz {
12   ...
13 }
```

Listing 5.23: *Identification* Example

Identification Semantics

Identification does not alter the meaning of a *Binding Specification*. It is assumed that the *Group Identifier* is unique in one level of the subtree built of *Groups*. If *Groups* are nested, they are referenced with dot-notation. This means that there might be one *Group* referenced with `foo` and another with `bar.foo`.

References to *Groups* always reference the *Binding Specification* at a state before its *Iterations* were expanded as described in section 5.1.3.

5.2.2. Insertion

The *Template DOM Fragment* is a static input to the *View Data Binding*. This means that after it is connected to the *Presentation Model* through the *Binding Specification* it cannot be changed by something that is not a *Concept*. In complex applications, however, the user interface which is represented by the *Template DOM Fragment* often does not reside in a single component. Instead, there may be a hierarchy of components and children need a designated spot to render their part of the user interface into the *Template DOM Fragment* of their parents. Figure 2.3, which we used earlier to introduce ComponentJS in section 2.1.3, shows such a user interface where the same login dialog is plugged four times into a container component.

We call the designated spot a *Socket* and assume that all content plugged into it, is managed externally. In particular, it means that it cannot be manipulated by the *Binding Specification*.

Insertion Syntax

We allow adding an additional *Label* to a *Scope*, which turns its matched element into a *Socket*. Since the *Binding Specification* may not reference content plugged into the *Socket*, this *Scope* must not have children and be always empty. The new syntax elements are defined with grammar 5.8, which reworks parts of grammar 5.7 for *Binding Specifications*. Listing 5.24 shows in lines 2 and 4 how the new syntax element can be used inside a *Binding Specification*.

```
Scope ← Selectors Iteration? "{" ScopeBody* "}" |
        Selectors (Iteration | Label)
Label ← "::" LabelIdentifier
LabelIdentifier ← [a-zA-Z_]([a-zA-Z0-9_-])*
Selectors ← Selector ("," Selector)*
ScopeBody ← Block | Binding
```

Grammar 5.8: Syntax of *Scope* with *Insertion*

```
1 @binding foo {
2   #bar::baz
3   div (@person: $people) {
4     .qux::quux
5   }
6 }
```

Listing 5.24: *Insertion* Example

Insertion Semantics

Adding a *Label* to a *Scope* creates a *Socket*. We assume that the *Selector* of that *Scope* exactly matches one element from the *Template DOM Fragment* whose children are removed.

Semantically *Sockets* do not affect a *Binding Specification*, but we assume that there is an interface which allows to request a *Socket*. *Sockets* are identified with dot notation including all *Group Identifiers* of the *Binding Specification*. In the example from listing 5.24 the *Socket* from line 2 would be referenced by `foo.baz` and the one from line 4 by `foo.quux`.

A *Socket* comprises a reference to its associated part of the *Template DOM Fragment*. Due to the concept of *Iteration* this part might be missing or be present multiple times. In the latter case each instance of the *Socket* also contains information about the *Entry* and *Key* which lead to its creation. Since *Iterations* may be nested, this information needs to be provided for every *Iteration* that occurred in any ancestor *Scope*.

Apart from requesting the current instances of a *Socket*, we further assume that another interface allows to register for changes of *Socket* instances. This means that it must be possible to register an observer that is notified whenever an instance of a *Socket* is created or destroyed through *Iteration*.

Last, the *Socket* provides a way to insert content into its associated part of the *Template DOM Fragment*. This content is guaranteed to be never modified by the *View Data Binding* defining the *Socket*.

5.3. Convenience Binding Concepts

After introducing all of our *Core Concepts* we now turn to *Convenience Concepts* for *Binding*. They all have in common that they make it easier to express complex facts and render our *Concepts* more desirable to work with.

To decide which *Convenience Concepts* to include we examined typical use cases for *View Data Binding* and how they would be realized using *Core Concepts* only. We then identified patterns that were recurring and cumbersome to implement. Whenever possible, we converted those patterns into independent *Convenience Concepts*, primarily to reduce the amount of *View Data Binding* code.

5.3.1. Two-Way Binding

It is often required to bind values in two directions. For example, the value of a text box in the user interface is bound to an attribute of the *Presentation Model* so that a change is reflected in the other end, no matter where it was made. In such instances it would be necessary to write two *Bindings* with opposing *Segment Binding Operators*. To halve the amount of code in such cases we introduce the *Concept* of *Two-Way Binding* that adds a third possibility to write *Binding Operators*.

Two-Way Binding Syntax

We rework grammar 5.4 for *Bindings* into grammar 5.9. It adds a new terminal symbol expressing a *Two-Way Binding* to both, the *Segment Binding Operator* (*Sbo*) and the *Connector Binding Operator* (*Cbo*).

```
Binding ← Adapter Sbo ConnectorChain Sbo Adapter |
          Adapter Sbo Adapter
ConnectorChain ← Connector (Cbo Connector)*
Sbo ← "<-" | "->" | "<->"
Cbo ← "<-" | "->" | "<->"
```

Grammar 5.9: Syntax of *Two-Way Binding*

Listing 5.25 shows examples of how the new *Binding Operators* can be used inside a *Binding*. Again to keep it simple, the grammar does not prevent ill-formed *Bindings*, such as the one in line 4.

```
1 Adapter <-> Adapter
2 Adapter <-> Connector <-> Adapter
3 Adapter <-> Connector <-> Connector <-> Adapter
4 Adapter <- Connector <-> Connector -> Adapter
```

Listing 5.25: *Two-Way Binding* Examples

Two-Way Binding Semantics

A *Binding* using the *Two-Way Binding Operator* expresses the same semantics as two *Bindings* using opposing *One-Way Binding Operators*. The two replacing *Bindings* may appear in any order, since the order of *Bindings* is irrelevant to their semantics. Listings 5.26 and 5.27 show an example of how a *Binding Specification* can be converted into a semantically equal version not using *Two-Way Bindings*. Since the algorithm performing this conversion is trivial, we do not show it explicitly.

```
1  div {
2    input.name {
3      value <-> $name
4
5    }
6    input.age {
7      value <-> convert <-> $age
8
9    }
10 }
```

Listing 5.26: Before Conversion

```
1  div {
2    input.name {
3      value -> $name
4      value <- $name
5    }
6    input.age {
7      value -> convert -> $age
8      value <- convert <- $age
9    }
10 }
```

Listing 5.27: After Conversion

5.3.2. One-Time Binding

In addition to the *Two-Way Binding Operator* , we introduce the *One-Time Binding Operator* that allows to specify a *Binding* which is executed exactly once. This allows a more performant implementation of such *Bindings*, since their *Adapter* do not need to be watched for changes. Another use case of this concept is to declare *Binding Scope Adapters* and assign them an initial value as described in section 5.1.2. Examples for this are shown in listings 5.11 and A.2.

We want to note that this *Concept* is inspired by the *View Data Binding* functionality of *Windows Presentation Foundation*, which we examined in section 3.1.3.

One-Time Binding Syntax

We rework grammar 5.9 into grammar 5.10 for *Bindings* and add the two additional terminal symbols <∼ and ∼> to both *Segment Binding Operator* (Sbo) and *Connector Binding Operator* (Cbo). Listing 5.28 shows syntactically valid examples of *Bindings* using the *One-Time Operator*. Line 4, however, shows a *Binding* which is semantically wrong.

```
1 Adapter ∼> Adapter
2 Adapter <∼ Connector <∼ Adapter
3 Adapter ∼> Connector ∼> Connector ∼> Adapter
4 Adapter <∼ Connector ∼> Connector <-> Adapter
```

Listing 5.28: *One-Time Binding* Examples

```
Binding ← Adapter Sbo ConnectorChain Sbo Adapter |
          Adapter Sbo Adapter
ConnectorChain ← Connector (Cbo Connector)*
Sbo ← "<-" | "->" | "<->" | "<~" | "~>"
Cbo ← "<-" | "->" | "<->" | "<~" | "~>"
```

Grammar 5.10: Syntax of *One-Time Binding*

One-Time Binding Semantics

As with any other *Binding Operator*, the *One-Time Binding Operator* may not be mixed with other *Binding Operators*. A *Binding* using it therefore has the same semantics as if it would use a *One-Way Binding Operator* indicating the same direction. The only difference is that this *Binding* is only propagated once.

The *Concept* can be derived by either using an *Aborter* (`doOnce`) or a *Connector* that always returns the value it first received (`returnFirst`).

doOnce Remembers if it has ever produced an output. If not, its output is equal to its input. If it has produced an output before, it acts as an *Aborter* as described in section 5.1.2.

returnFirst Remembers if it has ever produced an output and what that output was. If the *Connector* is executed for the first time, it just returns its input and for any subsequent call again returns the input of its first execution.

Any *Binding* can now be transformed by placing one of the two *Connectors* after the *Source Adapter* of the *Binding*. Listings 5.29 and 5.30 illustrate this conversion. To keep the example abstract, we use the non-terminals *Adapter* and *Connector* and abbreviate them with `A` and `C` respectively.

```
1  A ~> A
2
3
4  A <~ A
5
6
7  A <~ C <~ A
8
9
10 A ~> C ~> A
11
12
13 ...
```

Listing 5.29: Before Conversion

```
1  A -> doOnce -> A
2  A -> returnFirst -> A
3
4  A <- doOnce <- A
5  A <- returnFirst <- A
6
7  A <- C <- doOnce <- A
8  A <- C <- returnFirst <- A
9
10 A -> doOnce -> C -> A
11 A -> returnFirst -> C -> A
12
13 ...
```

Listing 5.30: After Conversion

5.3.3. Resource Sequence

Resources are *Adapter* and with *Resource Sequence* we allow that multiple *Adapter* are used as *Data Source* or *Data Sink*. This way *Connectors* may obtain their inputs from multiple sources and produce outputs for more than one sink.

Resource Sequence Syntax

To implement *Resource Sequences* we modify the latest version of grammar 5.10 for *Bindings* into grammar 5.11 by simply replacing every *Adapter* with a *Resource Sequence*. The *Resource Sequence* itself is nothing but a list of *Adapter* separated by commas.

```
Binding ← ResourceSequence Sbo ConnectorChain Sbo ResourceSequence |
          ResourceSequence Sbo ResourceSequence
ResourceSequence ← Adapter ("," Adapter)*
ConnectorChain ← Connector (Cbo Connector)*
Sbo ← "<-" | "->" | "<->" | "<∼" | "∼>"
Cbo ← "<-" | "->" | "<->" | "<∼" | "∼>"
```

Grammar 5.11: Syntax of *Resource Sequence*

Resource Sequence Semantics

To define the meaning of *Resource Sequences* we need to differentiate between two different cases. One of them considers *Bindings* that do not include a *Connector Chain*, and the other such that do.

Without *Connector Chain* The semantics of the *Binding*

```
@in1, @in2, ..., @inN -> @out1, @out2, ..., @outM
```

is equal to the semantics of the following set of *Bindings*

$$B = \{b | b = \texttt{@in}_\text{i}\ \texttt{->}\ \texttt{@out}_\text{j} \wedge i = j \wedge 1 \leq i \leq N \wedge 1 \leq j \leq M\}$$

Obviously $|B| = max(N, M)$. This means that if there are more *Data Source* than *Data Sink Adapters* they will be dropped. Also, if the opposite is true, the superfluous *Data Sink Adapters* will never receive a value through that *Binding*. Assuming that $N = M$, above *Binding* could be transformed to the set of *Bindings* shown in 5.31.

```
1 @in1 -> @out1
2 @in2 -> @out2
3 ...
4 @inN -> @outM
```

Listing 5.31: Transformation of *Resource Sequence* to *Core Concepts*

With *Connector Chain* It is irrelevant if only one or more *Connectors* are in the *Connector Chain*, since a *Connector Chain* could always be expressed by a single *Connector* (see section 5.1.2). We therefore only illustrate the semantics of cases with one *Connector*.

Intuitively, instead of receiving one value during *Propagation* a *Connector* receives a list of input values and may also produce a list of output values. There are three cases that we need to consider separately.

1. `@in1, @in2, ..., @inN -> connector -> @out`
   There is a *Resource Sequence* as the input of the *Connector*, but a single *Resource* as the output.
2. `@in -> connector -> @out1, @out2, ..., @outN`
   The input is a single *Resource*, while the output is a *Resource Sequence*.
3. `@in1, @in2, ..., @inN -> connector -> @out1, @out2, ..., @outM`
   Both input and output of the *Connector* are *Resource Sequences*.

For the first case we need to assume that there are *Adapter* `$coll1`, `$coll2` ... `$collN` which yield lists of length N, while the values of the list items are irrelevant. Also, we assume that there are the *Connectors* `get1`, `get2`, ... `getN`, which pick the N-th item from a list and preserve its reference as described in section 5.1.2. The *Binding* can now be transformed as shown in listing 5.32.

```
1  @temp1 <- get1 <- $collN
2  @in1 -> @temp1
3
4  @temp2 <- get2 <- $collN
5  @in2 -> @temp2
6
7  ...
8
9  @tempN <- getN <- $collN
10 @inN -> @tempN
11
12 $collN -> connector -> @out
```

Listing 5.32: Transformation of *Resource Sequence* to *Core Concepts* (Case 1)

The second case is more simple and makes use of the same *Filters* `get1`, `get2`, ..., `getN`. The *Binding* can be transformed into a set of *Bindings* as shown in listing 5.33.

```
1 @in -> connector -> get1 -> @out1
2 @in -> connector -> get2 -> @out2
3 ...
4 @in -> connector -> getN -> @outN
```

Listing 5.33: Transformation of *Resource Sequence* to *Core Concepts* (Case 2)

Case 3 then is just a combination of the previous two. For the sake of completeness it is shown in listing 5.34.

```
1  @temp1 <- get1 <- $collN
2  @in1 -> @temp1
3
4  @temp2 <- get2 <- $collN
5  @in2 -> @temp2
6
7  ...
8
9  @tempN <- getN <- $collN
10 @inN -> @tempN
11
12 $collN -> connector -> get1 -> @out1
13 $collN -> connector -> get2 -> @out2
14 ...
15 $collN -> connector -> getN -> @outN
```

Listing 5.34: Transformation of *Resource Sequence* to *Core Concepts* (Case 3)

Due to the transformation, the interpretation of the output and input values of *Connectors* in *Bindings* comprising *Resource Sequences* changes. In the first case the *Connector* receives a list of input values and its single output value is assigned to `@out`, no matter if it is a list or not. In the second case, however, the *Connector* receives a single input and is expected to produce a list as its result which is then distributed to the output *Resource Sequence*. This means that the function of a *Connector* must be specifically designed to work with a certain combination of *Resource Sequences*.

Listing 5.35 shows some use cases of the *Concept*.

```
1 @dirtyValue                -> validate -> @sanitizedValue, @valueValid
2 @fullName                  -> split    -> @firstName, @lastName
3 @time, @day                -> makeDate -> @date
4 @boxFilled, @acceptChecked -> and      -> @formValid
5 @num1, @num2, ..., @numN   -> stats    -> @sum, @average, @median
```

Listing 5.35: Use Cases for *Resource Sequences*

Before concluding this section, we want to explain why we decided not to allow *Connector Sequences*. Consider the following hypothetical example.

```
@a, @b, @c -> connectorA, connectorB -> @d, @e, @f
```

This *Binding* is highly ambiguous. It is unclear which of `@a`, `@b` or `@c` act as the inputs of `connectorA` or `connectorB`. Also, it is not recognizable how to distribute the outputs of the two *Connectors* to `@d`, `@e` or `@f`.
To achieve this, every *Connector* would have to specify exactly ...

... how many inputs and outputs are produced at most and at least.

... how many inputs and outputs are produced depending on the number of inputs or outputs respectively.

... which number of inputs or outputs is preferred in case there are several options.

To avoid this complexity, which would also require a powerful algorithm to decide on a matching, we decided to not include *Connector Sequences*. Furthermore they would hinder maintainability and not increase expressiveness, since it is always possible to split such a *Binding* into multiple ones. This makes the matching explicit and is much easier to understand. One example of splitting the above example would be as follows.

```
@a -> connectorA -> @d, @e

@b, @c -> connectorB -> @f
```

5.3.4. Initiator

A *Binding* becomes active as soon as one of its *Data Source Adapter* notifies it that its value has changed (see section 5.1.2). An *Initiator* is an *Adapter* that is not used for reading or writing values, but only for initiating the *Propagation* of a *Binding*. It is always attached to an *Adapter* and replaces that *Adapter*'s role as an *Initiator*.

Initiator Syntax

An *Initiator* could be theoretically attached to any *Adapter* in a *Binding*. At the moment, there are only *Data Source* and *Data Sink*, but with the introduction of *Parameters* (see section 5.3.5) they might appear in other locations of a *Binding* as well. To ensure that a *Binding* remains readable we decided that an *Initiator* may only be attached to the *Data Source* and *Data Sink*.

Syntactically an *Initiator* is nothing but a *Resource Sequence* attached to another *Resource Sequence* by a special *Initiator Binding Operator*. Both of these *Resource Sequences* might consist of only one *Adapter*. To include the concept, grammar 5.11 for *Bindings* needs to be reworked into grammar 5.12. *IRS* is an abbreviation for both *Initiated Resource Sequence* and *Initiator Resource Sequence*, while *RS* is the shorthand for *Resource Sequence*. Listing 5.36 shows examples of how *Initiators* may be used inside *Bindings*.

```
Binding ← IRSleft Sbo ConnectorChain Sbo IRSright |
          IRSleft Sbo IRSright
IRSleft ← (RS "+>")? RS
IRSright ← RS ("<+" RS)?
RS ← Adapter ("," Adapter)*
ConnectorChain ← Connector (Cbo Connector)*
Sbo ← "<-" | "->" | "<->" | "<∼" | "∼>"
Cbo ← "<-" | "->" | "<->" | "<∼" | "∼>"
```

Grammar 5.12: Syntax of *Initiator*

```
1 @i1 +> @a1 -> @a2
2 @a1 <- @a2, @a3 <+ @i1
3 @i1, @i2 +> @a1, @a2 <-> c <-> @a3, @a4 <+ @i3
```

Listing 5.36: *Initiator* Examples

Semantically it only makes sense to attach an *Initiator* to the *Data Source* of a *Binding*, but the grammar is not restricted to that to support *Two-Way Bindings*. Since *Data Sink Adapter* never trigger the *Propagation* of a *Binding*, an *Initiator* used there just has no effect and is redundant.

Initiator Semantics

If an *Initiator* consists of a *Resource Sequence*, its semantics are as if the *Binding* was written multiple times each time with one of the *Adapter* from the *Resource Sequence* as their *Initiator*. If the *Resource Sequence* that an *Initiator* is attached to consists of more than one *Adapter*, the *Initiator* applies to all elements. Therefore, we reduce our explanation of the semantics to the case where neither *Resource Sequence* consists of more than one *Adapter*.

```
Initiator +> DataSource -> ...  -> DataSink
```

Intuitively, the *Initiator* and the *Data Source Adapter* are merged together into a new *Adapter* which behaves like the *Data Source Adapter*. The only difference is that *Propagation* is not triggered when the underlying value of the *Data Source Adapter* changes, but when the underlying value of the *Initiator Adapter* changes.

We decided on this replace semantic instead of triggering *Propagation* for both *Data Source Adapter* and *Initiator* to be as flexible as possible. If it should be required to add the trigger instead of replacing it, the *Data Source Adapter* may always be duplicated to the *Initiator Resource Sequence*.

```
Initiator, DataSource +> DataSource -> ...  -> DataSink
```

Expressing an *Initiator* with the concepts already introduced can be easily done by adding it to the *Initiated Resource Sequence* to initiate *Propagation*. To not alter the

semantics of the *Binding* a *Connector* `drop` is added to remove the values read from the *Initiators*. Listings 5.37 and 5.38 visualize the conversion.

```
1 @foo +> @bar ->
2   ... -> @baz
3
4 @i1, @i2, ..., @iN +>
5   @a1, @a2, ..., @aM ->
6     ... -> @t
```

Listing 5.37: Before Conversion

```
1 @bar, @foo -> dropSecond ->
2   ... -> @baz
3
4 @a1, @a2, ..., @aM,
5   @i1, @i2, ..., @iN ->
6     dropLastN -> ... -> @t
```

Listing 5.38: After Conversion

There are several use cases. One is inspired by Windows Presentation Foundation (see section 3.1.3) which offers a feature to imperatively execute certain *Bindings*. With an *Initiator* we can solve this by using a *Presentation Model Adapter* as an *Initiator*. If the value inside the *Presentation Model* is changed, the *Binding* starts to propagate.

A more practical use case arises from the area of web development. It is often required that values are synchronized with the *Presentation Model* only if certain events happen. For instance, if a user enters content into a text box, the value should be written to the *Presentation Model* whenever a key is pressed or not until the text box loses focus. To express this, assume that there is an *Adapter* called `on` that provides the last event fired by an element from the *View Mask*. As its *Qualifier* it takes the type of event. With this we could express the previous two cases as follows.

```
on:keydown +> value <-> $value
```

```
on:change +> value <-> $value
```

5.3.5. Parameter

With this concept we allow that both *Adapter* and *Connectors* may have *Parameters*, which are *Adapter*. Thus, it is possible to configure the behavior of *Adapter* and *Connectors*.

Parameter Syntax

Parameters are a comma-separated list of *Adapter* specified after another *Adapter* or *Connector* enclosed in parentheses. Parameter may be positional or name-based. To define them we rework both grammar 5.2 for *Adapter* and grammar 5.3 for *Connectors* into grammar 5.13. Listing 5.39 shows abstract examples of how to use *Parameters* in *Bindings*. With these examples we want to make clear that positional and name-based parameters may be mixed and that *Adapter* that are *Parameter* can again have *Parameter*.

```
Adapter ← AdapterId Parameters?
Connector ← Id Parameters?
Parameters ← "(" Parameter ("," Parameter)* ")"
Parameter ← Adapter | Id "=" Adapter
AdapterId ← [@$#%&] [a-zA-Z_] ([a-zA-Z0-9_-])* |
            [a-zA-Z_] ([a-zA-Z0-9_])*
             (":" [a-zA-Z_] ([a-zA-Z0-9_-])*)?
Id ← [a-zA-Z_] ([a-zA-Z0-9_])*
```

Grammar 5.13: Syntax of *Parameter*

```
1 @a(@b)      -> connector(@c)      -> @d(@e)
2 @a(@b, @c) -> connector(d = @e) -> @f(@g, @h)
3 @a(b = @c) -> connector(@d(@e)) -> @d(e = @f)
4
5 @a(b = @c, d = @e)          -> @f(g = @h, i = @j)
6 @a(@b, c = @d, @e, f = @g) -> @h(i = @j, @k, l = @m, @n)
7 @a(@b(@c), d = @e(f = @g)) -> @h(i = @j(@k))
```

Listing 5.39: *Parameter* Examples

Parameter Semantics

We first convert all positional *Parameter* into name-based *Parameter*, with their index as their name. The set of *Parameters* can then always be converted into one value, containing all *Parameters* as an associative array. We, therefore, only show how the concept is derived for the case that there is a single positional parameter.

Every *Adapter* defines the logic that is executed when a value is written to the *Adapter* (see section 5.1.2). Therefore, this logic could as well set an internal configuration of the *Adapter* instead of writing to its *Data Target*. We assume that the *Connector* `makeParam` is able to add information to a value in such a way that the *Adapter* recognizes it as a configuration *Parameter* instead of a value that should be written to the *Data Target*.

We can then resolve *Parameters* for *Adapter* in a bottom-up fashion by passing them to the *Adapter* explicitly. To illustrate this process consider the *Binding* from listing 5.40 which can be transformed into a set of *Bindings* as in listing 5.41.

```
1 @a(@b(@c(...))) -> @d
```

Listing 5.40: Before Conversion

```
1 ...
2 @c -> makeParam -> @b
3 @b -> makeParam -> @a
4 @a -> @d
```

Listing 5.41: After Conversion

For *Connectors* the process is different and needs to take special care about *Connector Chains* because they have to be split up into multiple *Bindings*. Consider the example from listing 5.42 first. For the *Connector* it makes no difference if `@b` is written as a *Parameter* or as an additional item of the input *Resource Sequence*. This means that the *Binding* can be transformed into the *Binding* from listing 5.43.

```
1 @a <- c(@b) <- @c
```

Listing 5.42: Before Conversion

```
1 @a <- c <- @b, @c
```

Listing 5.43: After Conversion

Now for the more complex example of a *Connector Chain* where each *Connector* has *Parameters* we only show a case with two *Connectors* in listing 5.44. The process follows naturally if more than two *Connectors* are in the *Connector Chain*.

```
1 @a <- c1(@b) <- c2(@c) <- @d
```

Listing 5.44: Before Conversion

Since *Connector Sequences* are not allowed (see section 5.3.3), the process described earlier does not work and the *Binding* needs to split into two parts as in listing 5.45. Now the same logic as earlier can be applied twice to arrive at the two *Bindings* from listing 5.46.

```
1 @a <- c1(@b) <- @temp
2 @temp <- c2(@c) <- @d
```

Listing 5.45: Intermediate Step

```
1 @a <- c1 <- @b, @temp
2 @temp <- c2 <- @c, @d
```

Listing 5.46: After Conversion

If we look back at *Initiators* it becomes more clear now that they cannot be used with *Adapter* which have the role of a *Parameter*. This restriction, however, can be circumvented if the transformation is done manually because there every *Adapter* that was a *Parameter* can be modified by an *Initiator*. Also, we would like to note that we decided not to express *Initiators* as *Parameters*, since it would make them appear to be relevant for both directions inside a *Two-Way Binding*. By placing them in a designated location left or right of the *Binding* their role becomes more clear syntactically. Use cases for *Parameters* include configuration and default values as shown in listing 5.47.

```
1 // Replace $text with correct localization
2 text <- i18n($lang) <- $text
3
4 // Decide whether HTML should be escaped
5 text(escapeHtml = $escapeHtml) <- $text
6
7 // Configuration parameter and default value
8 on:keydown(timeout = $waitMillis) +> value <-> $text($defaultText)
```

Listing 5.47: *Parameter* Use Cases

5.3.6. Expression

The goal of *Expressions* is to dramatically increase readability and usability of our Concepts. As our examples of *Bindings* using *Expressions* and how they are expressed by using just *Core Concepts* show, *Expressions* also greatly reduce the amount of code that is necessary to express complex *Bindings*.

Expression Syntax

Syntactically *Expressions* are allowed wherever *Adapter* were possible until now with the exception of the *Entry* and *Key Adapter* of an *Iteration*. This means that we need to change the grammars 5.6 for *Iteration*, 5.12 for *Initiators* and 5.13 for *Parameters* to include *Expressions*. Grammar 5.14 shows the relevant parts that have to be updated.

```
Iteration ← "(" (Entry ("," Key)? ":")? Collection ")"
Entry ← Adapter
Key ← Adapter
Collection ← Expr
...
Binding ← IRSleft Sbo ConnectorChain Sbo IRSright |
          IRSleft Sbo IRSright
IRSleft ← (RS "+>")? RS
IRSright ← RS ("<+" RS)?
RS ← Expr ("," Expr)*
...
Adapter ← AdapterId Parameters?
Connector ← Id Parameters?
Parameters ← "(" Parameter ("," Parameter)* ")"
Parameter ← Expr | Id "=" Expr

Expr ← ...
```

Grammar 5.14: Syntax of *Binding* with *Expression*

The most basic *Expressions* are static values, numbers, regular expressions and quoted string literals, which we all call *Literals*. Although we will not show their exact grammar for reasons of shortness, their syntax should be very intuitive. Table 5.4 gives an overview of *Literals*.

Literals are the basic elements of an *Expression*. All other syntactical elements of an *Expression* recursively structure one or more *Literals*. Before introducing the Parsing Expression Grammar for *Expression*, we want to give a qualitative overview of the remaining *Expression* that are supported with table 5.5.

Name	Description	Examples
Static Value		`true` `false` `null` `NaN` `undefined`
Number	Numeric values supporting signs, decimal points and exponents. Further hexadecimal, octal and binary numbers are possible.	`+314.592654e-2` `0xABAD1DEA` `0b101010`
Regular Expression	Special string literal, that requires less escaping	`/[a-z]/` `/\[A-Z0-9]+@[A-Z0-9]+[̇A-Z]2,4\b/`
Quoted String	Might comprise only ASCII characters or escapes for UTF-8 encoded characters	`"foo"` `'foo\tbar'` `"25\uE282AC"`

Table 5.4.: Overview of *Literals*

Name	**Syntax**	**Examples**
Conditional	`Expression ? Expression : Expression` `Expression ?: Expression`	`$checked ? $password : "*****"` `$name ?: "Please enter name"`
Logical	`! Expression` `Expression && Expression` `Expression || Expression`	`!$checked` `$checked && $valid` `$toBe || !$toBe`
Relational	`Expression == Expression` `Expression != Expression` `Expression <= Expression` `Expression >= Expression` `Expression < Expression` `Expression > Expression`	`$name == "admin"` `$duration != 0` `$age <= 120` `$end >= $start` `!($money < $cost)` `$amount > 0`
Additive	`Expression + Expression` `Expression - Expression`	`$price + $tax` `$price - $discount`
Multiplicative	`Expression * Expression` `Expression / Expression` `Expression % Expression`	`$quantity * $price` `$sea / 2` `$people % $groupSize`
Dereference	`Expression(.Id)+` `Expression([Expression])+`	`@person.name` `@person["name"]`
Array	`[Expression (, Expression)*]`	`[$name, "Tom", 5]`
Hash	`{(Id:Expression)? (, Id:Expression)*}`	`{name: $name, age: 25 }`
Parenthesis	`(Expression)`	`!(@foo && (@bar || @baz))`

Table 5.5.: Overview of *Expressions*

Since Parsing Expression Grammar does not allow left recursion, we cannot specify a rule like `Expression ← Expression "+" Expression`. We, therefore, use a trick of making a round trip over all available expressions. Finally, grammar 5.15 shows how the syntax of *Expressions* is defined and should make the idea of the round trip clearer.

```
Expr ← Conditional
Conditional ← Logical "?" Expr ":" Expr |
              Logical "?:" Expr |
              Logical
Logical ← "!" Expr |
          Relational (LogicalOp Expr)+ |
          Relational
LogicalOp ← "&&" | "||"
Relational ← Additive (RelationalOp Expr)+ |
             Additive
RelationalOp ← "==" | "!=" | "<=" | ">=" | "<" | ">"
Additive ← Multiplicative (AdditiveOp Expr)+ |
           Multiplicative
AdditiveOp ← "+" | "-"
Multiplicative ← Dereference (MultiplicativeOp Expr)+ |
                 Dereference
MultiplicativeOp ← "*" | "|" | "%"
Dereference ← Other ("." Id)+ |
              Other ("[" Expr "]")+ |
              Other
Other ← Literal | Array | Hash | Adapter | Parenthesis
Literal ← QuotedString | Regexp | Number | Value
Array ← "[" (Expr ("," Expr)*)? "]"
Hash ← "{" (HashKV ("," HashKV)*)? "}"
HashKV ← Id ":" Expr
Parenthesis ← "(" Expr ")"
```

Grammar 5.15: Syntax of *Expression*

Expression Semantics

To explain the semantics of *Expressions* we proceed in a bottom-up fashion. We start with *Literals* and work our way up to the compound *Expressions*.

All *Literals* are interpreted as if they were *Adapter* that cannot be used as the *Data Sink*. To reach this we assume that there is a certain attribute in the *Presentation Model* for each *Literal* used in a *Binding*. We call the *Adapter* pointing to this value `$literal`. Since this *Adapter* may never be written both directly or indirectly through references, we introduce a *Connector* `makeValue` which converts the *Reference* that is retrieved from the *Adapter* into a value. It is then possible to replace any *Literals* with a *Binding Scope Adapter* that was assigned with the *Literal*. Consider the following example from listing 5.48 that can be easily converted into the two *Bindings* from listing 5.49.

```
1 attr:id <- "id"
```

Listing 5.48: Before Conversion

```
1 @temp <- makeValue <- $literal
2 attr:id <- @temp
```

Listing 5.49: After Conversion

Since *Literals* are *Expressions* and *Expressions* are syntactically only allowed where *Adapter* are possible, this conversion always resolves *Literals*.

Conditional, *Logical*, *Relational*, *Additive* and *Multiplicative* can all be converted by first converting their sub *Expressions*. This means that we recursively convert the parts that they are made of first and can assume that they consist of only *Adapter* now. Each of them can then be realized by a *Connector* which stores its result in the *Binding Scope*. This *Binding Scope Adapter* then replaces the original expression.

We only show the conversion for a *Conditional*. All other mentioned types of *Expressions* should follow naturally. The *Connector* called `ifThenElse` expects as its input a *Resource Sequence* of length three and operates as we would intuitively expect. Listing 5.50 shows a *Binding* using a *Conditional* which can be easily transformed into the two semantically equal *Bindings* from listing 5.51. It should be noted that it depends on the implementation of `ifThenElse`, whether the *Conditional* is evaluated eagerly or lazily. The transformation itself does not impose eager evaluation, since the *Connector* only receives a list of three *References* as its input.

```
1 @foo <- @bar ? @baz : @quux
```

Listing 5.50: Before Conversion

```
1 @temp <- ifThenElse <-
2   @bar, @baz, @quux
3 @foo <- @temp
```

Listing 5.51: After Conversion

The *Array* and *Hash Expressions* can be expressed by using a *Connector* that creates the desired data structure from multiple inputs. Consider the following example from listing 5.52 which contains both an *Array* and a *Hash*. It can be easily transformed into the set of *Bindings* from listing 5.53.

```
1 @foo <- [@bar, {baz: @quux}]
```

Listing 5.52: Before Conversion

```
1 @t1 <- genHash("baz") <- @quux
2 @t2 <- genArray <- @bar, @t1
3 @foo <- @t2
```

Listing 5.53: After Conversion

The *Dereference* can be interpreted as a *Connector* `addToPath` which adds all `Ids` to the *Path* that it reads from the *Expression*. Consider the example from listing 5.54 which intentionally is a bit more complex to also give another example on how to convert a *Conditional* and a *Relational*. The names of the temporary variables from listing 5.55 are only to make it easier to comprehend. An automatic conversion would just number the variables similar to the algorithm shown in section 5.1.2.3. The example also shows how powerful *Expressions* are and how much code can be saved when using them.

```
1 @person <-
2   @people[0].age
3     > @people[1].age ?
4   @people[0] :
5   @people[1]
```

Listing 5.54: Before Conversion

```
1  @person0 <- addToPath(0)
2    <- @people
3
4  @person1 <- addToPath(1)
5    <- @people
6
7  @person0Age <- addToPath("age")
8    <- @person0
9
10 @person1Age <- addToPath("age")
11   <- @person1
12
13 @p1OlderP2 <- gt
14   <- @person0Age, @person1Age
15
16 @person <- ifThenElse
17   <- @p1OlderP2, @person0,
18        @person1
```

Listing 5.55: After Conversion

Parenthesis control the order in which compound *Expressions* are resolved recursively. Implicitly every *Expression* is evaluated from left to right.

It is important to note that whether an *Expression* might occur as the *Data Sink* solely depends on how its conversion to *Core Concepts* is defined. *Conditional, Dereference, Array* and *Hash* are transformed in a way that allows the *Connector* involved to preserve *References*. So all *Bindings* from listing 5.56 are semantically valid and make sense.

```
1 value -> @condition ? $target : $alternative
2 $people[0].age <-> @firstPersonAge
```

Listing 5.56: Before Conversion

All other *Expression* cannot be used as *Data Sink*. Obviously, it does not make sense to write *Literals*, but also for other *Expression* in most cases, the original values could not possibly be recovered. Consider as an example an addition of two numbers to one number. It is impossible to recover the summands from the sum unequivocally.

5.4. Convenience Structure Concepts

In this section we introduce our only *Convenience Structure Concept* which allows to use *Templates*, a way of structuring components on the level of *View Masks*.

5.4.1. Template

With *Templates* it is possible to mark a certain part of a *View Mask* and its associated section of a *Group* as a blueprint for the same or another *View Mask*. Since the *Template* may then be injected at another place once or multiple times, less code has to be written and reusing elements becomes possible.

Template Syntax

Both, *Template Extraction* and *Injection* are written as an additional information of a *Scope*. Therefore, we rework grammar 5.8 for *Scopes* into a new version 5.16 including *Templates*.

```
Scope ← Selectors Iteration? TplExtraction? "{" ScopeBody* "}" |
        Selectors (Iteration | Label | TplInjection | TplExtraction)
TplInjection ← "<<" Id ("." Id)* "(" RS? ")"
TplExtraction ← ">>" Id "(" (Adapter ("," Adapter)*)? ")"
```

Grammar 5.16: Syntax of *Scope* with *Templates*

Listing 5.57 shows an example how templates may be used together with the *Template DOM Fragment* from listing 5.58. It should be noted that the brackets are mandatory even if there are no parameters, so that the *Concept* can not be confused with a selector. In addition, defining a *Template* without parameters can be seen as an edge case.

```
1 @binding foo {
2   #markup >> baz(@text) {
3     span { text <- @text }
4   }
5 }
6 @binding bar {
7   #hookOne << foo.baz("Hello!")
8   #hookTwo << foo.baz($name)
9 }
```

Listing 5.57: *Template* Example

```
1 <div id="template">
2   <div id="markup">
3     <span class="text"></span>
4   </div>
5   <div id="hookOne" />
6   <div id="hookTwo" />
7 </div>
```

Listing 5.58: *Template DOM Fragment*

To make the example easier both *Template Extraction* and *Injection* happen within the same *Template DOM Fragment*. In a real application this would probably be not the case.

Template Semantics

The *Adapter* which are specified like parameters of a *Template Extraction* can be assigned with *Expressions* when *Injection* is used. We assume that the amount of parameters is in both cases the same. Semantically a new *Binding* is added for each parameter to reflect its assigned value. The children of the extracting *Scope* are appended to the injecting *Scope*. The descendants of the part matched by the *Selector* of the injecting *Scope* are replaced by the descendants of the part matched by the *Selector* of the extracting *Scope*. The extracted parts are removed from the *Binding Specification* and the *Template DOM Fragment*. We assume that all participating *Selectors* always exactly match one element of the *Template DOM Fragment*. Each *Template* has a name and is referenced by including the names of its surrounding *Groups* in a dot-notation.

To illustrate this we convert the example from listings 5.57 and 5.58 into listings 5.59 and 5.60 respectively.

```
1  @binding foo { }
2  @binding bar {
3    #hookOne {
4      @text <- "Hello!"
5      span { text <- @text }
6    }
7    #hookTwo {
8      @text <- $name
9      span { text <- @text }
10   }
11 }
```

Listing 5.59: *Binding Specification*

```
1  <div id="template">
2    <div id="hookOne">
3      <span class="text"></span>
4    </div>
5    <div id="hookTwo">
6      <span class="text"></span>
7    </div>
8  </div>
```

Listing 5.60: *Template DOM Fragment*

While other libraries allow to dynamically decide which *Template* should be used we do not allow this. The reason is that then *Template* is not a *Convenience Concept* anymore which easily can be resolved with a simple preparation step. Therefore a *Template* is always referenced by an *Id*, instead of an *Expression*.

Another restriction – similar to *Iteration* (see section 5.1.3) – is that the *Binding Scope Adapter* used during *Extraction* may not be used in any ancestor *Scope*. An example of this is the *Adapter* `@text` in listing 5.57 which must not be used in any of `#markup`'s ancestor *Scopes*.

5.5. Domain Specific Language

After reworking the grammar for a *Binding Specification* multiple times while gradually introducing all of our *Concepts*, we want to conclude this chapter by combining all revisions into a final grammar 5.17. Figure 5.4 is an annotated example of a *Binding Specification* to provide a visual reference for most of the terms from our ontology and domain specific language.

```
BindingSpecification ← Blocks
Blocks ← (Block)*
Block ← Group | Scope
Group ← Identification "{" Blocks"}"
Identification ← "@binding " GroupIdentifier
GroupIdentifier ← [a-zA-Z_]([a-zA-Z0-9_-])*
Scope ← Selectors Iteration? TplExtraction? "{" ScopeBody* "}" |
        Selectors (Iteration | Label | TplInjection | TplExtraction)
Selectors ← Selector ("," Selector)*
ScopeBody ← Block | Binding
Iteration ← "(" (Entry ("," Key)? ":")? Collection ")"
Entry ← Adapter
Key ← Adapter
Collection ← Expr
Label ← "::" LabelIdentifier
LabelIdentifier ← [a-zA-Z_]([a-zA-Z0-9_-])*
TplInjection ← "<<" Id ("." Id)* "(" RS? ")"
TplExtraction ← ">>" Id "(" (Adapter ("," Adapter)*)? ")"
Binding ← IRSleft Sbo ConnectorChain Sbo IRSright |
          IRSleft Sbo IRSright
IRSleft ← (RS "+>")? RS
IRSright ← RS ("<+" RS)?
RS ← Expr ("," Expr)*
ConnectorChain ← Connector (Cbo Connector)*
Sbo ← "<-" | "->" | "<->" | "<∼" | "∼>"
Cbo ← "<-" | "->" | "<->" | "<∼" | "∼>"
Adapter ← AdapterId Parameters?
Connector ← Id Parameters?
Parameters ← "(" Parameter ("," Parameter)* ")"
Parameter ← Expr | Id "=" Expr
AdapterId ← [@$#%&] [a-zA-Z_] ([a-zA-Z0-9_-])* |
            [a-zA-Z_] ([a-zA-Z0-9_])*
             (":" [a-zA-Z_] ([a-zA-Z0-9_-])*)?
Id ← [a-zA-Z_] ([a-zA-Z0-9_])*
Expr ← (see grammar 5.15)
```

Grammar 5.17: *Binding Specification*

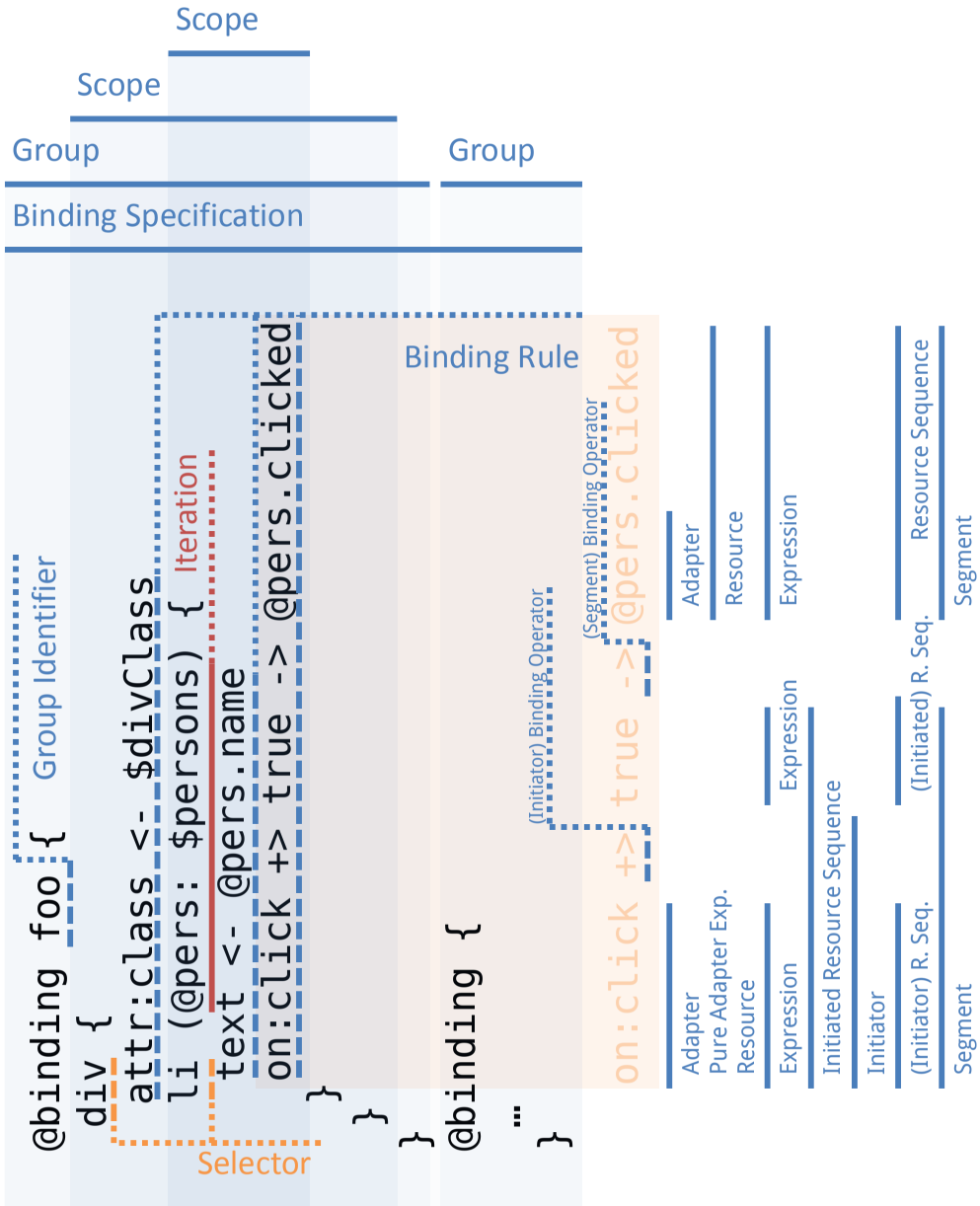


Figure 5.4.: Annotated Binding Specification

6. Implementation, BindingJS

This chapter is all about demonstrating how the previously defined *Concepts* can be put into action. We do so with a JavaScript library called *BindingJS*, which is publicly accessible at bindingjs.org. It is executed by a web browser on the client. In this chapter we first describe the architecture of *BindingJS* and show how its various components interact. After that, we explain its API and how it can be operated by a user. Finally, the algorithms that operate behind the scenes of *BindingJS* are shown and clarified. While we plan on implementing all *Concepts* with *BindingJS* in the future, only the implementation of *Core Concepts* is considered in this chapter. All *Convenience Concepts* may be implemented by performing their transformation to *Core Concepts* (see chapter 5) in a preparation step. This, however, might not always be the most efficient way to realize them.

6.1. Architecture

In this section we outline and name the components that make up *BindingJS* with the help of context diagrams, showing how it interacts with peripherals and with the help of component diagrams, illustrating its internal structure. We also give a short introduction about the technologies that are used, how *BindingJS* is tested and how it can be debugged.

Context

The goal of *BindingJS* is to bind a *Presentation Model* to a *DOM Fragment*. The *Binding Specification* and *Template DOM Fragment* are inputs to the library and can be seen as static instructions on how to perform this binding or synchronization. These relationships are visualized in figure 6.1. It also shows the flow of data. Since these elements are inputs, information is never written to the *Template DOM Fragment* or *Binding Specification*. The *DOM Fragment* and *Presentation Model* on the other hand are both read and written.

To give a more intuitive understanding of the components which are utilized by *BindingJS* the figure also shows examples of how to represent each of them. We assume that both *DOM Fragment* and *Template DOM Fragment* always comprise HTML, while the *Binding Specification* can be recognized by our domain specific language from chapter 5. Since *BindingJS* is model-agnostic, the *Presentation Model* might be a *ComponentJS* (see section 2.1.3) or *Backbone* (see section 2.1.1) model, too, but for simplicity we usually show examples having a plain JSON object as their *Presentation Model*.

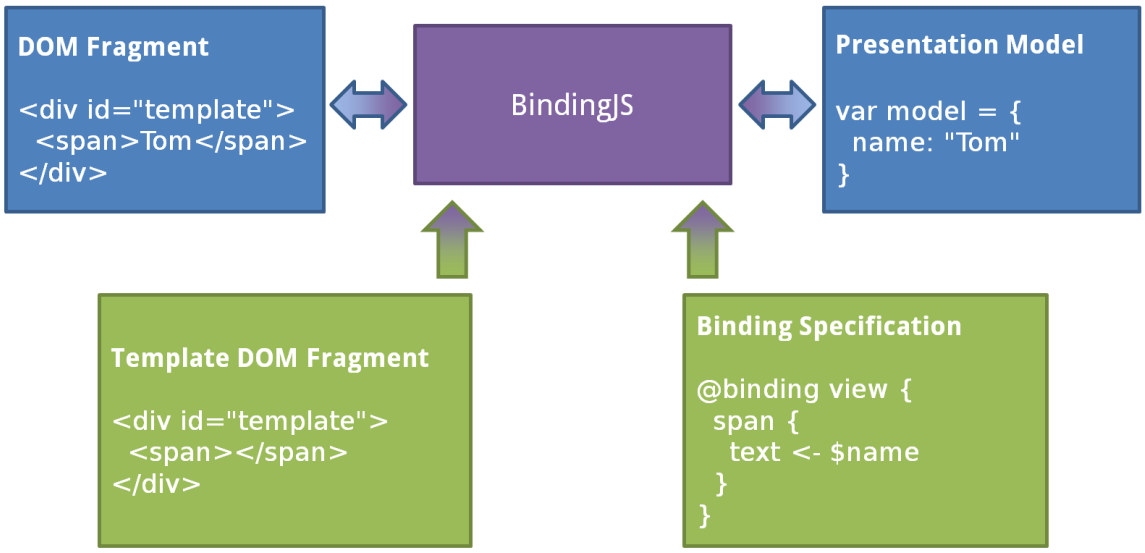


Figure 6.1.: BindingJS, Context Diagram

Components

Figure 6.2 shows how *BindingJS* is structured internally. The *Binding Specification* is first processed by a *Parser* (see section 6.3.3) that produces an abstract syntax tree (AST). This tree, together with the *Template DOM Fragment*, is converted into an *Iteration Tree* by the *Preprocessor* (see section 6.3.4). Last a component called *Engine* uses and manipulates the *Iteration Tree* to perform the actual *View Data Binding* with the information provided by two repositories, one for *Adapter* and another for *Connectors* (see section 6.3.1).

As we zoom deeper into the *Engine* in figure 6.3, more components are revealed. The task of the *Iterator* (see section 6.3.5) is to manipulate the *DOM Fragment* and to produce or delete *Iteration Instances* with the help of the *Binding Scope* (see section 6.3.2). The *Propagator* (see section 6.3.6) takes care about *Bindings* and their *Propagation* (see section 5.1.2). They are a part of *Iteration Instances* and their comprised *Adapter* and *Connectors* are resolved by looking them up in the appropriate repository. These *Adapter* may then read from any of the three depicted data sources *DOM Fragment*, *Binding Scope* or the *Presentation Model*.

Technologies and Dependencies

BindingJS is programmed in JavaScript, since it is the standard web programming language and supported by all web browsers, thus, making it ideal for our intended use in web applications. Since the behavior of JavaScript DOM manipulation is inconsistent across different web browsers, *BindingJS* depends on jQuery. This allows for portability, while keeping the implementation compact by avoiding browser specific code and instead using a high level API. jQuery is the only dependency and is present in most of todays web applications already. The share of web pages using jQuery was around 60 percent as of 1st of September 2014 [Webac].

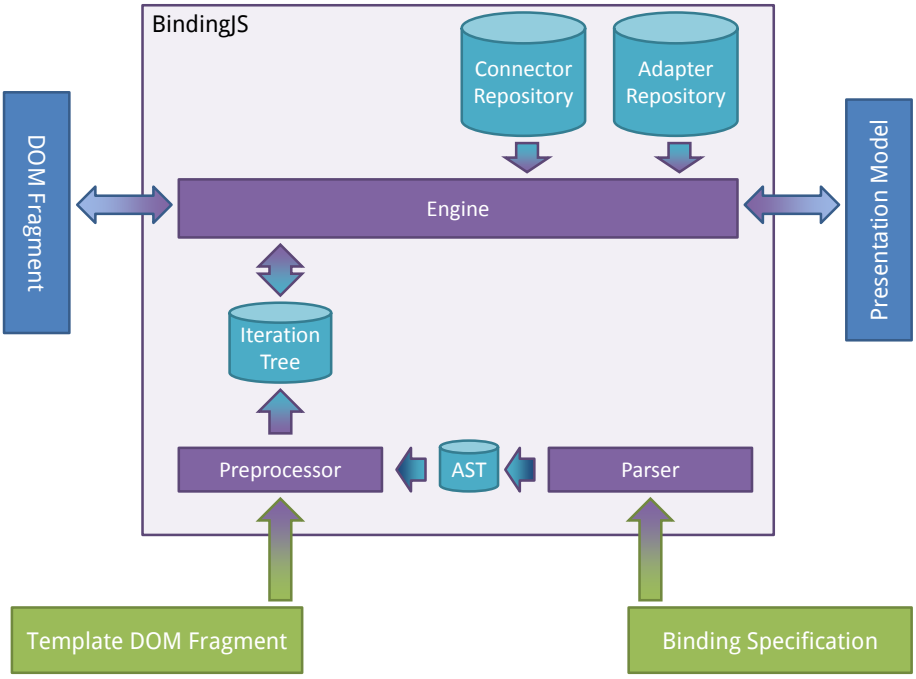


Figure 6.2.: BindingJS, Component Diagram

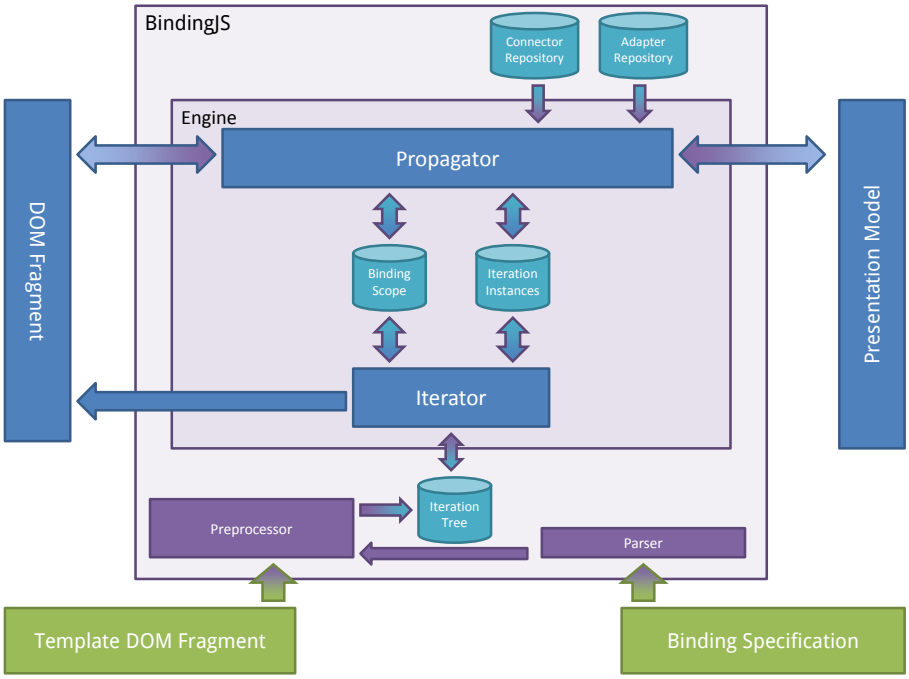


Figure 6.3.: BindingJS Engine, Component Diagram

In addition, we took care that our solution is still usable even when jQuery would otherwise cause trouble by supporting its no conflict strategy[1] (see section 6.2).

Development dependencies are managed with *node.js* and the project is built with *Grunt*. The build process is configured so that it comprises lint and minification done by *UglifyJS*. The result of the build process is a set of minified JavaScript files including the library itself and a selection of *Adapter* and *Connector* implementations. These can all be included in any web page. *BindingJS* is open source and uses *Git* for revision control. More information about the technologies can be found on the web pages given in table 6.1.

Name	Official Page
node.js	http://nodejs.org/
Grunt	http://gruntjs.com/
UglifyJS	https://github.com/mishoo/UglifyJS
Git	http://git-scm.com/
mocha	http://visionmedia.github.io/mocha/
Selenium IDE	http://docs.seleniumhq.org/projects/ide/

Table 6.1.: *BindingJS* Technology Overview

Testing

BindingJS uses *mocha* for unit and *Selenium IDE* for integration tests. The latter has the advantage over using DOM virtualization like *jsdom*[2] that it operates on a real browser in an almost identical fashion to an actual user and is much easier to setup and use. Listing 6.1 shows a very simple example of a test case written with the JavaScript framework *mocha*. Figure A.2 in the appendix demonstrates the *Selenium IDE* which is used for both defining and executing test cases.

Debugging

Input components, such as the *Binding Specification* and the *Template DOM Fragment*, are modified inside the library before being put into action. Therefore it could be difficult for a user to retrace the cause of problems. We provide mechanisms to aid finding problems which are either caused by erroneous input or internal processing errors.

- Whenever possible, we validate assumptions made in the implementation and replace meaningless with explanatory error messages.
- Users of the library may set an individual log level (see 6.2) to get more information about internal processing.

[1] http://api.jquery.com/jquery.noconflict/
[2] https://github.com/tmpvar/jsdom

- Once the *Binding Specification* is parsed into an abstract syntax tree, this tree is modified by both the *Preprocessor* and the *Engine*. To visualize these changes our implementation of the abstract syntax tree is able to render itself as a legible *Binding Specification*[3].
- During preprocessing several validation steps are performed (see section 6.3.4). These algorithms recognize restrictions and erroneous *Bindings* which have not already been prevented by the domain specific languages, such as those contained in listing 5.9.

Further, it should be noted that, since the code of *BindingJS* is open source, every user has the possibility to retrace problems by stepping through its execution. Also, with the project being hosted on GitHub support from the developers or other users is available through the social features of the platform.

```
1  describe("BindingJS DSL Parser", function () {
2    describe("parse()", function () {
3      it("allows the binding DSL to be parsed", function () {
4        // Setup
5        let dsl = fs.readFileSync(
6          path.join(__dirname, "parsertest.bind"),
7          { encoding: "utf8" }
8        )
9
10       // Perform
11       let ast = BindingJS.parse(dsl)
12
13       // Assert
14       expect(ast).to.be.a("object")
15     })
16   })
17 })
```

Listing 6.1: Example of a Mocha Testcase

[3] https://github.com/rummelj/bindingjs/blob/023b220246d797f24d0418d57a3c0c0c227a4fe1/src/core/util/tree.js#L266

6.2. Application User Interface

Before presenting the algorithms behind the components shown in the previous section, we want to point out how one can interact with *BindingJS* through its API. Figure 6.4 shows all relevant interfaces in a class diagram of the Unified Modeling Language.

6.2.1. Public API

By default, upon including the library the symbol `BindingJS` becomes globally available within the context of the web page. It provides the methods shown in the upper left box.

$ If no parameter is provided, a reference to the jQuery instance that *BindingJS* is currently working with is returned. This method is used for all internal accesses to jQuery and by default provides access to the globally available jQuery element of the web page. If, however, this is not available, it is possible to manually set jQuery by providing it as a parameter so that only the given reference is used. This is necessary if jQuery is registered under a different symbol or is not globally available at all.

symbol Similar to jQuery's no conflict strategy. This method allows to change the name of the global variable that *BindingJS* uses. By default this is `BindingJS` which can be changed to an arbitrary identifier. If no parameter is provided, *BindingJS* removes itself from the pool of global variables. By storing a reference beforehand conflicts and the pollution of the global JavaScript name space are prevented

version Provides a JSON object comprising information about the version of *BindingJS* including major, minor, micro version numbers and its build time.

debug This method is available in three different flavors. If no parameter is provided, the current log level is returned. With only the first parameter a new log level can be set and by adding a message it is logged on the given log level. This way it is possible to specify how much information is required and a consistent interface for providing log messages is defined. Typically, the log level will be set to a high value in a development and to a low value in a production environment.

plugin Through this method *Model View Adapter* and *Connectors* can be registered at the library. By defining the name of these components at the stage of registration, instead of specifying it as part of their implementations, we avoid that they have conflicting names and allow the greatest flexibility. All plug-ins have to implement one of the interfaces shown in the upper right of figure 6.4.

create This *Factory Method*[4] creates an instance of *View Data Binding*, whose interface is shown in the lower left of figure 6.4.

Whenever possible, we provide a fluent API so that consecutive calls can be chained into one statement. Therefore, if there would otherwise be no return value, the API returns a reference to itself allowing expressions, such as the one shown in listing 6.2.

[4] http://www.oodesign.com/factory-pattern.html

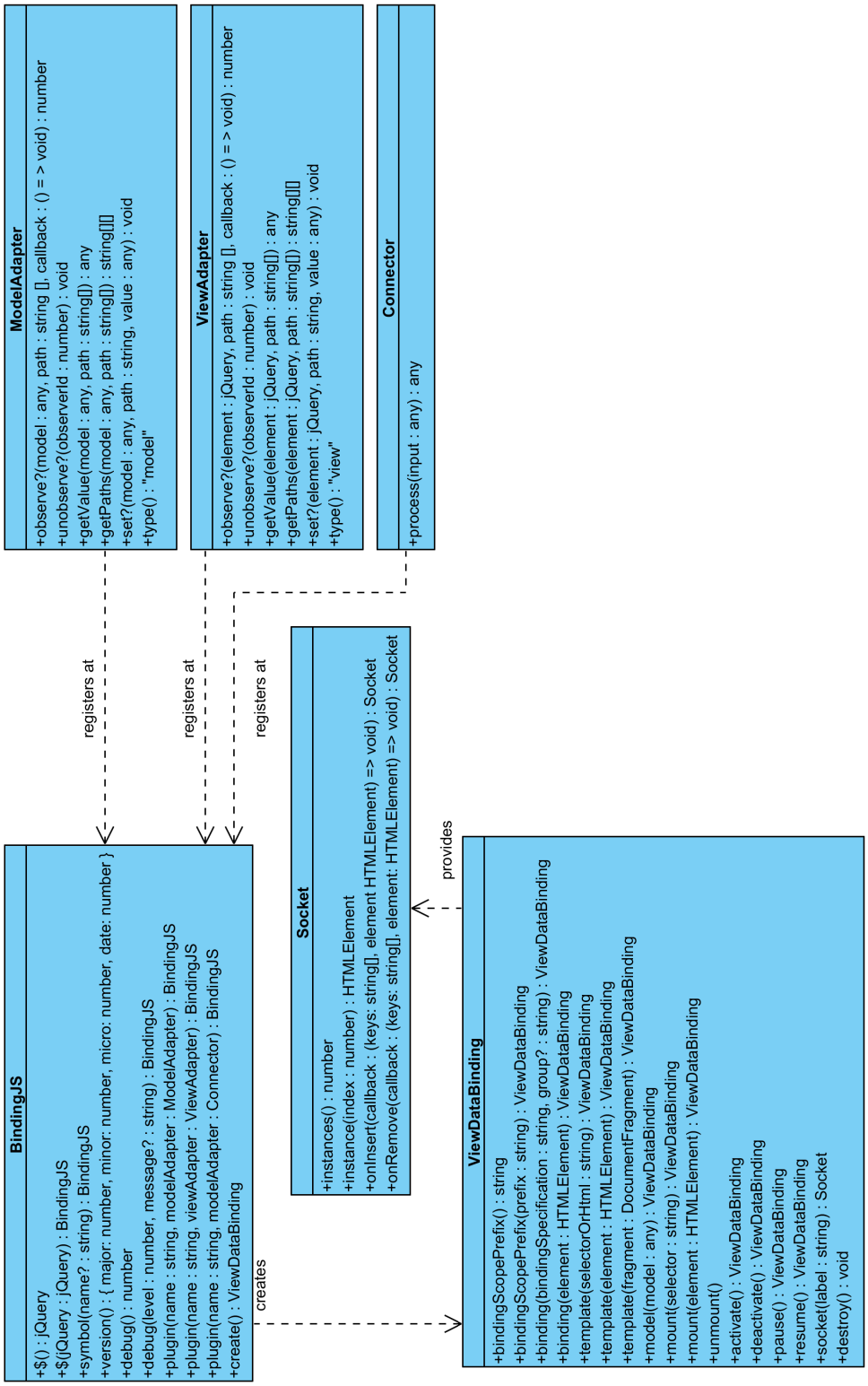


Figure 6.4.: BindingJS API

```
1 var vdb = BindingJS
2             .$(myJQuery) // Setting jQuery Reference
3             .debug(1)    // Setting Debug Level
4             .plugin("$", myModelAdapter)
5                          // Plugging In Presentation Model Adapter
6             .create()    // Creating Instance of View Data Binding
```

Listing 6.2: *BindingJS* API Example

6.2.2. Binding API

Once an instance of *View Data Binding* is created through `create()` it can be configured by the methods shown in the lower right box of figure 6.4.

bindingScopePrefix Every instance of *View Data Binding* provides a built-in instance of the *Binding Scope Adapter* that does not have to be plugged in explicitly. Through this method it is either possible to retrieve the current *Prefix* of that *Adapter* by providing no parameter, or to set it to a new value by passing a string. We decided that it is better to define this value locally to *View Data Binding* instead of globally in *BindingJS*, since it relates to the *Binding Specification*, which is also local to *View Data Binding*. It could be that not all *Binding Specifications* employed by a single web application use the same *Binding Scope Adapter Prefix*. All internal processing of the *Binding Specification* never assumes that a certain *Prefix* is used and always uses this method without a parameter to decide whether an *Adapter* refers to the *Binding Scope*. The default value returned by this method is `@`.

```
1  // With Group String
2  BindingJS.create()
3    .binding("@binding foo { " +
4               "@binding bar { " +
5                 "input { " +
6                   " value -> $value" +
7             "}}}", "foo.bar")
8
9  // Without Group String
10 BindingJS.create()
11   .binding("input { value -> $value }")
```

Listing 6.3: Setting the *Binding Specification* of *View Data Binding*

binding This polymorphic method allows to set the *Binding Specification* and may be only called once, since the *Binding Specification* is a static input and can not be changed after preprocessing. First, it is possible to pass the *Binding Specification* as a string which might be defined in-place, wrapped in a variable or pulled from an external file. An optional group string may be specified to realize *Identification* (see section 5.2.1). If no group is specified, the whole *Binding Specification* is used and otherwise only the given group is used for further processing while ignoring all other parts of the *Binding Specification*. The group string is expected

to identify a *Group* with a dot-notation. Listing 6.3 shows an example of two statements, which both set the same *Binding Specification*. Another possibility lies in passing an *HTMLElement* into the method which realizes declarative *View Data Binding*. Since this feature is not part of our *Core Concepts*, we only give an idea about how to implement it in chapter 8 and do not consider it any further within this chapter.

```
1  // 1. Selector
2  BindingJS.create().template("#template")
3
4  // 2. HTML
5  BindingJS.create().template("<div id='template'>...</div>")
6
7  // 3. HTML Element
8  var template = jQuery("#template").get()
9  BindingJS.create().template(template)
10
11 // 4. Document Fragment
12 var $dF = jQuery(document.createDocumentFragment())
13 $dF.append("<div id='template'>...</div>")
14 var dF = $dF.get()
15 BindingJS.create().template(dF)
```

Listing 6.4: Setting the *Template* of *View Data Binding*

template In the same way as with `binding` we want to provide the most flexibility in specifying the source of the *Template DOM Fragment*. It can be passed as a string representing a piece of HTML or a jQuery selector which is applied to the body of the current page. The selector must exactly match one element. Another option lies in providing a reference to an *HTMLElement* or a *Document Fragment*. The difference between the two is that they inherit from different interfaces and that *HTMLElements* are usually attached to the document while *Document Fragments* are detached and only exist virtually. Listing 6.4 shows all options that are there to set the *Template DOM Fragment*. This method may be called only once for the same reason as `binding`.

model With this method the *View Data Binding* instance is informed how to get a reference to the *Presentation Model*. It may be of any type and is never accessed by *View Data Binding* itself, but only through the registered *Presentation Model Adapter*. This method may be called only once because only the values inside the *Presentation Model* are dynamic.

mount After the three input parameters are given through `binding`, `template` and `model`, the *DOM Fragment* produced by *BindingJS* may be attached to the document with this method. Again, a jQuery selector, which must match exactly one element, can be used to reference an *HTMLElement*, or it may be passed directly into the method. In both cases, `mount` has a replace semantic. That means that the *HTMLElement* is replaced by the *DOM Fragment*, instead of the *DOM Fragment* replacing its children.

```
1  // 1. Selector
2  BindingJS.create()
3    .binding(...).template(...).model(...)
4    .mount("#template")
5
6  // 2. HTML Element
7  var template = jQuery("#template").get()
8  BindingJS.create()
9    .binding(...).template(...).model(...)
10   .mount(template)
```

Listing 6.5: Mounting the *DOM Fragment*

Listing 6.5 shows all possibilities which are there to mount the *DOM Fragment*. A typical use case, which is shown in line 2, is to mount the *DOM Fragment* at the same position where the *Template DOM Fragment* resides. This way the latter does not need to be removed manually and it seems as if the *Template DOM Fragment* was processed in-place. `Mount` may be called more than once, but the *DOM Fragment* never exists multiple times. That means that if `mount` is called twice, the content at the first *Mount Point* disappears and is moved to the second.

unmount Opposite of `mount`. Removes the *DOM Fragment* entirely.

activate This method puts the *View Data Binding* into action. It may called before or after `mount`. The *Bindings* from the *Binding Specification* are initialized and made active so that they start to synchronize the *Presentation Model* and the *DOM Fragment* (see section 6.3). A *View Data Binding* may only be activated if it was not previously activated.

deactivate Opposite of `activate` that puts the *View Data Binding* in a state as if it was never activated. It is possible to reactivate a *View Data Binding* which was deactivated.

pause A *View Data Binding* which is active may also be paused. The effect of it is that the *Bindings* of the *Binding Specification* no longer propagate if their *Source Adapter* change. These changes, however, are stored so that upon resuming the state of the *View Data Binding* is consistent again. Similar to `activate` a *View Data Binding* may only be paused if it was not previously paused.

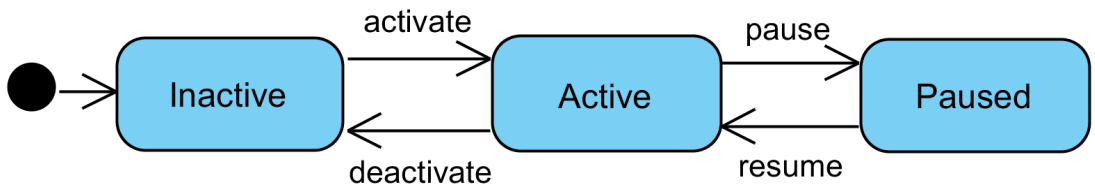


Figure 6.5.: State Diagram, View Data Binding

resume Opposite of `pause` that puts the *View Data Binding* in a state as if it was never paused. It is possible to pause a *View Data Binding* again after resuming it. Figure 6.5 shows a state diagram summarizing how a *View Data Binding* may change its state. The transitions of the diagram represent the methods which are allowed in each state and what their effect is.

```
1  var binding =
2    "@binding foo {" +
3      "@binding bar {" +
4        "#socket::socket" +
5      "}" +
6      "@binding baz {" +
7        "// ..." +
8    "}}" +
9
10 // Whole Binding
11 BindingJS.create()
12   .binding(binding).template(...).model(...)
13   .socket("foo.bar.socket")
14
15 // Part of Binding
16 BindingJS.create()
17   .binding(binding, "foo.bar").template(...).model(...)
18   .socket("bar.socket")
```

Listing 6.6: Accessing *Socket* API

socket This method provides access to the *Socket* API of a certain *Socket* referenced by its *Label* together with its surrounding *Group Identifiers*. It should be noted that the correct label depends on the part of the *Binding Specification* which was selected when calling `binding`. Listing 6.6 shows two examples how the *Socket* API may be retrieved.

destroy When the *View Data Binding* is no longer needed it should be destroyed to avoid unnecessary memory consumption. This method can be seen as the opposite of *BindingJS'* `create`. `Destroy` performs unmounting if it has not been done already.

Like *BindingJS, View Data Binding* provides a fluent interface so that method calls can be chained. We conclude this section with the example from listing 6.7. It shows how many of the methods can be used to configure and set up *View Data Binding* for a web page.

```
<html>
  <head>
    <script src="jquery.js"></script>
    <script src="binding.js"></script>
    <script type="text/binding">
      // Binding Specification
      @binding view {
        input {
          value <-> $name
        }
      }
    </script>
    <script type="text/javascript">
      // Model
      var model = {
        name: "Enter your name!",
      }

      // On Page Ready
      $(function () {
        BindingJS
          .plugin("$", jsonModelAdapter)
          .plugin("value", valueViewAdapter)
          .create()
          .template("#template")
          .binding($("script[type='text/binding']"))
          .model(model)
          .mount("#template")
          .activate()
      })
    </script>
  </head>
  <body>
    <!- Template DOM Fragment ->
    <div id="template">
      <input type="text" />
    </div>
  </body>
</html>
```

Listing 6.7: Typical Use of the *View Data Binding* API

6.2.3. Socket API

All methods of the *Socket* API refer to exactly one *Socket* of the initial *Binding Specification*. Their purpose is to get the correct *HTMLElements* inside the *DOM Fragment* that are produced from the *Template DOM Fragment* by the *View Data Binding*.

instances Counts the current number of instances of a *Socket*. The result may be greater or equal to zero. *Sockets* cannot be present at all when they are hidden or exist multiple times when repeated by *Iteration*.

instance This method can be used to iterate over all instances of a *Socket* by passing all numbers in the interval $[0; \texttt{instances()})$. The order in which the *HTMLElements* are returned remains unspecified. In particular, the order could be different from the order of appearances within the *DOM Fragment*. However, it is guaranteed that if the number of instances does not change while they are retrieved, each of them will appear exactly once.

onInsert Allows to register a callback that is notified after a *Socket* is inserted by an *Iteration*. Callbacks of *Sockets* which are in a *Scope* that has no iterated ancestor *Scope* are called on activating the *View Data Binding*. The callback receives the newly inserted *HTMLElement* and an array of keys. This key array represents the keys of the *Iterations* that lead to the creation of this *Socket* instance in top-down order. To better explain this listing 6.8 shows which arrays the callback receives. The array is required if the content that should be inserted into the *Socket* depends on the current *Entry* or *Key* of the *Iteration*. It is not necessary to give an array of *Entries*, since they can be retrieved from the *Presentation Model* by using the provided `keys`.

onRemove While `onInsert`'s callback is notified after the *HTMLElement* of a *Socket* was inserted into the *DOM Fragment*, with `onRemove` it is possible to be notified just before this *HTMLElement* is removed again. This can happen because of *Iteration* or when the *View Data Binding* is destroyed or unmounted. The callback receives an array of `keys` which contains the same information as the `keys` array of `onInsert`.

```
...
<script type="text/binding">
  // Binding Specification
  @binding view {
    .plainSocket::plainSocket
    .iterationOuter (@list: $data) {
      .iterationInner (@elem : @list) {
        .iteratedSocket::iteratedSocket
  }}}
</script>
<script type="text/javascript">
  // Model
  var model = { data: { foo: ["a", "b"], bar: ["c", "d"] } }

  $(function() {
    var binding = BindingJS.create()
                .template("#template")
                .binding($("script[type='text/binding']"))
                .model(model)

    binding.socket("view.plainSocket").onInsert(
      function (keys, element) {
        console.log("PlainSocket:  " + JSON.stringify(keys))
    })
    binding.socket("view.iteratedSocket").onInsert(
      function (keys, element) {
        console.log("IteratedSocket:  " + JSON.stringify(keys))
    })

    binding.mount("#template")
         .activate()

    // Prints on console
    // PlainSocket: []
    // IteratedSocket: ["foo", 0]
    // IteratedSocket: ["foo", 1]
    // IteratedSocket: ["bar", 0]
    // IteratedSocket: ["bar", 1]
  })
</script>
...
<!- Template DOM Fragment ->
<div id="template">
  <div class="plainSocket"></div>
  <div class="iterationOuter">
    <div class="iterationInner">
      <div class="iteratedSocket"></div>
</div></div></div>
...
```

Listing 6.8: Registering Callbacks for *Socket* Insertions

6.2.4. Plugin API

In section 6.2.1 we have shown how *Adapter* and *Connectors* can be plugged into *BindingJS* so that they are ready to be used in a *Binding Specification*. We now want to explain what their interfaces must be and provide example implementations. Both *Adapter* and *Connectors* are defined as a set of the methods shown on the top right of figure 6.4. Optional methods are indicated by a question mark.

Adapter

There are *Adapter* that cannot be observed, unobserved or set. For example, it usually does not make sense to observe the *View Adapter* `text`, since the text of an HTML element cannot be changed by the user.

The difference between *Model* and *View Adapter* is that they expect different associated elements for the methods `observe`, `getValue`, `getPaths` and `set`. The associated element of a *Model Adapter* always is the *Presentation Model* while it is an element from the *DOM Fragments* for *View Adapter*. Since the interfaces of the two are equal except for these parameter types, there is a method `type` that is used by *BindingJS* to determine which kind of *Adapter* is at hand.

observe Observes the associated element at the given *Path* and executes the callback whenever the value of the element identified by the *Path* changes. We expect that the implementation of an *Adapter* handles multiple observation of the same *Path* efficiently. This means that an *Adapter* should take care that the same associated element is not observed multiple times. The return value of the method is a number that can later be used to call `unobserve`.

unobserve Removes the observer previously registered which leads to the callback not being notified anymore. We decided that the *Adapter* interface is easier to use if observers are identified by numbers instead of the parameters which created the observer.

getValue Returns the value of the element referenced by the given *Path*. The return type will usually be primitive, but could be of any type.

getPaths Generates a list of *Paths* that are reachable from the given *Path* including the initial *Path*. The return type is an array of *Paths* where each *Path* is represented by an array of strings.

set Sets the value of the element at the given *Path* to a new value.

type Must either return the string `model` or `view` indicating the type of the *Adapter*.

To better understand how an *Adapter* can be implemented we want to demonstrate two examples. First, a *Model Adapter* for a JSON *Presentation Model* in listing A.3. Second, a *View Adapter* that reads and writes the value of HTML elements in listing A.5. These listings can be found in the appendix.

Both *Adapter* do not directly observe the *Presentation Model* or the `element` of the *DOM Fragment* when `observe` is called, but add a new observer to an internal list. The callback `notify` then calls all observers from that list so that the same *Path* is never observed more than once. To observe a JSON object, the *Presentation Model*

Adapter in this case uses *WatchJS*[5]. Observing *DOM Fragment* elements can be easily done with the help of jQuery.

Listing A.3 also shows how the method `getValue` can be implemented. It recursively navigates through the `model` and adds all values to an accumulator which builds the overall result. The `vaue` *Adapter* from listing A.5, on the other hand, has not to deal with *Paths*, since it is not meant to be used with a *Qualifier* or as part of a *Dereferencing Expression*. Hence, the methods `getValue` and `set` are easier to implement because in this simplified example they can just ignore the `path` parameter. Since there are no *Paths* to return except the input `path` required by the specification of the *Adapter* API, `getPaths` exactly returns a list comprising that single input `path`.

By considering the examples it should be clear that there is a great flexibility in implementing both various *Presentation Model* and *View Adapter*. For the latter, apart from the `value` *Adapter*, we consider the following set of *View Adapter* as essential.

text Manipulates the content of HTML tags, such as `span`, `div` or `p` tags.

attr Modifies attributes of HTML tags, such as the `id`, `enabled` or `checked` attributes.

css Changes the appearance of an HTML element by setting the value of a CSS property defined by its *Qualifier*.

class Adds and removes class identifiers inside the string of the `class` attribute of an HTML tag. Typically, this attribute represents a whitespace separated set of classes. Therefore, this *Adapter* uses its *Qualifier* as the information, which class to add or remove. If the *Adapter* is then written with `true`, the class is added; if it is written with `false`, the class is removed.

on Reads the event whose type is defined by its *Qualifier* and that is fired by an HTML tag. It is mainly used as an *Initiator*, but also allows to retrieve the event instance when denoted as the *Source Adapter* of a *Binding*.

There are two special ways of implementing an *Adapter* so that it can either be used as a function or as a callback. If the *Presentation Model Adapter* is extended so that it recognizes if the element that is written by `set` is a function or not, an *Adapter* can be used to call a function inside the *Presentation Model* with the values that were passed to `set`. An example of this is shown in listing 6.9.

Similarly by modifying the implementation of `getValue` an *Adapter* can behave like a function. Instead of returning the value from a *Data Target* it could execute a function in the *Presentation Model* and return the result of that function as its value. An example of this is shown in listing 6.10.

[5] https://github.com/melanke/Watch.JS/

```
// Binding Specification
button {
  on:click -> $performClickLogic
}

// Adapter Implementation
function set(model, path, value) {
  var element = resolvePath(model, path)
  if (isFunction(element)) {
    element(value) // call function
  } else {
    element = value // set new value
  }
}

// Presentation Model
var model = {
  performClickLogic: function (event) {
    // Perform Click Logic
  }
}
```

Listing 6.9: Using *Adapter* to Execute Callbacks

```
// Binding Specification
li (@number: $genArray) { ... }

// Adapter Implementation
function getValue(model, path) {
  var element = resolvePath(model, path)
  if (isFunction(element)) {
    return element() // result of function
  } else {
    return element // value of element
  }
}

// Presentation Model
var model = {
  genArray: function () {
    var result = []
    for (i in 1..100) {
      result.add(i)
    }
  }
}
```

Listing 6.10: Using *Adapter* as a Function

Connector

The interface for a *Connector* is quickly explained, since it only contains one method whose input may be a primitive value, a *Reference* or any structured JSON that contains primitive values and *References*. A *Reference* is a wrapper for an *Adapter* and is provided and generated by *BindingJS* when reading from *Adapter* (see section 5.1.2 and 6.3.6).

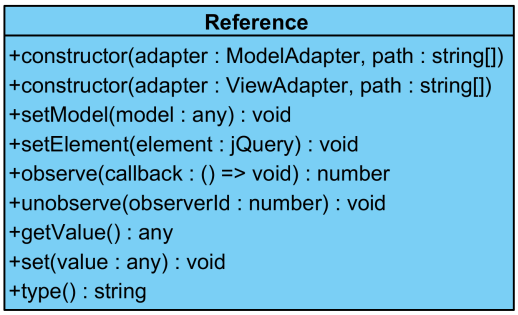


Figure 6.6.: Reference, Class Diagram

```
1 BindingJS.plugin("revert", {
2   process: function (input) {
3     if (isReference(input))
4       input = input.getValue()
5     return (input + "")
6               .split("")
7               .reverse()
8               .join("")
9 }})
```

Listing 6.11: *Connector* `revert`

The interface of a *Reference* is shown in figure 6.6. Since it is a wrapper for an *Adapter* together with a *Path*, its methods are very similar to those of an *Adapter*. *Connectors* may either keep *References* or replace them by their underlying or arbitrary values. However, they are not allowed to create new *References*. This way all types of *Connectors* that were described in section 5.1.2.2 can be implemented. We want to demonstrate this with two examples, a *Connector* that reverts a string in listing 6.11, and one that sorts a list of strings in listing 6.12.

```
1  BindingJS.plugin("sort", {
2    process: function (input) {
3      var values = []
4      foreach (item in input)
5        values.add(item)
6      values.sort(function /* compare */ (a, b) {
7        var aValue = isReference(a) ? a.getValue() : a
8        var bValue = isReference(b) ? b.getValue() : b
9        return aValue < bValue
10     })
11     return values
12 }})
```

Listing 6.12: *Connector* Sorting a List of Strings

`revert` expects a primitive value or a *Reference*. If it receives a *Reference*, it first converts it to its underlying value and thereby destroys it. It always returns a primitive value. `sort`, on the other hand, expects a list on which some items may be primitive values and some *References*. It uses the underlying values of the *References* only to compare them, but does not destroy them. This means that they still exist in the result `values`.

6.3. Algorithms

Now that we defined how it is possible to interact with *BindingJS* through its API, we turn back to figure 6.3 from section 6.1 and explain what happens inside the components when the API is called. We will explain each algorithm by providing examples and showing what its challenges are. For reference, their actual implementations are shown in pseudo JavaScript code in the appendix.

6.3.1. Repositories

Both *Adapter* and *Connector Repository* are simple key value stores. The key is the name of the component that was defined when it was plugged in through the *BindingJS* API and the value is the component itself. Figure 6.7 shows their interfaces.

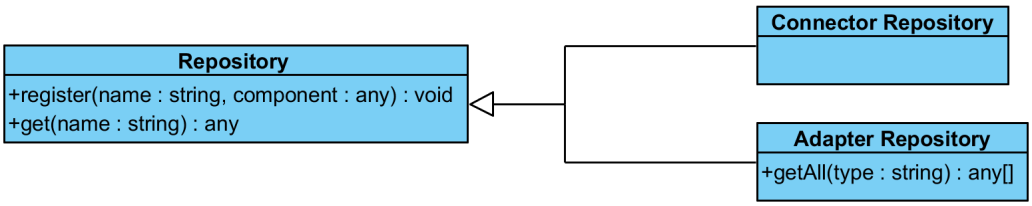


Figure 6.7.: Repositories, Class Diagram

The implementation of `plugin` which registers an *Adapter* or *Connector* at *BindingJS* is trivial. It does nothing but passing its parameter to the `register` method of the appropriate *Repository*. When the *Bindings* inside a *Binding Specification* are initialized by the *Propagator* (see section 6.3.6), `get` is used to resolve the names of *Adapter* or *Connectors* to their actual implementations.

6.3.2. Binding Scope

Not only the *Repositories*, but also the *Binding Scope* is a simple map data structure with the interface shown in figure 6.8. It is used to store all values and *References* that are written into the *Binding Scope Adapter* and represents the third *Data Target* alongside the *DOM Fragment* and *Presentation Model*.

get Returns the value that was previously stored through `set` at the given identifier.

set Sets the value of an identifier to a given value. All *References* that are in `to` are observed and all observers of the identifier will be notified when one of their underlying values changes. If the previous value at the given identifier contains *References* they are unobserved before being overwritten. When setting a new value to an `id` that was previously set, all observers of that `id` are notified if the new value is different from the old one.

observe Allows to be notified about any changes to the data referenced with `id`. The method returns a unique number which can be used to call `unobserve`.

unobserve Stops executing the callback of an observer that was previously registered through `observe`.

destroy Removes the value and an id from the *Binding Scope* as if it were never there, and also stopping the observation of probable *References*.

It is necessary to observe *References* that are written into the *Binding Scope* because it stores objects that are or contain *References*. Since *References* return their new underlying value as soon as it is changed, using these objects for comparison is impossible. The correct execution of *Propagation*, however, depends on being able to compare old and new values to decide if a *Binding* must be propagated. A solution would be to store the underlying values of *References* and use them for comparison. This, however, would make the implementation unnecessarily complex.

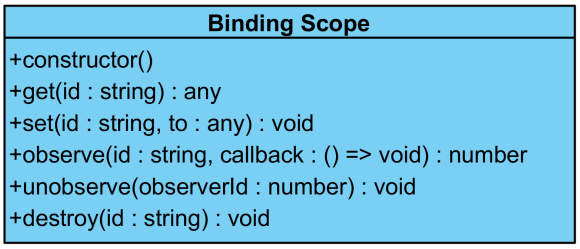


Figure 6.8.: Binding Scope, Class Diagram

6.3.3. Parser

When the *Binding Specification* is set for an instance of *View Data Binding* through `binding` it is first handed to the *Parser* which converts the input string into an abstract syntax tree (AST). The *Parser* is implemented with the help of *PEG.js*[6], a parser generator for JavaScript. With this tool we can use the Parsing Expression Grammars from chapter 5 as the input for the generator that is created by *PEG.js* when transforming this grammar (See Appendix A.6) into JavaScript code. It is able to process any string starting from any given symbol. In addition, the library offers helpful error messages so that in case of syntax errors, *BindingJS* can clearly state where an error occurred and which characters it expected instead of the ones found.

In listing 6.13 a *Binding Specification* is shown which produces the AST from listing 6.14. The latter visualizes the tree by first showing the type of each node, followed by its attributes. The last information in each line is the position where the element was parsed and should be read as `[line/index]`.

Once parsing the string received through the method `binding` was successful, it is discarded and only the AST is saved for further processing. In the case that the optional group string was used with the method the correct group is searched in the AST and only that part of it is stored.

The circumstance that many *Binding Specifications* can be merged into one file forms our primary use case for *Identification*. This, however, would mean that the same potentially big *Binding Specification* would have to be parsed over and over again just to extract parts of it while discarding all other *Groups*. To solve this problem we plan on adding a separate parser that only parses *Groups* before fully parsing the relevant *Group* in the future.

[6] http://pegjs.majda.cz/

```
@binding foo {
  bar (@entry, @key: $collection) {
    baz <- @entry
}}
```

Listing 6.13: Example *Binding Specification* as Input for the Parser

```
Blocks [1/1]
  Group (id: "foo") [1/1]
    Blocks [1/15]
      Scope [2/3]
        SelectorList [2/3]
          SelectorCombination [2/3]
            Selector [2/3]
              Element (name: "bar") [2/3]
        Iterator [2/7]
          Variables [2/7]
            Variable (ns: "@", id: "entry") [2/8]
            Variable (ns: "@", id: "key") [2/16]
          Expr [2/7]
            Variable (ns: "$", id: "collection") [2/22]
        Binding [3/5]
          Adapter [3/5]
            ExprSeq [3/5]
              Variable (ns: "", id: "baz") [3/5]
          Connector [3/9]
            BindingOperator (value: "<-") [3/9]
          Adapter [3/12]
            ExprSeq [3/12]
              Variable (ns: "@", id: "entry") [3/12]
```

Listing 6.14: Example Abstract Syntax Tree as Output of the Parser

6.3.4. Preprocessor

As soon as all three input components, *Template DOM Fragment*, *Binding Specification* and *Presentation Model*, are present, *BindingJS* initiates the preprocessing. The *Presentation Model* is stored as a reference inside the *View Data Binding* instance, since it is a component that is both read and written. The *Template DOM Fragment*, on the other hand, is cloned when set through `template` so that preprocessing and all further algorithms do not operate on the *Template DOM Fragment* that is attached but on an internal copy of it. The preprocessing performs a mix of transformation and validation operations and is divided into eleven steps. Its purpose is to recognize and eliminate erroneous *Binding Specifications* upfront and to convert all static input so that it can be easier processed by the *Engine*. This mainly includes flattening the hierarchy of *Scopes* so that it is only reflected in the resulting *Iteration Trees*, but neither in the *Binding Scope* nor the *Selectors* of *Scopes*. Figure 6.9 shows an overview of all preprocessing operations which we explain in detail in this section.

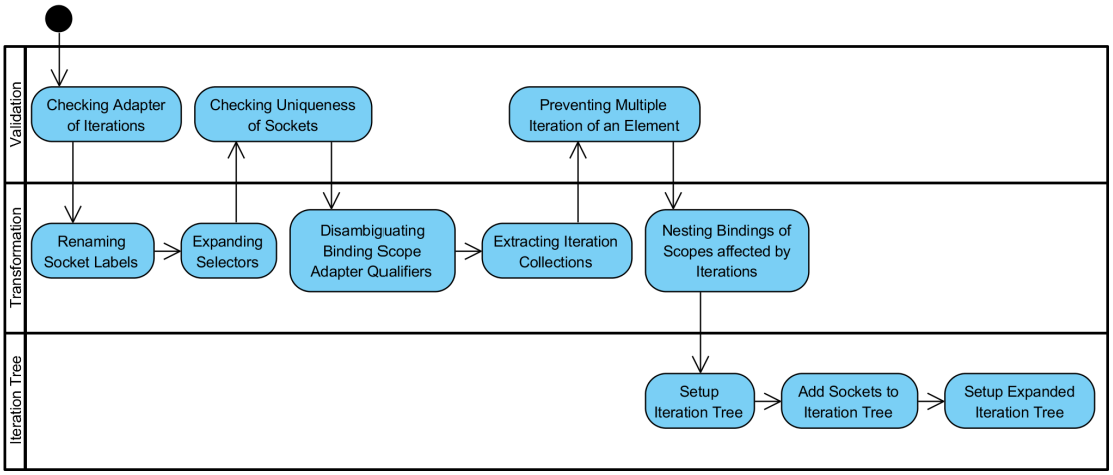


Figure 6.9.: Preprocessor, Overview

1. Checking *Adapter* of *Iterations*

The first preprocessing step checks if every *Iteration* uses only the *Binding Scope Adapter* as their *Entry* and *Key Adapter* (See Appendix A.7). In addition, it tests whether their *Qualifiers* where used earlier. Both of this is disallowed according to section 5.1.3. The recursive algorithm works top-down on the abstract syntax tree collecting all *Qualifiers* of *Binding Scope Adapter*. For every occuring *Iterator* its *Entry* and *Key Adapter* are checked against the aforementioned conditions.

Listing 6.15 shows a *Binding Specification* (See Appendix A.9 for the corresponding AST) that fails step 1. When the algorithm is executed the first *Scope* that occurs is the one with *Selector* `div`. Since `@temp` is the only *Binding Scope Adapter* that occurs directly within this *Scope*, it is remembered. When the recursion reaches the *Scope* from line 8 with Selector `li`, an exception is thrown because `@temp` is used as the *Key Adapter* of an *Iteration*. Although `@entry` is used in both, the `span` and `li` *Scope*, it

```
@binding counter {
  div {
    @temp <- $temp
    span {
      @entry <- $spanText
      text <- @entry
    }
    li (@entry, @temp: $collection) {
      text <- @entry + " - " + @temp
}}}
```

Listing 6.15: Step 1: Counter Example

is not the reason why the algorithm fails. According to section 5.1.2.3, in this case `@entry` refers to different items of the *Binding Scope*.

2. Renaming *Socket Labels*

Sockets are identified by their enclosing *Group Identifiers* and their *Label* combined in a dot-notation. To make access to certain *Sockets* more efficient their *Labels* are renamed in step 2 so that they already include their enclosing *Group Identifiers*. This is done by recursively prepending *Group Identifiers* to *Labels* in a bottom-up fashion. An example of a successful transformation is provided by listings 6.16 and 6.17.

```
@binding foo {
  @binding bar {
    #socketA::sA
  }
  @binding baz {
    #socketB::sB
  }
}
```

Listing 6.16: Before Step 2

```
@binding foo {
  @binding bar {
    #socketA::foo.bar.sA
  }
  @binding baz {
    #socketB::foo.baz.sB
  }
}
```

Listing 6.17: After Step 2

3. Expanding Selectors

In step 3 the *Selectors* of *Scopes* are evaluated against the *Template DOM Fragment* (See Appendix A.8). In a top-down fashion, each *Scope* is recursively manipulated so that its *Selectors* are removed and replaced by their matched elements. The goal of this step is that every *Scope* is assigned a reference to an element of the internal *Template DOM Fragment*. During the process *Scopes* may be multiplied if their *Selector* matches multiple elements or if a list of *Selectors* is used. Also, *Scopes* may disappear if their *Selector* does not match any elements. This is done to ensure that every *View Adapter* of a *Binding* inside a *Scope* has to only consider a single element.

```
1 @binding view {
2   div {
3     attr:data-info <- "div"
4     span {
5       attr:data-info <-
6         "span in a div"
7 }}}
```

Listing 6.18: *Binding Specification*

```
1 <div id="template">
2   <div id="divA"></div>
3   <span id="spanA"></span>
4   <div id="divB">
5     <span id="spanB"></span>
6   </div>
7 </div>
```

Listing 6.19: *Template DOM Fragment*

Listings 6.18 and 6.19 serve as an example to explain what the algorithm does. There are two *Scopes*, one with `div` and the other with `span` as its *Selector*. Since the algorithm is top-down, the `div` *Selector* from line 2 is first evaluated. It only matches elements which are descendants of the current context, which is the whole *Template DOM Fragment* in the beginning. Therefore, the `div`s from lines 2 and 4 of listing 6.19 are matched. This leads to the *Scope* being cloned twice. This intermediate step is shown in listing 6.20. Although the *Scopes* from lines 2 and 8 do not have a *Selector* anymore, we visualize their matched elements by the unique *Selectors* `#divA` and `#divB`.

```
1  @binding view {
2    #divA {
3      attr:data-info <- "div"
4      span {
5        attr:data-info <-
6          "span in a div"
7    }}
8    #divB {
9      attr:data-info <- "div"
10     span {
11       attr:data-info <-
12         "span in a div"
13 }}}
```

Listing 6.20: *Binding Spec.* Intermed.

```
1  @binding view {
2    #divA {
3      attr:data-info <- "div"
4    }
5
6    #divB {
7      attr:data-info <- "div"
8      #spanB {
9        attr:data-info <-
10         "span in a div"
11     }
12   }
13 }
```

Listing 6.21: *Binding Spec.* Final

For both of the new *Scopes* their children are recursively processed now having the element matched by their parent as their context instead of the whole *Template DOM Fragment*. As a consequence, the `span` from line 4 does not match any elements and the *Scope* is removed. The second `span` from line 10, however, matches the `span` element, which is inside `#divB` in listing 6.19. The final result of the third preprocessing step – again visualized as a *Binding Specification* – can be seen in listing 6.21.

4. Checking Uniqueness of *Sockets*

A *Sockets* may never refer to an element of the *Template DOM Fragment* which was matched by another *Scope*. Verifying this can be easily done after step 3 by just counting how many times any element appears as the element of any *Scope*. If for any *Scope* that defines a *Socket* this count is greater than one, the validation fails and an exception is thrown. It should be noted that it does not make sense to execute this validation step earlier, since it would have to evaluate the *Selectors* of *Scopes* manually. Counter-examples that would not pass this step include a *Socket* that is defined for a *Scope* whose *Selector* matches more than one element, or a *Binding Specification* where the same *Label* appears multiple times within the same *Group*.

5. Disambiguating *Binding Scope Adapter Qualifiers*

In section 5.1.2.3 we have shown that the *Binding Scope* is hierarchical. In particular, this means that in some cases *Binding Scope Adapter* that use the same *Qualifier* refer to the same or different elements of the *Binding Scope*. Renaming those *Qualifiers* in such a way that they never refer to different elements, allows us to implement the *Binding Scope* as a flat key value store that does not need to know about the hierarchy of *Scopes*. The algorithm that performs this renaming was shown earlier in listing 5.17 and section 5.1.2. All we want to amend at this point is that the *Qualifiers* of *Entry* and *Key Adapter* of *Iterations* have to be included. The listings 6.22 and 6.23 show a *Binding Specification* before and after being transformed by step 5.

```
1  @binding view {
2    @name <- $name
3    span {
4     @text <- @name
5     text <- @text
6    }
7    div {
8      @text <- @name
9      text <- @text
10 }}
```

Listing 6.22: *Binding Spec.* Before

```
1  @binding view {
2    @temp1 <- $name
3    span {
4     @temp2 <- @temp1
5     text <- @temp2
6    }
7    div {
8      @temp3 <- @temp1
9      text <- @temp3
10 }}
```

Listing 6.23: *Binding Spec.* After

6. Extracting *Iteration* Collections

As shown in section 5.1.3, the *Source Adapter* of an *Iteration* may be of any type. With the introduction of *Expressions* in section 5.3.6, complex statements comprising multiple different *Adapter* can be used, too. Implementing *Iteration* becomes easier if the *Source Adapter* of any *Iteration* always is a *Binding Scope Adapter*. This way, *Iteration* never depends on the *Presentation Model* or *Document Fragment*, but needs to only observe the *Binding Scope*.

The algorithm iteratively manipulates each iterated *Scope*. If it has a parent *Scope*, a new *Binding* is added to that *Scope*. If no such parent *Scope* exists, it is first checked

if the *Source Adapter* or *Expression* of the *Iteration* contains any *View Adapter*. According to section 5.1.3, this is not allowed and an exception is thrown. If no *View Adapter* are present, a new *Scope* is added after the iterated *Scope* comprising the new *Binding* and having the parent of the element from the iterated *Scope* as its own element. Since the new *Binding* can not contain any *View Adapter*, this element has no semantic meaning. For further preprocessing steps, however, an element is required and choosing the parent element causes no extra efforts in dealing with this special case.

```
1  @binding view {
2    #li (@entry, @key:
3         $collection) {
4      text <- @key + ". " + @entry
5    }
6
7    #div {
8      #span (@name: $names) {
9        text <- @name
10     }
11   }
12 }
```

Listing 6.24: *Binding Spec.* Before

```
1  @binding view {
2    #li (@entry, @key: @temp1) {
3      text <- @key + ". " + @entry
4    }
5    #template {
6      @temp1 <- $collection
7    }
8    #div {
9      @temp2 <- $names
10     #span (@name: @temp2) {
11       text <- @name
12 }}}
```

Listing 6.25: *Binding Spec.* After

Listing 6.24 shows a *Binding Specification* before, and listing 6.25 after the sixth preprocessing step was executed. It should be noted that although the references to `@temp1` in both lines 2 and 6 would usually refer to different elements of the *Binding Scope*, this is no longer the case after preprocessing step 5.

7. Preventing Multiple *Iteration* of an Element

This preprocessing step realizes the requirement from section 5.1.3 that one element may not be iterated multiple times, since this would put semantics to the order of *Scopes*. Every *Iteration* is iterated and its element stored. If one element occurs twice, an exception is thrown. Listing 6.26 shows an example of a *Binding Specification* which would fail this validation step if any `li` elements are present in the *Template DOM Fragment*.

```
1  @binding view {
2    li (@entry, @key: $collection) {
3      // ...
4    }
5    li ($condition) {
6      // ...
7  }}
```

Listing 6.26: Step 7: Counter Example

8. Nesting *Bindings* of *Scopes* affected by *Iterations*

In step 9 both the *Binding Specification* and *Template DOM Fragment* are dismembered to reflect *Iterations*. Therefore, any *Scope* that refers to an element that is iterated must be a descendant of this *Iteration*. Listing 6.27 shows a simple *Binding Specification*, where this is not the case. Assuming that the *Template DOM Fragment* contains exactly one `div` element with id `iteratedDiv`, both *Scopes* refer to that `div`.

```
1 @binding view {
2   div {
3     attr:data-info <- "div"
4   }
5   #iteratedDiv (@entry, @key:
6                 $collection) {
7     text <- @key + ". " + @entry
8   }
9 }
```

Listing 6.27: *Binding Spec.* Before

```
1 @binding view {
2   #iteratedDiv (@entry, @key:
3                 $collection) {
4     text <- @key + ". " + @entry
5     div#iteratedDiv {
6       attr:data-info <- "div"
7     }
8   }
9 }
```

Listing 6.28: *Binding Spec.* After

All relevant *Scopes* are moved so that they are correctly nested (See Appendix A.4). This can be done regardless of the hierarchy of *Scopes*, since it has no semantics after steps 3 and 5. The process could lead to *Scopes* remaining empty if their only child was a *Scope* that has been moved. Although they would not cause harm these empty *Scopes* are removed. Listing 6.28 shows the *Binding Specification* after being transformed by step 8.

9. Setup *Iteration* Tree

Similar to the hierarchy of *Scopes*, *Iterations* build a tree, too. Inside a *Binding Specification* there may be an arbitrary number of *Iterations* which again can have child *Iterations*. We treat the whole *Binding Specification* as an *Iteration* which is always present exactly once. A node of this *Iteration Tree* comprises the information depicted in figure 6.10. Since we will later add an additional tree of *Expanded Iterations*, we call a node inside this tree a *Plain Iteration*.

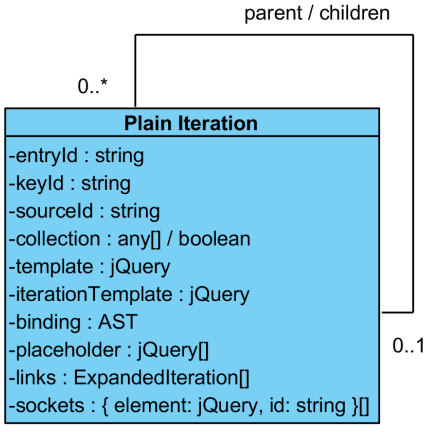


Figure 6.10.: Plain Iteration, Class Diagram

Each *Plain Iteration Node* stores the *Qualifiers* of its *Entry*, *Key* and *Source Adapter* as a string. It also stores the last value it received from its *Source Adapter* which might be a collection or a Boolean value. While the `template` attribute stores the part of the *Template DOM Fragment* that is visible due to this *Iteration*, the part of the *Template DOM Fragment* is stored in the `iterationTemplate` which is added for new instances of the *Iteration*. In addition, the part of the *Binding Specification*, which is covered by this *Iteration*, is stored in `binding`.

The elements of *Scopes* in `binding` always refer to elements of `iterationTemplate`. A `placeholder` array comprises information where child nodes would add their `iterationTemplate` inside the node's own `iterationTemplate`. It should be noted that such placeholders are not required for `binding`, since the semantics of its hierarchy are no longer relevant after steps 3 and 5. At the moment, we do not consider the attribute `links`. It will become relevant in step 11. Listings 6.29 and 6.30 show a *Binding Specification* and a *Template DOM Fragment* which we are using to explain the attributes in more detail.

```
1  @binding view {
2    #label {
3      text <- $labelText
4    }
5    #divA {
6      @temp1 <- $items
7      #liA (@item:  @temp1) {
8        text <- @item
9  }}
10   #divPeople {
11     @temp2 <- $countries
12     #divCountry (@country:  @temp2) {
13       @temp3 <- @country.names
14       #divNames (@name:  @temp3) {
15         text <- @name
16       }
17       @temp4 <- hasAttribute(@country.flag)
18       #flagImg (@temp4) {
19         attr:src <- @country.flag
20 }}}}
```

Listing 6.29: *Binding Specification*

All previous preprocessing steps need to be taken into consideration so that all *Iterations* already have a *Binding Scope Adapter* as their *Source Adapter* and every *Scope* refers to exactly one element of the *Template DOM Fragment*. We did not rename all *Binding Scope Adapter*, as done by step 5, to keep the example readable. There are multiple and nested *Iterations* in the example. `$items` which is an array of strings is iterated and bound to an `li` element. In addition, `$countries` and the names of the people living in each of the countries are iterated. For each country a picture of its flag is shown in an `img` element if the `flag` attribute of the country exists. The *Plain Iteration Tree* for this *Binding Specification* together with its attributes is shown in figure 6.11. The algorithm (see Appendix A.11) converting the abstract syntax tree and the *Template DOM Fragment* into a *Plain Iteration Tree* works recursively in a top-down fashion on the *Iterations* inside the *Binding Specification*.

```
<div id="template">
  <span id="label"></span>
  <div id="divA">
    <li id="liA"></li>
  </div>
  <div id="divPeople">
    <div id="divCountry">
      <div id="divNames"></div>
      <img id="flagImg"></img>
    </div>
  </div>
</div>
```

Listing 6.30: *Template DOM Fragment*

10. Add *Sockets* to *Iteration* Tree

Once the *Plain Iteration Tree* is built, an additional attribute `sockets` is added to each node of the tree. We decided to factor out this step to keep step 9 comprehensible, even though it would be possible to combine both algorithms into one. The purpose of step 10 is to move all information about *Sockets* from the abstract syntax tree into the *Plain Iteration Nodes*. Therefore, `sockets` is an array of items where each item comprises the two attributes `element` and `id`. The `element` is a reference to an element of the `iterationTemplate` of the node and `id` stands for the *Label* of the *Socket*. With this array it becomes easier to implement the *Socket* API (see section 6.3.7).

11. Setup Expanded *Iteration* Tree

Our last preprocessing step consists of preparing an *Expanded Iteration Tree* for the *Iterator* component that will be described in the next section. The *Plain Iteration Tree* reflects *Iterations* as they appear within the *Binding Specification* or the abstract syntax tree. The *Expanded Iteration Tree*, on the other hand, reflects the presence of *Iterations* given their current `collections`. We will explain this in more detail later but want to give an intuitive understanding with an example at this point. Aganin, consider the *Plain Iteration Tree* from figure 6.10. If all attributes are left out, it can be illustrated easier as the orange tree on the left of figure 6.13.

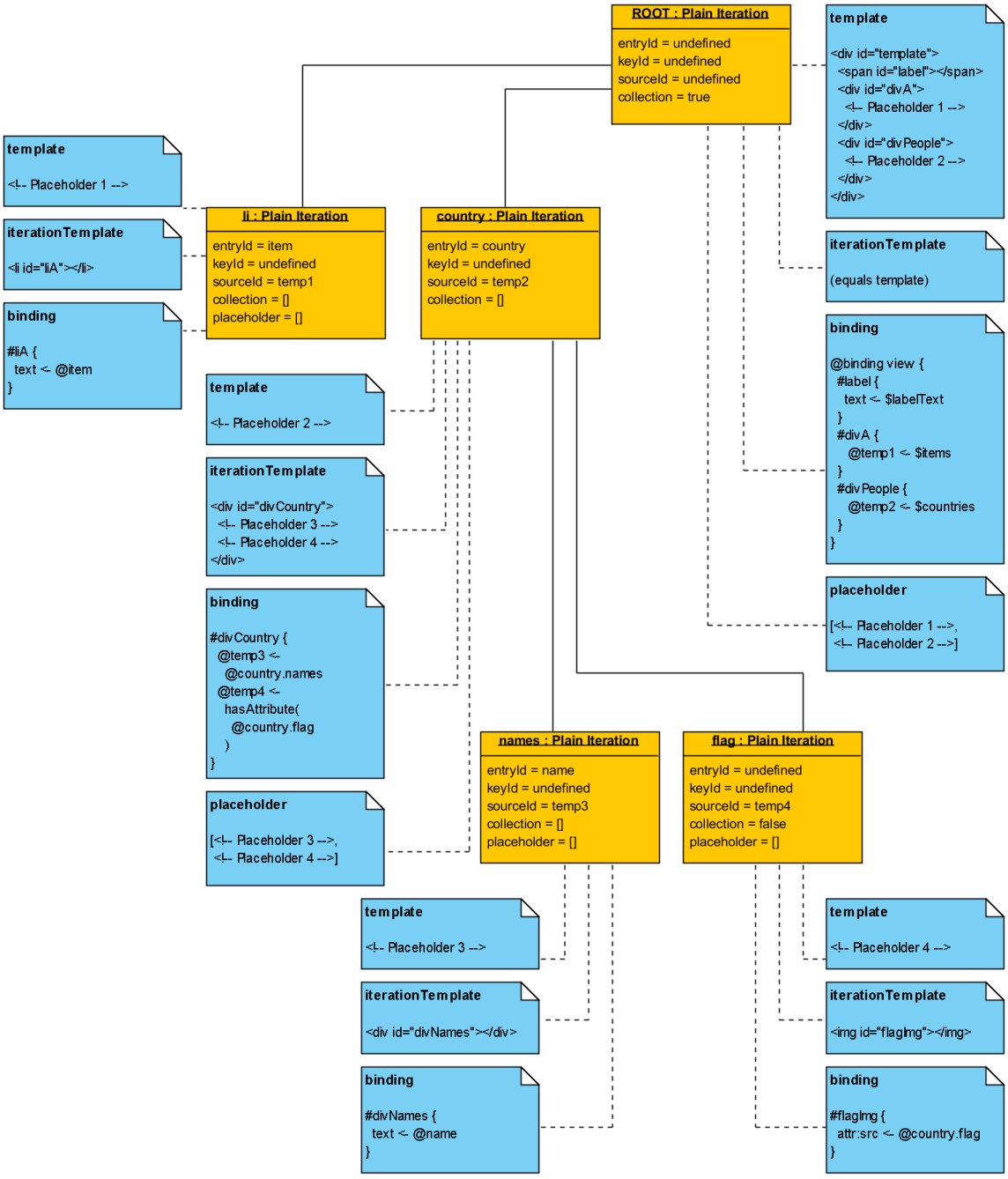


Figure 6.11.: Plain Iteration Tree, Object Diagram

If we now assume that there are two countries that are iterated, both the `names` and `flag` *Iteration* are actually there twice. What appears twice has nothing to do with the items that are iterated, but only how many times the *Iteration* itself is present. That means that if there are two countries, there need to be two *Iterations* listing all names and deciding whether the flag should be shown. These new set of *Iterations* that we call *Expanded Iterations* builds a new tree. It is linked to the *Plain Iteration Tree*, shown on the right of figure 6.13 colored in turquoise. The dashed lines represent the `links` attribute of every *Plain Iteration* node. In this case, every node has exactly one link, with the exception of `names` and `flag` which have two. The attributes that an *Expanded Iteration* node stores are shown in figure 6.12. Before explaining the attributes in more detail, we need to leap ahead to the next section where the idea of *Iteration Instances* is explained in detail. An *Iteration Instance* is an element produced for each item in the collection retrieved from the *Source Adapter* of the *Iteration*. In our previous example the `country` *Iteration* had two instances that lead to the creation of two `names` and `flag` *Expanded Iterations*.

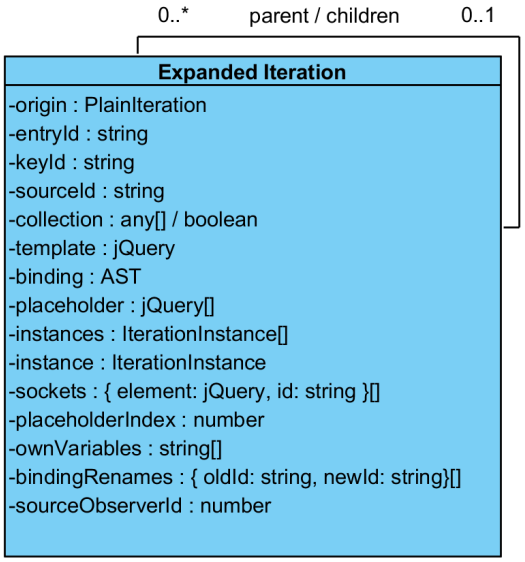


Figure 6.12.: Expanded Iteration, Class Diagram

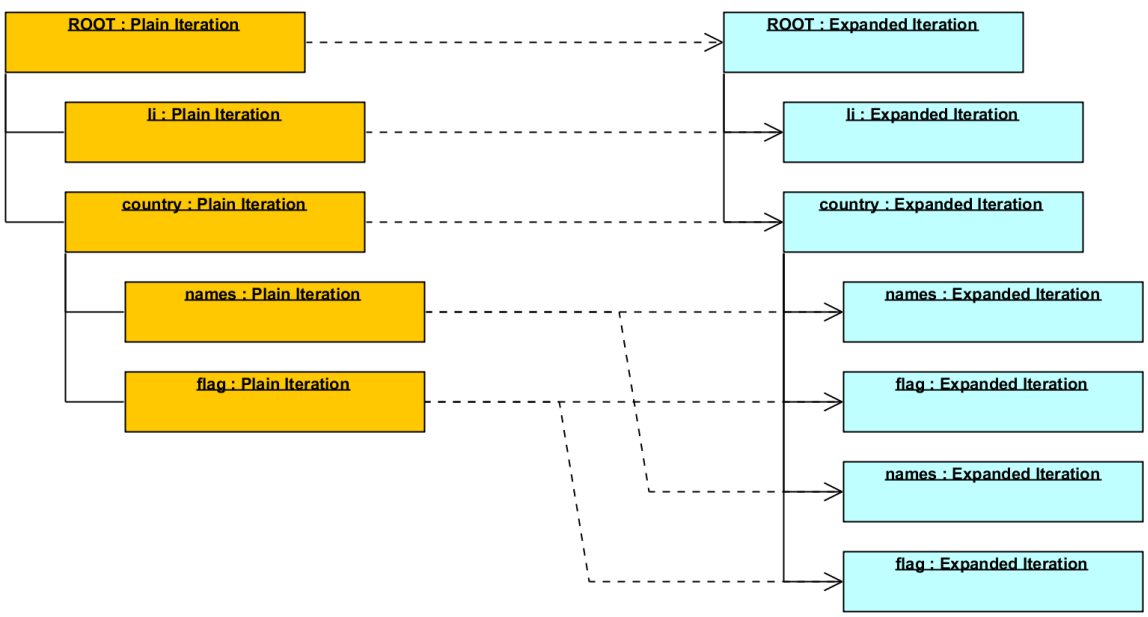


Figure 6.13.: Plain together with Expanded Iteration Tree

origin
Counterpart to the `links` from *Plain Iteration*. This reference is used to navigate effortlessly to the originating *Plain Iteration* instance.

entryId
Qualifier of the *Entry Adapter* of this *Iteration*. This might be different from the `entryId` of the *Plain Iteration* in `origin`. Since we disambiguate *Binding Scope Adapter Qualifiers*, multiple *Expanded Iterations* that were created from the same *Plain Iteration* need to use different *Entry Adapter Qualifiers*.

keyId
Qualifier of the *Key Adapter* of this *Iteration*. The reason why this information has to be stored is the same as for `entryId`.

sourceId
Qualifier of the *Source Adapter* of this *Iteration*. The reason, why this information has to be stored is the same as for `entryId`.

collection
Similar to *Plain Iteration*, the last value retrieved from the *Source Adapter* is stored here. Again, this value might be different from the one in the *Plain Iteration* because the collection might be retrieved from a parent's *Entry Adapter*. In the previous example we would expect that the source collections of both `names` *Iterations* are different.

template
Clone of `origin`'s `template`. The original template needs to be cloned to be able to provide a unique set of placeholders.

binding
Clone of `origin`'s `binding` with renaming performed.

placeholder
Clone of `origin`'s `placeholder` array, but referring to the `template` of this *Expanded Iteration*.

instances
The set of *Iteration Instances* for this *Expanded Iteration*

instance
Reference to the *Iteration Instance* inside the `instances` set of the parent of this *Expanded Iteration*. This information is necessary to decide which *Iteration Instance* was responsible for the creation of this *Expanded Iteration*.

sockets
Copy of `origin`'s sockets, but elements referring to the `template` of this *Expanded Iteration*.

placeholderIndex
In the *Plain Iteration Tree* any child can determine its placeholder by choosing the one with the same index as itself in the array of children. This is not possible in the *Expanded Iteration Tree*, since there might be more children than elements in parent's `placeholder` array.

ownVariables
Qualifiers of all *Binding Scope Adapter* that occurred within the `binding` of this *Expanded Iteration* and that have never occured in any ancestor of this *Expanded Iteration*.

bindingRenames
All renamings of *Binding Scope Adapter Qualifiers* performed by this *Expanded Iteration*. If new *Expanded Iterations* are created, among other things, they need to rename their *Entry* and *Key Adapters*. If those are used again in any child, they need to know about this renaming.

sourceObserverId
This field is used to store the id retrieved from the *Binding Scope* when observing the *Source Adapter*. We will use it to stop observation once the *Expanded Iteration* node is destroyed.

Managing changes to *Expanded Iteration Tree* and its *Iteration Instances* is the task of the *Iterator*. The *Preprocessor*, however, provides the initial *Plain* and *Expanded Iteration Tree*. Since before *Bindings* are active, the `source` attribute of any *Plain Iteration* node is defined to be the empty collection, the *Expanded Iteration Tree* initially always consists of one element for the root and each of its immediate children. While the root always has one *Iteration Instance*, none of the children have any *Iteration Instances* initially. In the example from figure 6.13 this would mean that only the *Expanded Iteration* nodes `ROOT`, `li` and `country` exist and that `ROOT` has one instance.

The algorithm (see Appendix A.12) setting up the *Expanded Iteration Tree* first creates an *Expanded Iteration* node from the root of the *Plain Iteration Tree* and adds a single *Iteration Instance* to it. We will explain *Iteration Instances* in detail in section 6.3.5. In addition, it creates an *Expanded Iteration* node for each immediate child of *Plain Iteration Tree*'s root and adds it to the previously generated root of the *Expanded Iteration Tree*. When initializing new nodes of the *Expanded Iteration Tree* all information from the originating *Plain Iteration* node is cloned. Therefore, special care has to be taken to update all references to elements of the `template` that occur in the `binding`, `placeholder` and the array of `sockets`. In addition, all *Qualifiers* of *Binding Scope Adapter* that reside inside the `binding` need to be renamed so that they do not interfere with other *Expanded Iterations* originating from the same *Plain Iteration*.

To conclude this section we want to show what the *Expanded Iteration Tree*, created from the *Plain Iteration Tree* from figure 6.11, looks like. Figure 6.14 shows the tree together with all the attributes previously introduced.

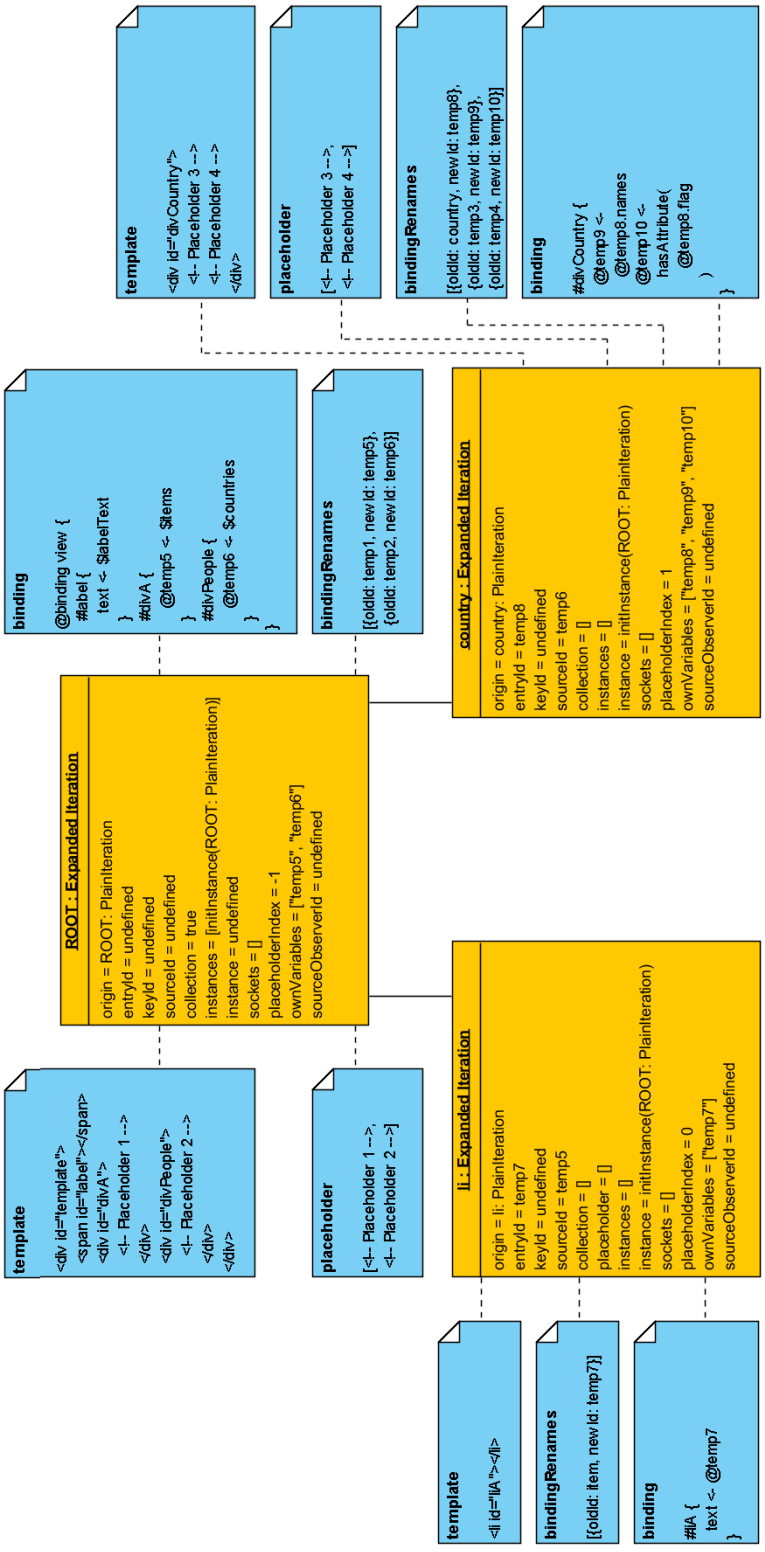


Figure 6.14.: Expanded Iteration Tree, Object Diagram

6.3.5. Iterator

After the *Preprocessor* has initially set up the *Plain* and *Expanded Iteration* trees it is the task of the *Iterator* to manage them by adding, replacing or removing *Iteration Instances*. An *Iteration Instance* comprises the attributes shown in figure 6.15. It represents one element of the iterated `collection`, and does not build a tree itself, since it is attached to an *Expanded Iteration Node*. However, it creates a second structure of references inside the *Expanded Iteration Tree*. Each node is both the child of another *Expanded Iteration Tree*, but also the child of a certain instance inside the array of `instances` of its parent.

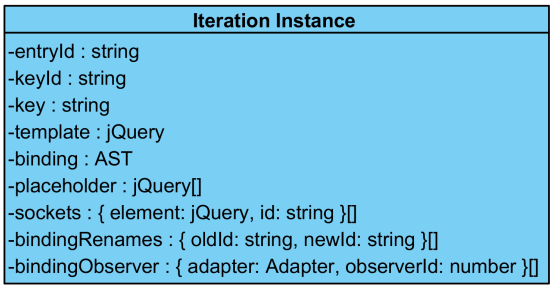


Figure 6.15.: Iteration Instance, Class Diagram

entryId
Qualifier for the *Binding Scope* which is used inside `binding` to access the *Entry*.

keyId
Qualifier for the *Binding Scope* which is used inside `binding` to access the *Key*.

key
Key of the element from the *Expanded Iteration*'s `collection` that is represented by this *Iteration Instance*

template
Reference to a part of the *DOM Fragment* which is managed by this *Iteration Instance*.

binding
Binding Specification that is applied to `template`.

placeholder
List of placeholders that can be used by child instances to plug in their `template`.

sockets
List of sockets used to notify each of them when this *Iteration Instance* is created or destroyed.

bindingRenames
List of renames of *Binding Scope Adapter Qualifier* done for this *Iteration Instance*.

bindingObserver
This list is filled when the `binding` of this *Iteration Instance* is initialized. It is used to stop observing the *Adapter* when the `binding` is shut down.

Activating the *View Data Binding*

Initially, there only exists one *Iteration Instance*, and that is the one for the root *Expanded Iteration*. It is neither removed nor changed and is created by just copying the attributes `template`, `binding`, `placeholder`, `bindingRenames` and `sockets` from the root *Expanded Iteration*. When the *View Data Binding* is mounted the template of this root *Iteration Instance* is inserted at the given position. In the example from figure 6.14 this would be equal to the template shown on the upper left.

When `activate` is called, first, the `binding` for the root *Iteration Instance* is initialized (See Appendix A.13). We will explain how this is done in section 6.3.6. After that, the *Source Adapter* of every immediate child in the *Expanded Iteration Tree* is observed. If it changes its value, the algorithm described in the next section is executed. We expect that, upon initializing the root instance's `binding`, the `collection` of each *Expanded Iteration Node* is set to a new value by that `binding`.

Reacting to Changes

To reflect changes to the `collection` of an *Expanded Iteration Node* the new `collection` from the *Binding Scope* is compared to its previous value. There are two cases *Each* and *When*. If the new collection is a Boolean value, the *Iteration* represents *When*. In that case, there are two possibilities that require a change. First, if the old value was `true` and the new value is `false`, it means that the *Iteration* was previously visible and should now be invisible. To realize this the algorithm acts as if the only *Iteration Instance* was removed from the *Expanded Iteration* node. Second, if the old value was `false` and the new value is `true`, an *Iteration Instance* with a key of `0` and a value of `true` is added.

For the other case where the new collection actually is a collection we use the *Levenshtein Distance* [Lev66] for arrays, and a simple comparison algorithm for name-based hash maps to generate a set of changes that transforms the old into the new collection.

Surgical Updates with Levenshtein Distance

The edit-distance algorithm that we use is well-known as the *Levenshtein Distance* operating on strings . Strings are nothing but a list of characters. With this we can modify the algorithm so that it compares two collections instead of two strings. Also, our algorithm does not only calculate the number of operations, but also tells us which operations need to be performed. There are three types of operations.

add
: A new item needs to be inserted at a certain index

remove
: An old item needs to be removed at a certain index

replace
: An item needs to be replaced by another item at a certain index

To give an intuitive understanding of what the algorithm produces, listing 6.31 shows the result of calculating the *Levenshtein Distance*. The first parameter in line 1 is the old, and the second in line 2 the new collection.

```
var result = levenshtein(["a", "b", "c", "d"],
                         ["b", "c", "e", "f"])
result.each(change => logAndApply(change))
/*
  - Replace "d" with "f" at    position 3
       ["a", "b", "c", "d"]      => ["a", "b", "c", "f"]
  - Add     "e"          after position 2
       ["a", "b", "c", "f"]      => ["a", "b", "c", "e", "f"]
  - Remove  "a"          at    position 0
       ["a", "b", "c", "e", "f"] => ["b", "c", "e", "f"]
*/
```

Listing 6.31: Levenshtein Example Call

With this algorithm (see Appendix A.14) we solve the problem of surgical updates. First, to compare the new to the old collection all *References* in the new collection need to be replaced by their underlying values. The old collection was stored after *References* were already resolved so that it is not necessary to do repeat this procedure.

The implementation uses the dynamic programming approach which needs a two-dimensional matrix to store intermediate results. The space and time complexity of the algorithm is in $O(|\texttt{new collection}| * |\texttt{old collection}|)$ for storing and filling the matrix.

Adding *Iteration Instances*

In figure 6.16 a *Plain Iteration Tree* is shown in blue together with its *Expanded Iteration Tree* in orange. The dotted green lines represent links and the red boxes stand for *Iteration Instances*. The figure shows clearly that nodes of the *Expanded Iteration Tree* actually have *Iteration Instances* as their parent. In the example no *Iteration* other than the root and `B` have any *Iteration Instances*. In this case, `B` has two instances which lead to the duplication of `C` and `D`.

We now want to explain the algorithm (see Appendix A.15) that adds new *Iteration Instances* to any node of the *Expanded Iteration Tree*. First, a new instance is created and a new child *Expanded Iteration* node is added for each child in the corresponding *Plain Iteration* node. After that, the `binding` of the newly generated instance is initialized. We will show this part in section 6.3.6. If the collection that is iterated is an array, the keys of all existing *Iteration Instances* need to be refreshed. All keys that are greater or equal to the newly inserted key need to be increased by one. In the example adding an instance to `B` would mean that the nodes `C` and `D` from the *Plain Iteration Tree* are used to create two new *Expanded Iteration Nodes*.

To create the new instance `binding` `placeholder` and `sockets` are cloned in the same fashion as when creating a new *Expanded Iteration Node*. Every new instance renames all own variables, and the *Qualifier* of all *Entry* and *Key Adapter* if they are used within `binding`. In addition, everything that the parent of the instance renamed is changed according to the same rules. After the *Entry* and *Key Adapter* are renamed, their correct values are set in the *Binding Scope*.

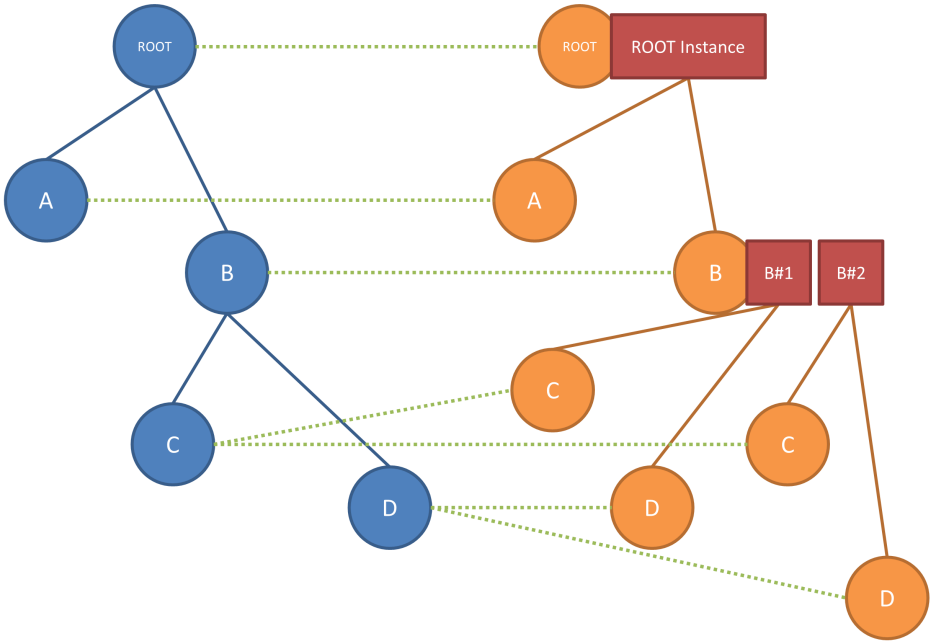


Figure 6.16.: Iteration Components, Simplified

The new template is injected into the template of its parent. Since ultimately the root instance's template is mounted into the view port and every instance injects its template into its parent template, the template of *Iteration Instances* is always visible in the web browser. There are two cases, one where there are already other instances and it is known where to insert the new template, and the other where there is no instance or it is unknown where to insert the template. In the first case, the template is inserted after the template of that instance which has a key equal to the key which was determined by calculating the *Levenshtein Distance*. In the second case, the template is inserted after the correct placeholder of its parent instance. Last, the new instance object is created and added to the *Expanded Iteration Node*. All observers of *Sockets* that were just injected as part of the template are called.

Removing *Iteration Instances*

Removing an *Iteration Instance* does the exact opposite of any step made during its creation in reverse order. First, the algorithm (see Appendix A.16) shuts down the `binding`. We will show how this is done in section 6.3.6. Each child of the instance is destroyed and the instance itself is removed. When removing an instance from an array-based *Iteration* all keys of *Iteration Instances* that are greater or equal to the removed key need to be decreased by one.

An *Expanded Iteration Node* is destroyed by first observing its *Source Adapter*. To recursively destroy its children its collection is manually set to the empty collection or `false` depending on its previous value. Last, the *Expanded Iteration Node* is removed from the `links` of its `origin`.

The *Iteration Instance* itself is removed by first calling all observers of *Sockets* which are part of its `template` that is detached right after. Last, the *Qualifiers* of its *Entry* and *Key Adapter* are destroyed in the *Binding Scope* to stop observing probable *References* that are stored inside them.

Replacing *Iteration Instances*

Replacing *Iteration Instances* is the easiest of the three algorithms. All that has to be done is to notify the affected child about a change to its *Source Adapter* by setting it to its new value in the *Binding Scope*. This indirectly invokes the procedure that reacts to changes. At some point of recursion, the actual structural difference then becomes `add` or `remove`. Consider the model from listing 6.32 where `binding` first iterates over each group of people and then over their names.

```
1 var model = [["Alice", "Eve"], ["Bob", "Mallet"]]
```

Listing 6.32: *Presentation Model*

If now a new name is added to the second array of male names, a call to `replace` at index 1 on the first level of *Iterations* is issued. On the next level the new name is recognized and a call to `add` is made. It should be noted that changes to elements themselves, such as changing the name `Bob` to `Max`, are not handled by the *Iterator* but the *Propagator* (see section 6.3.6).

Deactivating the *View Data Binding*

When `deactivate` is called for an instance of *View Data Binding* the opposite steps performed when activating it are executed in reverse order (see Appendix A.10). The process is very similar to that when destroying instances of *Expanded Iteration Nodes*. The `collection` of each immediate child of the *Expanded Iteration Tree*'s root is set to the empty collection or false and the `binding` is shutdown.

6.3.6. Propagator

The *Propagator* is the component that sets up and shuts down *Bindings* to connect the three *Data Sources* which are the *DOM Fragment* that is part of the *Iteration Instances*, the *Binding Scope* and the *Presentation Model*. It uses both the *Adapter* and *Connector Repositories* to resolve any reference to such components in a *Binding*. We have already shown its basic algorithm in section 5.1.2 and listing 5.12. The algorithm there, however, is simplified and, among other things, does not consider how to exactly address the different types of *Adapter*.

Initializing an *Iteration Instance*'s *Bindings*

The actual implementation is divided into two parts. For each *Scope* in the abstract syntax tree of the *Iteration Instance*'s `binding`, first, all of its *Bindings* are iterated to observe their *Source Adapter*. Whenever this is done for each of the three possible *Adapter* types, the resulting observer id is stored in an array. This array is used to fill the `bindingObserver` attributes of the *Iteration Instance*. It allows to shut down those observers if necessary. Since implementing the `observe` method of an *Adapter* is optional, their presence needs to be checked, and an exception thrown if they are missing.

After all observers are in place each of them is executed in a certain order. First, all *Model*, then all *Binding* and last all *View Adapter* are executed. This is done to ensure that values coming from the *Presentation Model* always have priority over those from the *DOM Fragment*. Since the *DOM Fragment* is inserted right before the *Bindings* are initialized, it usually contains no meaningful values. For example, with a *Two-Way Binding* between the value of a text box and an attribute of the *Presentation Model* the value of the text box initially is always empty. We would expect that it is initialized with the value from the *Presentation Model*, instead of the attribute of the *Presentation Model* being overwritten by an empty string.

Shutting down an *Iteration Instance*'s *Bindings*

All observer ids together with the *Adapter*, where the observer was set, were stored in an attribute of the *Iteration Instance* when initializing its *Bindings*. The only thing that has to be done to shut down these *Bindings* is stopping their observers. This is done by calling the method `unobserve` for each *Adapter* in `bindingObserver`.

Converting a set of *Paths* to structured JSON

As shown in line 18 of listing 5.12 in section 5.1.2, the set of *Paths* returned from an *Adapter* needs to be somehow converted into structured JSON. Listing 6.33 shows the algorithm achieving this.

As its input the algorithm needs the *Adapter* that produced the set of `paths`, the `originalPath`, which was used to call the method `getPaths` and – to initialize *References* – a reference to the *Presentation Model* `model` and the associated `element` from the *DOM Fragment*. First, from lines 2 to 12 the basic structure of the result is created. It is done by iterating all *Paths* and iterating their comprised `items` while leaving out those elements that were already in the `originalPath`. To illustrate this, listing 6.34 shows a sample call to the method and what is stored in `result` after line 12.

```
function convert(adapter, originalPath, paths, model, element) {
  var result = {}
  foreach (path in paths) {
    var pointer = result
    foreach (var j = originalPath.length; j < path.length; j++) {
      var key = path[j]
      if (!pointer[key]) {
        pointer[key] = {}
      }
      pointer = pointer[key]
    }
  }

  foreach (path in paths) {
    var pointer = result
    for (var j = originalPath.length; j < path.length - 1; j++) {
      var key = path[j]
      pointer = pointer[key]
    }

    if (path.length - originalPath.length > 0) {
      if (pointer[path[path.length - 1]] == {})) {
        var reference = new Reference(adapter, path)
        if (adapter.type() == "view") {
          reference.setElement(element)
        } else if (adapter.type() == "model") {
          reference.setModel(model)
        }
        pointer[path[path.length - 1]] = reference
      }
    } else {
      var reference = new Reference(adapter, path)
      if (adapter.type() == "view") {
        reference.setElement(element)
      } else if (adapter.type() == "model") {
        reference.setModel(model)
      }
      result = reference
    }
  }
  result = recognizeArrays(result)
  return result
}
```

Listing 6.33: Converting a set of *Paths* to Structured JSON

```
1  var originalPath = ["p"]
2  var paths        = [["p", 0, "name"],
3                      ["p", 0, "age"],
4                      ["p", 1, "name"],
5                      ["p", 1, "age"]]
6  convert(..., originalPath, paths, ...)
7  /*  resultAfterLine12 =
8        {
9          0: {
10           name: {},
11           age: {},
12         },
13         1: {
14           name: {},
15           age: {}
16         }
17       }                    */
```

Listing 6.34: Example Call to `convert`

Wherever there are empty objects `{}` in `result`, this entry cannot be structured further and must be a primitive value. For each *Path* this condition is checked in line 22 after navigating to the correct position in `result` from lines 15 to 19. The condition in line 21 is false if, and only if, `path` and `originalPath` are equal. This means that there is only one element in `paths` and that `result` is empty. This implies that `result` itself needs to be a *Reference* which is initialized in lines 32 to 38. An example when this is happening is the *Value Adapter* that we have presented in listing A.5. `Convert`, in that case, would be called with `originalPath` = `[]` and `paths` = `[[]]`. As the last step, arrays are recognized by checking the keys of every element on each level. If they form a contingent series of numbers starting at `0`, the element is converted into an array. For our previous example the overall result of the method is shown in listing 6.35.

```
1  var result = [
2    {
3      name: Reference(adapter: ..., path: ["p", 0, "name"]),
4      age:  Reference(adapter: ..., path: ["p", 0, "age"])
5    },
6    {
7      name: Reference(adapter: ..., path: ["p", 1, "name"]),
8      age:  Reference(adapter: ..., path: ["p", 1, "age"])
9    }
10 ]
```

Listing 6.35: Example Result of Call to `convert`

Converting *References* to *Values*

Before writing the structured JSON back to the *Sink Adpater* of a *Binding*, it needs to be converted so that any remaining *References* are replaced by their values. This can be achieved by a recursive algorithm. For the example from listing 6.35 this would mean that each *Reference* is replaced by whatever is returned by a call to its `getValue` method.

Cycle Detection

Our *Concepts* allow to specify cyclic *Bindings*. The easiest example of this is shown in listing 6.36, but there could be also cycles that span over more than two *Bindings* such as the one in listing 6.37.

```
1 value <- $value + 1
2 value -> $value
```

Listing 6.36: Simple Cycle

```
1 @foo <- $value + 1
2 value <- @foo
3 @bar <- value
4 $value <- @bar
```

Listing 6.37: Complex Cycle

The first observation is that such cycles cannot be recognized by building a graph of *Adapter* that influence each other, and by checking this graph for cycles. Such a graph for listing 6.37 is shown in figure 6.17. The simplest counter example why this is not working is any *Two-Way Binding*.

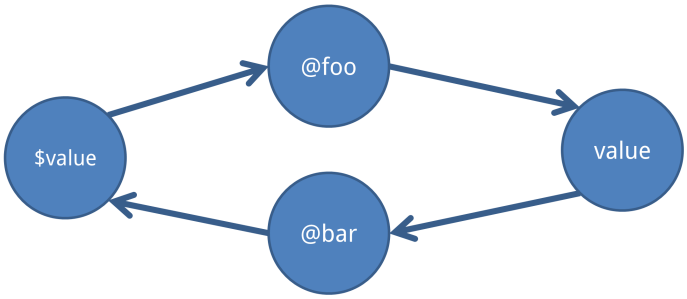


Figure 6.17.: Influences of Bindings as Graph

Since we do not support a *Connector* that delays the propagation of a *Binding*, the second observation is that there are no useful cycles [Mey+]. This means that a cycle is always an error, and the only reason why we would want to detect it is to help a developer finding that error. This behavior is well-known from Microsoft Excel where formulas of cells can create cycles. If that is the case, Microsoft Excel highlights the affected cells and warns the user about the erroneous input.

To solve the issue we could implement *Propagation* so that whenever the value of an *Adapter* changes, first, all possibly affected other *Adapters* are collected. If any of those *Adapter* is then written more than once with a different value, a cycle would be detected. This, however, requires that there is a way to differentiate between *Adapter*

callbacks that were caused by an external change and such that arose because *Propagation* internally changed the value of the *Adapter*. Including this feature into the *Adapter* API would, however, create an unreasonable amount of code, which would have to be implemented for each *Adapter*. Also, considering that cycle detection does not increase the expressiveness of our *Concepts*, but would only help debugging errors, we decided not to include any cycle detection into *BindingJS*.

Pausing and Resuming *Propagation*

To pause an instance of *View Data Binding* every place where any *Data Source* is observed needs to be modified so that the callback of the observer checks if the *View Data Binding* is currently paused. In total, there are two places where such an observation happens, first in the *Propagator* and second the *Binding Scope*. Instead of setting up the observation, as shown in listing 6.38, we modify its callback so that it stores all incoming notifications as in 6.39.

```
1 dataTarget.observe(
2   id, function () {
3     callback()
4   }
5 )
```

Listing 6.38: Observation

```
1 dataTarget.observe(
2   id, function () {
3     if (!paused) {
4       callback()
5     } else {
6       pauseQueue.push(callback)
7 }})
```

Listing 6.39: Pausable Observation

By doing this no *Propagation* is performed while the *View Data Binding* is paused. Upon resumption all that has to be done is to execute all accumulated callbacks from the two queues of the *Binding Scope* and the *Propagator*.

6.3.7. Sockets

The *Iterator* takes care about calling the observers for certain *Sockets* when *Iteration Instances* are added and removed. The *View Data Binding* instance stores a list of both types of observers and adds all callbacks that come in through `onInsert` and `onRemove` to those lists. When an *Iteration Instances* is added or removed, all of the *Labels* of its *Sockets* are compared to those previously registered and each callback is called. The information about `sockets`, their `id` and associated `element` that are required to execute the callback are already provided as attributes of any *Iteration Instance*. The `keys` array can be easily built by recursively traversing the tree of *Expanded Iterations*.

To implement the methods `instances` and `instace` of the *Socket* API we use an algorithm that recursively traverses all *Iteration Instances* and their `sockets` attributes.

7. Evaluation, BindingJS

In this chapter we evaluate the result of our work, *BindingJS*, by scoring it in the same fashion as in chapter 3. In addition, we use the examples from section 2.2 to show how *BindingJS* can be used to put them into action and what benefits come with its application.

7.1. Comparison with other Rich Web Client Libraries

In section 3.3 we evaluated and compared different *View Data Binding* libraries. To figure out if our approach and implementation are viable alternatives to the many libraries already present, we use the criteria catalog from section A.1.1 to grade *BindingJS*.

Compatibility

Criterion	Score	Reason
Adaptability to Hierarchical Model	+1	Being compatible with *ComponentJS* was one of our goals that we reached by performing all accesses to the *Presentation Model* exclusively through *Adapter*.
Browser Support	0	*BindingJS* was tested with the most recent versions of all relevant browsers. Since we strive for a modern solution and use cutting-edge technology, such as the latest language version of JavaScript and HTML 5 features, we cannot guarantee that *BindingJS* is compatible with older web browsers.
Compatibility with jQuery	+1	jQuery is the only dependency of *BindingJS*
Extensibility	+1	Users of *BindingJS* may plug-in arbitrary *Adapter* or *Connectors* through its plug-in API (see section 6.2.4).
Observation Mechanism	+1	All accesses to the *Presentation Model* are performed exclusively through *Adapter* in *BindingJS*, and it is easy to add new *Adapter*.
Total	**+4**	

Table 7.1.: *BindingJS* — Compatibility

Design

Criterion	Score	Reason
Amount of Logic in View	+1	Since *BindingJS* primarily uses an external file to define the *Binding Specification*, the view component typically comprises no logic at all. Iterating and conditionally displaying elements is possible, but it is not visible in the view component. Therefore, a grade of zero would not be justified.
Binding Separable from View	+1	This is one of the unique characteristics of *BindingJS*.
Efficiency by employing Surgical Updates	+1	*BindingJS* uses a modified edit distance algorithm to minimze the number of changes to *Iterations* (see section 6.3.5).
Intermediate Representation	+1	The layers of abstraction in *BindingJS* are its *Iteration Trees* and *Iteration Instances* (see section 6.3.5).
Paradigm for Binding	+1	Specifying the *Binding Specification* in an external file covers the declarative case. We even plan on adding the feature that this can be done inside the view component (see chapter 8). In addition, *BindingJS* offers to imperatively add new *Bindings* by spawning new instances of *View Data Binding* through `create`. This can be done at any time with arbitrary *Bindings* that might be defined in-place.
Limitation of Scope to Data Binding	+1	The only parts of *BindingJS* in question could be *Identification*, *Sockets*, and *Templates*. As we have shown in chapter 5, leaving these out would render the library inappropriate for modular single page applications.
Template Sources	+1	*BindingJS* offers four different ways of providing the *Template DOM Fragment* (see listing 6.4).
Total	**+7**	

Table 7.2.: *BindingJS* — Design

Feature Coverage

Criterion	Score	Reason
Animations	-1	We decided to not support animations because they present logic that typically should not be executed by a *View Data Binding* library.
Binding to DOM Events	+2	The on *Adapter* allows to store arbitrary DOM events in the *Presentation Model*.
Chains	+1	Performing multiple transformation steps inside a *Binding* is possible by using a *Connector Chain*.
Converter	+1	Modifying values as they are propagated through a *Binding* is a typical use case for a *Connector* (see section 5.1.2.2).
Expressions	+2	The set of *Expressions* provided by *BindingJS* is rich (see section 5.3.6) and even includes regular expressions.
Filter	+1	This feature can be realized with a *Connector*. An example of a *Connector* modifying a list is shown in listing 6.12. Although the implementation sorts the list, it should be easily imaginable how it can be changed into a *Filter*.
Subscription to Events Triggered Internally	0	Although it is possible to register *Socket* observer, in general, there currently is no mechanism of listening for changes of individual *Bindings*.
Two-way binding	+1	This feature is a *Convenience Concept* of *BindingJS* (see section 5.3.1).
Validation	+2	Validation tasks may be performed by *Connectors*. Since their logic is implemented with JavaScript, there are no limitations regarding its complexity.
Total	**+9**	

Table 7.3.: *BindingJS* — Feature Coverage

Usability

Criterion	Score	Reason
Changes necessary to Existing Solution	+1	We do not expect that much of an existing solution has to be rewritten in order to employ *BindingJS*. However, there lies effort in writing a suitable *Presentation Model Adapter* if none is available already.
Efficiency in Big Applications	0	Although our implementation is designed to be as efficient as possible even in big applications, there are currently no proofs other than theory for this.
Learning Curve	+1	We took care that the syntax of a *Binding Specification* defined with *BindingJS* remains as readable as possible. With the high number of examples within this thesis we expect the learning curve to be steep.
Number of Dependencies	0	jQuery is the only dependency of *BindingJS*
Size of the Framework	0	The *Parser* is responsible for most of the libraries code, but with currently 134kB it does not exceed 200kB
Intuitivity and Maintainability of the Syntax	+1	During the process of designing *BindingJS*, we performed multiple interviews with software engineers. Within those short talks, the person without any prior knowledge of *BindingJS* or its syntax was presented a *Binding Specification* and asked what he or she thinks is expressed by it. By evaluating the results from these answers we could increase the intuitivity of our syntax step by step and received positive feedback.
Total	**+3**	

Table 7.4.: *BindingJS* — Usability

Support

Criterion	Score	Reason
Number of Answered Questions on StackOverflow.com	-1	Since *BindingJS* has not been promoted and published, there is no community activity yet.
Number of Contributors on GitHub	0	The current contributors to the project are the three authors.
Number of Stars on GitHub	-1	*See Number of Answered Questions on StackOverflow.com*
Google Query about Typical Question	-1	*See Number of Answered Questions on StackOverflow.com*
Quality of Documentation	+1	This thesis comprises the documentation of *BindingJS*. All features and their implementations are explained in detail and illustrated with examples.
Total	**-2**	

Table 7.5.: *BindingJS* — Support

Reliability

Criterion	Score	Reason
Architectural Cleanliness	+1	We defined the architecture of *BindingJS* in section 6.1.
Implementation Cleanliness	0	The algorithms operating *BindingJS* are already difficult to understand when presented as simplified pseudo-code. Although the actual implementation uses comments, without the accompanying notes from this thesis and with the additional complexity because of technical limitations, the code of *BindingJS* cannot be fully understood easily and needs to be optimized for this in the future.
Maturity	-1	The implementation of *BindingJS* was initiated at the same time as writing this thesis and, therefore, cannot be more mature than six months.
Development Activity	+1	
Test Coverage	0	*BindingJS* employs *mocha* for unit and *Selenium IDE* for integration tests (see section 6.1). Their coverage, however, needs to be increased in the future.
Version	-1	*See Maturity*
Total	**0**	

Table 7.6.: *BindingJS* — Reliability

Interpretation

Summing up all intermediate scores yields a total score of twenty-one which is more than our previous winner *KnockoutJS* graded with eighteen points. Although, *BindingJS* has the obvious disadvantage of its low scores in the *Support* and *Reliability* category, it is convincing because of its *Design* and its *Compatibility*. Our goal was to create a separable, reactive, model-agnostic approach to *View Data Binding*. Each of these attributes materializes itself especially within those two categories that are not adequately addressed by existing solutions. Many of the features that we evaluated in the *Support* and *Reliability* category will improve in the future, once *BindingJS* is made available to a larger public and is provided time to mature. All scores that were presented should be taken with a grain of salt. Although they seem to be absolute if different features, evaluation instructions or weights were used, the result might be completely different.

They, however, show clearly that *BindingJS* is not just another *View Data Binding* library, but has unique characteristics while uniting many advantages of existing solutions. If it manages to be accepted by a broader audience, it could have significant impact on future and existing web applications due to its holistic and flexible nature.

7.2. Employment in Applications

In section 2.2 16 lines of code were necessary to set up a *Two-Way Binding* between a text box and an attribute from the *Presentation Model*. We can now express the same in listing 7.1 with the help of *BindingJS*. Both *Presentation Model* and *Template DOM Fragment* are unchanged, but instead of unmaintainable boilerplate code only three lines of code are necessary to express the *Binding* from lines 11 to 13. In addition, the library itself needs to be set up from lines 22 to 30 and included from lines 6 to 8. This means that, in total, the amount of code has only been reduced marginally. The maintainability and readability of the code, however, has tremendously increased. *BindingJS* also was not designed to express such minimal examples with the least amount of code possible. The initialization effort always stays the same no matter how much is added to the *Binding Specification*. If we extend the example so that not only one name from the *Presentation Model* is bound, but a list of names, in *BindingJS* this would only require adding an *Iteration* in line 11. Doing this with JavaScript would require a great amount of code even if surgical updates were not considered.

```
1  <html>
2    <head>
3      <title>BindingJS - Textbox</title>
4      <script src="jquery.js"></script>
5      <script src="binding.js"></script>
6      <script src="valueAdapter.js"></script>
7      <script src="jsonAdapter.js"></script>
8      <script type="text/binding">
9        // Binding Specification
10       #username {
11         on:change +> value <-> $user.name
12       }
13     </script>
14     <script type="text/javascript">
15       // Presentation Model
16       var model = {
17         user: { name: "Johannes" }
18       }
19
20       $(function() {
21         BindingJS
22           .plugin("value", ValueAdapter)
23           .plugin("$", JsonAdapter)
24           .create()
25           .template("#template")
```

```
26            .binding($("script[type='text/binding']"))
27            .model(model)
28            .mount("#template")
29            .activate()
30        })
31      </script>
32    </head>
33    <body>
34      <!- Template DOM Fragment ->
35      <div id="template">
36        <input id="username" type="text" />
37      </div>
38    </body>
39  </html>
```

Listing 7.1: Binding a Text Box

7.2.1. TodoMVC

Since a full implementation of *TodoMVC* (see section 2.2.1) would go beyond the scope of this section, we restricted listing 7.2 to the most important features. It is structured into three parts, the *Binding Specification* from lines 6 to 47, the *Presentation Model* from lines 50 to 60 and the *Template DOM Fragment* from lines 75 to 93. The *Presentation Model* comprises a function `getRemaining` and a list of `todos` that each have the attributes `title` and `completed`. The *Template DOM Fragment* is structured into a header where new todos may be entered into the text box from lines 78 and a repeating part for each todo from lines 82 to 89.

If a user enters text into the text box from line 78 and presses enter, a new todo having the text entered as its title is added to the list of todos and the value of text box is reset to an empty string. We can easily realize this behavior with the two *Bindings* from lines 8 and 10. The *Connector* `push` adds the element that it receives as its *Parameter* to its input. By using a *Hash Expression* it is possible to directly express the new todo item as a *Parameter*. In addition, the *Expression* neatly uses the `value` *Adapter* to initialize the field `title` of the *Hash*. The *Binding* from line 8 itself would create a cycle that would add new todos over and over, but since it is initiated by the `on` *Adapter* it is only executed if the enter key is pressed. The *Binding* from line 10 resets the value of the text box to an empty string. To ensure that it is executed after the *Binding* from line 8, it listens for the `keyup` instead fro the `keydown` event through the `on` *Adapter*.

For every todo item the part of the *Template DOM Fragment* from lines 82 to 89 is repeated. The *Concept* of *Iteration* is perfect to realize this in line 13. The current todo item and its index are stored in `@todo` and `@index` respectively.

- Each todo item comprises a check box that reflects the `completed` state of the item. Only line 15 is required to create this connection. Since the `attr` *Adapter* usually does not implement an `observe` method, it is used together with an *Initiator* that starts the propagation of the *Binding* whenever the check box is clicked. It should be noted that a change to the *Presentation Model* is always reflected by

the state of the check box because the *Initiator* does not affect this direction of the *Binding*.

- Apart from marking a todo item as completed, it can also be deleted by pressing the button from line 86. The *Binding* in line 18 that realizes this behavior works similar to the one adding new todo items. The *Connector* `destroy` removes the element from its input that is at the index of its *Parameter*.
- The label from line 85 displays the title of each todo item to the user which presents a good use case of the `text` *Adapter* in line 21.
- By double clicking any todo item a user may edit its title. The editing can be aborted by pressing escape so that whatever was entered into the text box from line 88 is discarded. If, however, enter is pressed the title of the todo is overwritten by the new value. Through CSS the text box is only visible to the user if the class `editing` is added to the `li` element from line 82. By default, any todo item is not in the editing state which we express by the *One-Time Binding* from line 24, initializing the *Binding Scope Adapter* `editing` with false once. Line 25 expresses that the class `editing` should be added to the `li` element whenever `@editing` becomes true and removed otherwise.

  Line 28 stores the todo's title into `@tempTitle`. Since by default, only a *Reference* would be stored the *Connector* `getValue` is used that resolves this *Reference* to its actual underlying value. This temporary title can now be used to bind it to the value of the text box in line 31. This way, the *Presentation Model* is not directly synchronized, but only if enter is pressed in line 29 which also leads to the editing mode being left in line 30. To implement the abort behavior we use a *Sequence* of *Initiators* to also leave the editing mode if escape is pressed.

  Last, there needs to be a way to enter the editing mode which is realized in line 34 that sets `@editing` to true whenever the `label` from line 85 is double clicked.

A footer after all todo items shows how many todos are still uncompleted. It contains text, such as *3 items left* or *1 item left*. First, to determine the number of items left we use the function `getRemaining` from the *Presentation Model* in line 40. Although a function can be observed, a notification would only be issued if the definition of the function changes. The expected behavior, however, is that the *Binding* from line 40 is propagated whenever `getRemaining` might return a different value. We can make the attributes from the *Presentation Model* that the result of the function depends on explicit by using them as *Initiators*. Line 44 decides whether the singular or plural of the term *item* needs to be used to correctly display the text. The *Connector* `count` is an *Aggregator* that returns the number of elements in its input collection. Based on this number, line 44 changes the text appropriately. It should be noted that we could only use `$todos.length` if the *Presentation Model* includes the `length` property in its *Paths* returned for `todos`.

Although listing 7.2 is shortened and does not show a full implementation of *TodoMVC*, it clearly illustrates how much logic can be expressed with just around 40 lines of code for the *Binding Specification*. By combining all of our *Concepts* intelligently, complex semantics can be expressed in a compact way without polluting the *Template DOM Fragment* or any other place with additional information.

```
<html>
  <head>
    <title>BindingJS - TodoMVC</title>
    <!- Include stylesheet, jQuery, BindingJS, Adapter & Connectors ->
    <script type="text/binding">
      @binding TodoMVC {
        #new-todo {
          on:keydown("enter") +>
            $todos -> push({ completed: false, title: value }) -> $todos
          on:keyup("enter") +> "" -> value
        }

        #todo-list li (@todo, @index: $todos) {
          .view input {
            on:click +> attr:checked <-> @todo.completed
          }
          .view button {
            on:click +> $todos -> destroy(@index) -> $todos
          }
          .view label {
            text <- @todo.title
          }

          @editing <~ false
          class:editing <- @editing

          .edit {
            @tempTitle <- getValue <- @todo.title
            on:keydown("enter") +> @tempTitle -> @todo.title
            on:keydown("enter"), on:keydown("esc") +> false -> @editing
            value <-> @tempTitle
          }
          label {
            on:dblclick +> true -> @editing
          }
        }

        #stats {
          strong {
            text <- $getRemaining <+ $todos
          }
          span {
            @count <- count <- $todos
            text <- @count == 1 ? "item" : "items"
          }
        }
      }
    </script>
```

```
49      <script type="text/javascript">
50        var model = {
51          todos: [],
52          getRemaining : function () {
53            var count = 0
54            for (var i = 0; i < model.todos.length; i++) {
55              var item = model.todos[i]
56              if (item.completed) {
57                count++
58            }}
59            return count
60        }}
61
62        $(function() {
63            var binding = BindingJS
64                .create()
65                .plugin(/* ... */)
66                .template("#todoapp")
67                .binding($("script[type='text/binding']"))
68                .model(model)
69                .mount("#todoapp")
70                .activate()
71        })
72      </script>
73    </head>
74    <body>
75      <section id="todoapp">
76        <header id="header">
77          <h1>todos</h1>
78          <input id="new-todo" placeholder="What needs to be done?">
79        </header>
80        <section id="main">
81          <ul id="todo-list">
82            <li>
83              <div class="view">
84                <input class="toggle" type="checkbox">
85                <label></label>
86                <button class="destroy"></button>
87              </div>
88              <input class="edit">
89            </li>
90          </ul>
91        </section>
92        <div id="stats"><strong></strong> <span></span> left</span></div>
93      </section>
94    </body>
95  </html>
```

Listing 7.2: TodoMVC

7.2.2. msg TimeSheet

In section 2.2 we have introduced an internal application that is used at msg systems AG. It can be used by employees to register their working hours and offers both a desktop and a mobile layout which automatically switches based on the size of the view port. To evaluate *BindingJS* we modified one of its modules called input screen to use the library instead JavaScript code for its *View Data Binding*. The composite which is shown in figure 7.1 is only visible in the mobile version of the *TimeSheet* application. It is used to enter data about the distribution of working hours on a single day. Apart from working hours, it is possible to specify what portions of the day have been used for specific projects at the bottom of the screen.

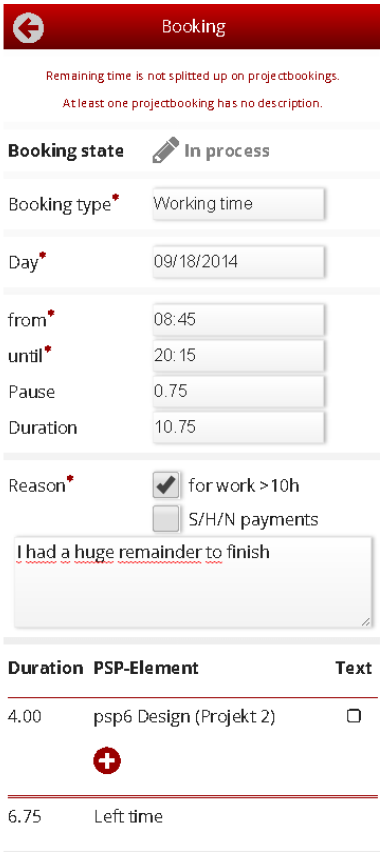


Figure 7.1.: Input Screen, msg TimeSheet

To operate the module lots of *View Data Binding* is required. Interactions that are made by the user result in setting event attributes in the *Presentation Model*. Based on the values entered in the form, certain elements appear or are hidden. For instance, the text box that allows a user to enter a reason for working more than ten hours only appears if the duration becomes greater than ten hours, or the appropriate tick is set.

Since the *TimeSheet* application uses *ComponentJS* to structure its code, the module consists of mainly four components. The **template** which comprises HTML code providing the initial visual structure of the *Composite*. The **controller** written in JavaScript that reacts to changes to the model or that changes the model. The **view** that sets up the template and in particular its *View Data Binding*. The *ComponentJS* **model** that equals our understanding of a *Presentation Model*.

When employing *BindingJS* mainly the view component is changed so that all *View Data Binding* related logic is extracted from it into an own file comprising a *Binding Specification*. Since the code of the application is not publicly available and confidential, we measure the impact of *BindingJS* in lines of code. While all other modules did not significantly increase or decrease their amount of code, we could cut down the view component from 802 to only 601 lines of code by only adding 41 lines for the *Binding Specification*. This means that we could reduce the amount of *View Data Binding* related code by around 20 percent[1] with the help of *BindingJS*. Since around 200 lines of the view's code are made up of code handling *mobiscroll*, a plug-in that provides scrolling dials, the actual percentage of reduction could even be seen as around 25 percent[2]. Also, it is worth noting that the old implementation did not consider surgical updates that are provided by *BindingJS* without additional effort.

1 $\frac{601+41}{802} \approx 80,1\%$

2 $\frac{601-200+41}{802-200} \approx 73,4\%$

8. Conclusion and Future Work

Web applications today are complex and need powerful tools that abstract away repetitive tasks like *View Data Binding*. To approach the topic of our work we first made clear what the problem domain is in the chapters 1 and 2. The task of *View Data Binding* typically appears in *Rich Web Clients* that mainly use one of the architectural patterns Model-View-Controller or Model-View-ViewModel. We introduced these patterns and showed how inefficient and lengthy typical use cases are solved without using a *View Data Binding* library. Thus, we made clear why such solutions are required urgently and what their characteristics should be. They should simplify and reduce the amount of code that has to be written. In addition, the library should be as flexible as possible and offer a reactive user experience. Since *View Data Binding* is an important concept on its own, mixing it into the view or model component seemed inappropriate and like inadequate design to us. Therefore, we state that another requirement should be that it should be possible to define *View Data Binding* separately to follow the principle of *Separation of Concerns*.

In chapter 3 we examined existing solutions targeting the problem of *View Data Binding*. We illustrated their advantages and disadvantages, and to be able to compare them we evaluated them against a criteria catalog. During this examination, we gained vital experience helping us in defining the features that a library should offer. In conclusion, we found that no solution currently available offers the characteristics we strive for. Although they reduce the amount of code, they mostly lack the flexibility to adapt to arbitrary *Presentation Model* implementations. In addition, there is no library that allows to define the *View Data Binding* separately.

To make a relevant contribution to the field we clearly defined the concepts that are necessary to realize *View Data Binding*. We took care that these concepts are independent of an implementation, clearly state their necessity and that they are minimal. As a result, we found that there are just three essential concepts for realizing *View Data Binding*, while one additional concept is necessary to satisfy the needs of structuring an application. To realize the features that most other solutions offer and to increase the usability of these concepts, we added further non-essential concepts. However, we could show that these syntactical features can all be reduced to essential concepts and do not increase the expressiveness of the solution.

For each such concept we defined its syntax with a domain specific language for *View Data Binding* that is optimized for readability, compactness and maintainability. By defining their semantics we could already take care that any implementation of these concepts is model-agnostic. In addition, we discovered the importance of a third *Data Target*, the *Binding Scope* which is needed to realize *View Data Binding* apart from the *Presentation Model* and *DOM Fragment*.

```
(value | css:color <- colorify)
  <- $name

value ->
  ($name | abbrev -> $initials)
```

Listing 8.1: Parallel *Binding*

```
value <- $name
css:color <- colorify <- $name

value -> $name
value -> abbrev -> $initials
```

Listing 8.2: Transformation

One concept that could be added to make our syntax even more compact in the future is shown in listing 8.1. It allows to combine two or more *Bindings* into one statement by using a new parallel operator. Obviously, this concept is not essential and an idea about how it can be transformed is shown in listing 8.2.

In chapter 6 we presented *BindingJS* as a proof that our concepts can be implemented. The JavaScript library provides an extensible and feature-rich API. *Adapter* and *Connectors* can be easily plugged in to adapt the solution to almost any environment. The challenge was finding preprocessing steps that transform the input components simplifying the remainder of the implementation. By using multiple trees we presented an efficient way to implement the concept of *Iteration*. We were attentive that changes to the *DOM Fragment* are surgical, meaning that the number of modifications is minimized when the iterated collection changes. To provide a reactive user experience we designed the propagation of *Bindings* to be asynchronous. By following the observer pattern all modifications are visible immediately.

In the future we plan on adding more *Adapter* and *Connectors* to fit more and more use cases out of the box. Most importantly, the implementation of *Convenience Concepts* which will be the most crucial improvement of *BindingJS'* usability still remains open. Apart from offering to specify a *Binding Specification* separately or imperatively, we want to allow defining it declaratively, as well. Listing 8.3 shows an example of a *Template DOM Fragment* that comprises *View Data Binding* information in attributes of its HTML elements.

```
<div id="template">
  <input type="text" data-bind="value <-> $name">
  <div data-iterate="@name:  $names">
    <div data-bind="img { attr:src <- getAvatar <- @name }"
      <img />
    </div>
  </div>
</div>
```

Listing 8.3: Declarative *View Data Binding*

We can include this feature without much effort by splitting up the declarative code into a *Template DOM Fragment* and a *Binding Specification*. This way, we can reuse apart from some early preprocessing steps all of *BindingJS'* code. Besides declarative *View Data Binding*, we consider dynamic template an interesting opportunity to improve *BindingJS*. At the moment, the *Template DOM Fragment* is a static input that cannot be changed after the *View Data Binding* is set up. Allowing this, however, would enable use cases where a user interface is exchanged seamlessly.

Further, we see possibilities to simplify the development process of a *Binding Speci-*

fication. For example, some *Bindings* could be set up automatically if a helpful default behavior is found. For instance, if the `id` attribute of an HTML tag equals the name of field from the *Presentation Model*, these two could be bound automatically. To abstract away from our domain specific language, we think, there might even be a graphical user interface for defining a *Binding Specification*. By showing all available items from the *Presentation Model* and the *Template DOM Fragment*, as well as all available *Connectors*, a *Binding* could be created with drag and drop.

In chapter 7 we evaluated how *BindingJS* can compete with the libraries that already exist and how it can be used in real world applications. We found that *BindingJS* is a promising alternative and has a high potential of becoming a popular *View Data Binding* library. Sure, this requires effort from our side in promoting our solution to a broader audience by offering support, improvements and sharing information about the solution in communities. When including *BindingJS* in *TodoMVC* and an internal application at msg systems AG, we could see the potential of the framework which lead to dramatic reduction of code that is still maintainable, flexible and well readable. In conclusion, we reached all of our goals and provided a holistic, separable, reactive and model-agnostic library with a solid conceptual foundation.

A. Appendix

A.1. Comparison of Rich Web Client Libraries

A.1.1. Criteria Catalog

This catalog explains the criteria that were used for comparing different view data binding frameworks in section A.1.2. It also describes how these criteria are scored.

Design

Amount of Logic in View Does the framework restrict the amount of logic in the view to the typical needs of a view component (e.g. iterations, conditions, etc.)?

- **-1** No (Use of JavaScript)
- **0** Partially (No logic at all)
- **+1** Yes

Binding Separable from View Is it possible to define the binding separately?

- **-1** No
- **0** Partially or unknown
- **+1** Yes

Efficiency by employing Surgical Updates Does the framework apply an algorithm to minimize the number of changes to the DOM when updating collections or nested structures?

- **-1** No
- **0** Partially or unknown
- **+1** Yes

Intermediate Representation Does the framework make use of one or more layers of abstraction between the view and data model or are update operations directly performed on the DOM?

- **-1** No
- **0** Partially or unknown
- **+1** Yes

Paradigm for Binding Does the framework offer both the imperative and declarative paradigm to define bindings?

- **-1** No (Imperative or declarative definitions only)
- **0** Partially or unknown

+1 Yes

Limitation of Scope to Data Binding Is the binding aspect the main purpose of the framework?

-1 No (Covers wide areas of web development)

0 Partially (Focuses primarily on binding, but also offers minor additional features)

+1 Yes (Provides functionality for binding only)

Template Sources Does the framework offer many or only few possibilities to include the template? (Possibilities include passing a string, referencing an existing DOM fragment or referencing an external file)

-1 Few (Exactly one)

0 Between

+1 Many (More than two)

Compatibility

Adaptability to Hierarchical Model Is it possible to adapt the solution to a hierarchical model like the one used in ComponentJS?

-1 No (The lookup mechanism cannot easily be changed to perform method calls instead of an attribute access for lookups)

0 Maybe (The lookup mechanism can be changed easily)

+1 Yes (The framework already provides the required flexibility)

Browser Support Does the framework support an adequate set of different web browsers?

-1 No (Only most recent versions of Chrome or Firefox)

0 Partially (Most recent versions of Chrome, Firefox and Internet Explorer)

+1 Yes (All relevant versions of Chrome, Firefox and Internet Explorer)

Compatibility with jQuery Does the framework easily work together with jQuery?

-1 No (Cannot be used with jQuery)

0 Maybe or unknown

+1 Yes (Is compatible with jQuery or depends on jQuery)

Extensibility Does the framework provide an easy way to add custom plug-ins?

-1 No (Not possible)

0 Partially (Possible, but difficult)

+1 Yes (Well documented with examples)

Observation Mechanisms Is the framework flexible regarding different presentation model implementations?

-1 No (Lacks adapter concept or unreliable access through adapter)

0 Partially (Exclusive access through adapter, adding new adapters is difficult)

+1 Yes (Exclusive access through adapter, new adapters can be added easily)

Feature Coverage

Animations Is it possible to animate value changes?

- **-1** No
- **0** Possible, but not integrated
- **+1** Yes

Binding to DOM Events Does the solution allow binding to DOM events?

- **-1** No
- **0** Possible by providing a callback function
- **+1** DOM events can be stored in the model, but only common DOM events
- **+2** Arbitrary DOM events may be stored in the model

Chains Does the framework allow chaining features like doing validation first, then a conversion and last an animation?

- **-1** No
- **0** Possible, but not integrated
- **+1** Yes

Converter Is it possible to convert bound values using custom converters?

- **-1** No
- **0** Maybe or Unknown
- **+1** Yes

Expressions Can bound values be modified using expressions? How rich is the set of available expressions?

- **-1** No
- **0** Partially (Using simple arithmetic or comparison operators)
- **+1** Yes with restrictions (Offering a complex set of possibilities like method calls or index access)
- **+2** Yes (Offering a complex set of possibilities including regular expressions)

Filters Is it possible to filter collections?

- **-1** No
- **0** Possible, but not integrated
- **+1** Yes

Subscription to Events Triggered Internally Is it possible to subscribe to events that are triggered when a bound value will change or has changed?

- **-1** No
- **0** Unknown
- **+1** Yes

Two-way binding Apart from binding values in one direction, does the solution offer two directional binding when appropriate?

-1 No

0 Unknown

+1 Yes

Validation Is there a validation mechanism included and how rich is it?

-1 No

0 Possible, but not integrated

+1 Yes with restrictions (Only for simple validation like range, length, character set checks)

+2 Yes (Including complex validation like email address, zip code checks)

Usability

Changes necessary to Existing Solution Is it possible to use the framework in an existing solution without rewriting most of the code?

-1 No

0 Partially (Moderate amount of code)

+1 Yes

Efficiency in Big Applications Does the solution perform well in a big web application?

-1 No (Only suitable for small projects)

0 Maybe (Theoretically performs well in big projects)

+1 Yes

Learning Curve Can an average software engineer understand and apply the framework after a short amount of time?

-1 No

0 Maybe

+1 Yes

Number of Dependencies Does the solution waive the need for dependencies that have to be included manually?

-1 No (More than two)

0 Partially (One or two)

+1 Yes (None)

Size of the Framework Does the minified version of the framework not require a lot of space?

-1 No (More than 200kB)

0 Between

+1 Yes (Less than 50kB)

Intuitivity and Maintainability of the Syntax Is a software engineer unfamiliar with the framework able to understand the intention of the binding code easily and will he or she therefore be able to maintain said code effortlessly?

-1 No (Difficult to read and understand)

0 Partially (Readable and understandable after a short time of research)

+1 Yes

Support

Number of Answered Questions on StackOverflow.com Are there many questions about the framework that have been marked as answered on StackOverflow.com?

-1 No (Less than 25)

0 Between

+1 Yes (More than 100)

Number of Contributors on GitHub Have many different people worked on the framework?

-1 No (Less than 3)

0 Between

+1 Yes (More than 10)

Number of Stars on GitHub Did many people show interest by adding their star to the solution?

-1 No (Less than 50)

0 Between

+1 Yes (More than 300)

Google Query about Typical Question Does a typical query about the framework on Google return the official documentation?

-1 No

0 Partially (Not within top three results)

+1 Yes

Quality of Documentation Do the authors of the framework maintain a helpful documentation?

-1 No

0 Partially (Outdated or too shallow)

+1 Yes

Reliability

Architectural Cleanliness Is the architecture employed by the framework well defined and beneficial?

-1 No

0 Partially

+1 Yes

Implementation Cleanliness Is code inside the framework easy to understand?

-1 No

0 Partially

+1 Yes

Maturity Was the first version of the framework released long enough ago to be considered mature?

-1 No (Less than six months ago)

0 Between

+1 Yes (More than one year ago)

Development Activity Do people work regularly on the solution?

-1 No (Less than five commits on average per week during the last six months)

0 Between

+1 Yes (More than ten commits on average per week during the last six months)

Test Coverage Is the framework tested well?

-1 No

0 Partially (Unsteady execution or low coverage)

+1 Yes

Version Do the developers feel confident about the quality of their product?

-1 No (Alpha or Beta version)

0 Between

+1 Yes (First major version or above)

A.1.2. Evaluation

The criteria and their scoring used for this comparison of view data binding frameworks is explained in detail in section A.1.1. In general a positive score denotes a good, a negative score a bad and a neutral score a neither good nor bad result. The number below the names of the frameworks in the second line of each table represents the version of the framework, which was tested.

Design

Table A.1.: Comparison of View Data Binding Frameworks — Design

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Logic in View	+1	+1	-1	-1	-1	0	+1	-1
Binding Separable	-1	-1	-1	-1	-1	0	-1	-1
Surgical Updates	+1	-1	0	+1	+1	0	-1	+1
Intermediate Representation	0	-1	-1	-1	+1	0	-1	0
Paradigm	-1	-1	-1	-1	-1	-1	-1	-1
Limitation of Scope	0	+1	+1	+1	0	-1	+1	-1
Template Sources	0	-1	0	0	+1	-1	0	+1
Total Score	**0**	**-3**	**-3**	**-2**	**0**	**-3**	**-2**	**-2**

Compatibility

Table A.2.: Comparison of View Data Binding Frameworks — Compatibility

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Hierarchical Model	-1	+1	-1	-1	-1	-1	0	-1
Browser	+1	0	0	0	0	0	0	+1
jQuery	+1	+1	+1	+1	+1	0	+1	+1
Extensibility	0	+1	+1	0	-1	0	0	+1
Observation Mechanism	-1	+1	-1	-1	-1	-1	+1	-1
Total Score	**0**	**+4**	**0**	**-1**	**-2**	**-2**	**+2**	**+1**

Features

Table A.3.: Comparison of View Data Binding Frameworks — Features

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Animations	+1	-1	-1	+1	0	-1	-1	+1
DOM Event Binding	0	0	0	0	0	0	0	0
Chains	0	-1	+1	+1	0	+1	-1	0
Converter	+1	+1	+1	+1	0	+1	0	+1
Expressions	+1	-1	+1	0	+1	+1	0	+1
Filters	+1	-1	0	0	0	+1	-1	+1
Subscription to Internal Events	+1	0	+1	0	+1	+1	0	+1
Two-way Binding	+1	+1	+1	+1	0	+1	-1	+1
Validation	0	-1	0	0	+1	+1	-1	+1
Total Score	**+6**	**-3**	**+4**	**+4**	**+3**	**+6**	**-5**	**+7**

Usability

Table A.4.: Comparison of View Data Binding Frameworks — Usability

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Existing Solution	0	0	-1	0	-1	-1	+1	0
Big Applications	+1	0	0	0	+1	0	-1	+1
Learning Curve	+1	+1	+1	+1	0	-1	+1	0
Dependencies	+1	+1	+1	+1	+1	+1	+1	+1
Size	0	+1	+1	+1	0	0	0	+1
Syntax	+1	+1	+1	0	-1	-1	+1	0
Total Score	**+4**	**+4**	**+3**	**+3**	**0**	**-2**	**+3**	**+3**

Support

Table A.5.: Comparison of View Data Binding Frameworks — Support

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Stackoverflow	0	0	-1	-1	+1	-1	-1	+1
Contributors GitHub	+1	+1	0	0	+1	+1	+1	+1
Stars GitHub	+1	+1	+1	+1	+1	+1	0	+1
Google	0	-1	+1	+1	0	+1	-1	+1
Documentation	0	-1	+1	0	+1	-1	-1	+1
Total Score	**+2**	**0**	**+2**	**+1**	**+4**	**+1**	**-2**	**+5**

Reliability

Table A.6.: Comparison of View Data Binding Frameworks — Reliability

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Architecture	-1	0	0	0	+1	-1	0	+1
Implementation	-1	+1	+1	0	+1	-1	0	+1
Maturity	0	+1	-1	-1	0	+1	+1	+1
Development Activity	+1	-1	0	+1	+1	0	-1	-1
Test Coverage	0	0	+1	+1	+1	0	0	+1
Version	-1	-1	-1	-1	-1	-1	+1	+1
Total Score	**-2**	**0**	**0**	**0**	**+3**	**-2**	**+1**	**+4**

Accumulated Scores

Table A.7.: Comparison of View Data Binding Frameworks — Accumulated Scores

	Ractive.js *0.4.0*	**RivetJS** *0.6.7*	**Ripple** *0.3.5*	**Vue** *0.10.4*	**FB React** *0.10.0*	**MontageJS** *0.14.2*	**Reactive** *1.1.0*	**KnockoutJS** *3.1.0*
Design	0	-3	-3	-2	0	-3	-2	-2
Compatibility	0	+4	0	-1	-2	-2	+2	+1
Features	+6	-3	+4	+4	+3	+6	-5	+7
Usability	+4	+4	+3	+3	0	-2	+3	+3
Support	+2	0	+2	+1	+4	+1	-2	+5
Reliability	-2	0	0	0	+3	-2	+1	+4
Accumulated Score	**+10**	**+2**	**+6**	**+5**	**+8**	**-2**	**-3**	**+18**

A.1.3. List of Libraries

This list contains most frameworks for view data binding written in JavaScript in alphabetic order. The solutions either target the problem exclusively or offer view data binding as a part of their broader functionality.

AngularJS
https://angularjs.org/

Backbone Bindem
http://dundalek.com/bindem/

Backbone Epoxy
http://epoxyjs.org/

BaconJS
https://github.com/baconjs/bacon.js

Batman Bindings
http://batmanjs.org/docs/api/batman.view_bindings.html

CanJS
http://canjs.com/

data-bind.lite
https://github.com/jhewlett/data-bind.lite

Derby
http://derbyjs.com/

dust.js
http://akdubya.github.io/dustjs/

Ember
http://emberjs.com/

ExtJS
http://www.sencha.com/products/extjs/

Facebook React
http://facebook.github.io/react/

Generic Data Binder
http://gdb.thewebdev.guru/

jQuery jBinder
https://github.com/Steve166/jBinder

jQuery.my
https://github.com/ermouth/jQuery.my

jQXB Expression Binder
http://jqxb.codeplex.com/

jsViews
https://github.com/BorisMoore/jsviews

KnockoutJS
http://knockoutjs.com/

Lava JS
http://lava.codeplex.com/

Meteor
https://www.meteor.com/

MontageJS / FRB
https://github.com/montagejs/frb

Ractive.js
http://www.ractivejs.org/

Reactive
https://github.com/component/reactive

riot.js
https://muut.com/riotjs/

Ripple
https://github.com/ripplejs/ripple

RiverJS
http://besideriver.com/RiverJS/

RivetJS
http://www.rivetsjs.com/

RxJS
http://reactive-extensions.github.io/RxJS/

Serenade
http://serenadejs.org/binding_data.html

Simpli5
https://github.com/jacwright/simpli5

Synapse
http://bruth.github.io/synapse/docs/

Vue
http://vuejs.org

A.2. Code Listings

```
that.$todoList.innerHTML = show(parameter);

function show(data) {
  var i, l;
  var view = '';

  for (i = 0, l = data.length; i < l; i++) {
    var template = this.defaultTemplate;
    var completed = '';
    var checked = '';

    if (data[i].completed) {
      completed = 'completed';
      checked = 'checked';
    }

    template = template.replace('{{id}}', data[i].id);
    template = template.replace('{{title}}', data[i].title);
    template = template.replace('{{completed}}', completed);
    template = template.replace('{{checked}}', checked);

    view = view + template;
  }

  return view;
};
```

Listing A.1: View Rendering Written in JavaScript (TodoMVC, Excerpt)

```
#container {
  @buttonActive <~ false
  input#textbox {
    @buttonActive <- value.length > 0
  }
  button#submit {
    attr:disabled <- !@buttonActive
  }
}
```

Listing A.2: Realizing Dependant Elements with the *Binding Scope*

```
var $ = class {
  constructor () { this.observer = {}; this.counter = new Counter() }

  notify (model, path) {
    foreach (observer in this.observer) {
      if (observer.model == model && observer.path == path) {
        observer.callback()
      }
    }
  }

  observe (model, path, callback) {
    var ref = model
    foreach (string in path.subList(0, path.length - 1)) {
      ref = ref[string]
    }
    var id = this.counter.getNext()
    this.observer.push({ model: model, path: path, ref: ref,
                         callback: callback, id: id })
    if (!alreadyObserved(ref, path[path.length - 1]) {
      var self = this
      WatchJS.watch(ref, path[path.length - 1],
                    function () { self.notify(model, path) })
    }
    return id
  }

  unobserve (observerId) {
    var observer = this.observer.findById(observerId)
    if (noOtherObserver(observer.model, observer.ref)) {
      WatchJS.unwatch(observer.model, observer.ref)
    }
    this.observer.remove(observer)
  }

  getValue (model, path) {
    var ref = model
    foreach (string in path) {
      ref = ref[string]
    }
    return ref
  }

  getPaths (model, path) {
    var result = [path]
    var value = this.getValue(model, path)
    if (!isPrimitive(value)) {
      foreach (key in value) {
```

```
49          var newPath = path.clone()
50          newPath.push(key)
51          var subPaths = this.getPaths(model, newPath)
52          foreach (subPath in subPaths) {
53            result.push(subPath)
54          }
55        }
56      }
57      return result
58    }
59
60    set (model, path, value) {
61      let ref = model
62      foreach (string in path.subList(0, path.length - 1)) {
63        elem = elem[string]
64      }
65      elem[path[path.length - 1]] = value
66    }
67
68    type () {
69      return "model"
70    }
71  }
72
73  BindingJS.plugin("$", new $())
```

Listing A.3: *Model Adapter* for JSON *Presentation Model*

```
1   function nestIteratedBindings(ast) {
2     var elementToScopesMap = {}
3     foreach (scope in  ast.getAllScopes()) {
4       elementToScopesMap[scope.element].add(scope)
5     }
6
7     foreach (element, scopes in elementToScopeMap) {
8       if (anyOf(scopes).hasIteratedAncestorScope()) {
9         foreach (otherScope in scopes.without(iteratedScope)) {
10          if (!otherScope.isDescendantOf(iteratedScope)) {
11            otherScope.moveInto(iteratedScope)
12          }
13        }
14      }
15    }
16
17    foreach (scope in ast.getEmptyScopes()) {
18      scope.remove()
19    }
20  }
```

Listing A.4: Step 8: Nesting *Bindings* of *Scopes* Affected by *Iterations*

```
var Value = class {
  constructor () { this.observer = {}; this.counter = new Counter() }
  notify (element) {
    foreach (observer in this.observer) {
      if (observer.element == element) {
        observer.callback()
  }}}
  observe (element, path, callback) {
    var id = this.counter.getNext()
    this.observer.push({ element: element, callback: callback,
                         id: id })
    if (!alreadyObserved(element)) {
      var self = this
      element.on("change", function () { self.notify(element) })
    }
    return id
  }
  unobserve (observerId) {
    var observer = this.observer.findById(observerId)
    if (noOtherObserver(observer.element)) {
      observer.element.off("change")
    }
    this.observer.remove(observer)
  }
  getValue (element, path) {
    return element.val()
  }
  getPaths (element, path) {
    return [path]
  }
  set (element, path, value) {
    let oldValue = element.val()
    element.val(value)
    if (value !== oldValue) {
      element.trigger("change")
  }}
  type () {
    return "view"
  }
}
BindingJS.plugin("value", new Value())
```

Listing A.5: *View Adapter* Reading and Writing Values of HTML Elements

```
1  // bindingspecification consists of blocks (alias b) followed by eof
2  // blocks is defined in line 8, eof in line 44
3  bindingspecification =
4    b:blocks eof {
5      // Result of parsing a bindingspecification is result of blocks
6      return b
7    }
8
9  // blocks consist of an arbitrary number of block elements
10 blocks =
11   b:(_ block)* _ {
12     // b comprises a set of matches for each _ and block
13     // result is an AST with root "Blocks" and one child for each block
14     return AST("Blocks").add(getFromEach(b).index(1))
15   }
16
17 block
18   = group
19   / scope
20
21 group =
22   "@binding" ws n:id ws "{" b:blocks "}" {
23     return AST("Group").set("id").to(n.get("id")).add(b)
24   }
25
26 scope =
27   s:selectors _ i:scopeIterator? _ e:scopeExport? _
28     "{" b:(_ scopeBody _ ";"?)* _ "}" {
29       return AST("Scope").add(s, i, e, getFromEach(b).inddex(1)))
30   }
31   /
32   s:selectors _ x:(scopeLabel / scopeIterator /
33                    scopeImport / scopeExport) {
34     return AST("Scope").add(s, x)
35   }
36
37 _ "optional blank" = (co / ws)*
38
39 co "end-of-line or multi-line comment"
40   =   "//" [^\r\n]*
41   /   "/*" (!"*/" .)* "*/"
42
43 ws "any whitespaces" = [ \t\r\n]+
44
45 eof "end of file" = !.
```

Listing A.6: *PEG.js* Grammar (Excerpt)

```
function checkIterationIds(ast) {
  checkIterationIdsRec(ast, [])
}

function checkIterationIdsRec(ast, ids) {
  if (isIteratedScope(ast)) {
    var entryAdapter = ast.getEntryAdapter()
    if (!isBindingScopeAdapter(entryAdapter)) {
          throw exception("You can only use the binding scope " +
                  "adapter as the entryAdapter for an iteration. " +
                  "Instead " + entryAdapter + "was used")
    }
    var keyAdapter = ast.getKeyAdapter()
    if (!isBindingScopeAdapter(keyAdapter)) {
      // See above
    }

    if (ids.contains(entryAdapter.getQualifier())) {
      throw exception("Adapter " + entryAdapter.getText() +
              " was used as an iteration variable," +
              " but was also declared in an ancestor scope"
    }
    if (ids.contains(keyAdapter.getQualifier())) {
      // See above
    }
  }

  // ids must be passed by value to recursive calls
  ids = ids.clone()
  // Add all direct Adapter to ids
  var allAdapter = ast.getAllAdapter()
  foreach (adapter in allAdapter) {
    if (isBindingScopeAdapter(adapter)) {
        ids.add(adapter.getQualifier())
    }
  }

  // Recursion
  foreach (child in ast.getChildren()) {
    checkIterationIdsRec(child, ids)
  }
}
```

Listing A.7: Step 1: Checking *Adapter* of *Iterations*

```
1  function expandSelectors(template, ast) {
2    expandSelectorsRec(template, ast)
3
4    // Remove placeholders (see line 27)
5    ast.removeAll("Placeholder")
6  }}
7
8  function expandSelectorsRec(template, ast) {
9    if (isScope(ast)) {
10       var newScopes = []
11       foreach (selector in ast.selectorList) {
12         // Evaluate Selector
13         var elements = jQuery(selector, template)
14         foreach (element in elements) {
15           // Clone Scope including all child Scopes
16           var newScope = ast.clone()
17           newScope.element = element
18           newScopes.push(newScope)
19       }}
20       ast.removeSelectors()
21
22       if (!newScopes.isEmpty()) {
23         ast.replace(newScopes)
24       } else {
25         // Replace with Placeholder to avoid changing collection
26         // of for loop below
27         ast.replace(new AST("Placeholder"))
28       }
29
30       // Recursion over every newly generated scope
31       foreach (newScope in newScopes) {
32         foreach (child in newScope.children) {
33           // Recursion, changes template to matched element of parent
34           expandSelectorsRec(newScope.element, child)
35       }}
36   } else {
37     // Recursion
38     foreach (child in ast.children) {
39       expandSelectorsRec(template, child)
40 }}}
```

Listing A.8: Step 3: Expanding Selectors

```
Scope [2/3] // div
  Binding [3/5]
    Adapter [3/5]
      ExprSeq [3/5]
        Variable (ns: "@", id: "temp") [3/5]
  Scope [4/5] // span
    Binding [5/7]
      Adapter [5/7]
        ExprSeq [5/7]
          Variable (ns: "@", id: "entry") [5/7]
  Scope [8/5] // li
    Iterator [8/8]
      Variables [8/8]
        Variable (ns: "@", id: "entry") [8/9]
        Variable (ns: "@", id: "temp") [8/17]
      Expr [8/8]
        Variable (ns: "$", id: "collection") [8/24]
```

Listing A.9: Step 1: Counter Example Abstract Syntax Tree Excerpt

```
function shutdown(plItRoot) {
  foreach (child in plItRoot.getChildren()) {
    var expItNode = child.links[0]
    bindingScope.unobserve(expItNode.sourceObserverId)
    var collection = bindingScope.get(expItNode.sourceId)
    if (isReference(collection)) {
      collection = collection.getValue()
    }
    collection = collection ? collection : []
    var newCollection
    if (isBoolean(collection)) {
      newCollection = false
    } else {
      newCollection = []
    }
    bindingScope.set(expItNode.sourceId, newCollection)
    changeListener(expItNode)
  }
  var rootInstance = plItRoot.links[0].instances[0]
  shutdownBinding(rootInstance)
}
```

Listing A.10: Deactivating the *View Data Binding*

```
1  function setupIterationTree(ast, template) {
2    var node = new PlainIteration()
3
4    if (ast.isIteratedScope()) {
5      node.entryId = ast.entryId
6      node.keyId = ast.keyId
7      node.sourceId = ast.sourceId
8      // div (...) { ... } => div { ... }
9      ast.removeIteration()
10
11     var placeholder = createNode("<!-- -->")
12     template.insertAfter(placeholder)
13     template.detach()
14
15     node.iterationTemplate = template
16     node.template = placeholder
17     node.collection = []
18   } else {
19     node.template = template
20     node.iterationTemplate = template
21     node.collection = true
22   }
23
24   node.placeholder = []
25   foreach (scope in ast.getDirectChildIteratedScopes()) {
26     // Recursion, changes template
27     var child = setupIterationTree(scope, scope.element)
28     node.placeholder.push(child.template)
29     node.addChild(child)
30     // This does not affect the childs binding (see line 33)
31     scope.remove()
32   }
33   node.binding = ast.clone()
34   node.links = []
35
36   return node
37 }
```

Listing A.11: Step 9: Setup *Iteration Tree*

```
function setupExpandedIterationTree(plainIterationRoot) {
  var expandedIterationRoot =
    initExpandedIterationNode(plainIterationRoot, null)
  var rootInstance = initInstance(expandedIterationRoot)

  expandedIterationRoot.instances = [rootInstance]
  plainIterationRoot.links = [expandedIterationRoot]

  foreach (plainIterationChild in plainIterationRoot.children) {
    var expandedIterationChild =
      initExpandedIterationNode(plainIterationChild,
                                expandedIterationRoot)
    expandedIterationRoot.add(expandedIterationChild)
    expandedIterationChild.instance = rootInstance
    plainIterationChild.links = [expandedIterationChild]
  }

  return expandedIterationRoot
}

function initExpandedIterationNode(plainIterationNode, parentLink,
                                   bindingScopePrefix, counter) {
  var result = new ExpandedIteration()
  result.template = plainIterationTemplate.clone()
  result.instances = []
  result.sourceId = plainIterationNode.sourceId
  result.origin = plainIterationNode

  // Adapt references to clone of template
  result.binding = plainIterationNode.binding.clone()
  result.placeholder = plainIterationNode.placeholder.clone()
  result.sockets = plainIterationNode.sockets.clone()

  if (plainIterationNode.isRoot()) {
    result.placeholderIndex = -1
    result.collection = true
  } else {
    result.placeholderIndex = plainIterationNode.childIndex() :
    result.collection = []
  }

  // Renaming
  var bindingScopeAdapter = result.binding.getAllBindingScopeAdapter()

  // Rename what parent renamed ...
  foreach (adapter in bindingScopeAdapter) {
    if (parentLinkHasRenamed(adapter)) {
      adapter.renameLikeParentLink()
```

```
49    }}
50    // ... including own source id
51    if (parentLink.renamed(result.sourceId) {
52      result.sourceId = parentLink.findHowRenamed(result.sourceId)
53    }
54
55    // Find all newly introduced Binding Scope Qualifiers
56    var ownVariables = []
57    foreach (adapter in bindingScopeAdapter) {
58      // Check if Qualifier was used in any ancestor
59      if (noAncestorKnowsAbout(adapter)) {
60        ownVariables.add(adapter)
61    }}
62    // Entry and Key Adapter are always own variables because of step 1
63    if (plainIterationNode.entryId) {
64      ownVariables.add(plainIterationNode.entryId)
65    }
66    if (plainIterationNode.keyId) {
67      ownVariables.add(plainIterationNode.keyId)
68    }
69
70    var ownRenames = []
71    ownRenames.addAll( renameAll(ownVariables, result) )
72    ownRenames.addAll( renameEntryAndKey(result) )
73    result.bindingRenames = parent.bindingRenames + ownRenames
74
75    return result
76  }
```

Listing A.12: Step 11: Setup Expanded *Iteration* Tree

```
1   function init(expandedIterationTreeRoot) {
2     var iterationInstanceRoot = expandedIterationTreeRoot.instances[0]
3     initBinding(iterationInstanceRoot)
4     foreach (plainIterationTreeChild in
5               expandedIterationTreeRoot.origin.children) {
6       var expandedIterationTreeChild = plainIterationTreeChild.links[0]
7       var observerId =
8         bindingScope.observe(expandedIterationTreeChild, function () {
9           changeListener(expandedIterationTreeChild)
10        })
11      expandedIterationTreeChild.sourceObserverId = observerId
12    }
13    changeListener(expandedIterationTreeChild)
14  }
```

Listing A.13: Activating the *View Data Binding*

```
function levenshtein(oldCollection, newCollection) {
  var newValues = []
  foreach (item, key in newCollection)
    newValues[key] = convertToValues(item)

  var result = []
  if (isArray(oldCollection) && isArray(newCollection)) {
    // Comparing Arrays
    // See Levenshtein Distance Implementations
    var matrix = [];
    for (var i = 0; i <= newCollection.length; i++)
      matrix[i] = [i]
    for (var j = 0; j <= oldCollection.length; j++)
      matrix[0][j] = j

    for (var i = 1; i <= newCollection.length; i++) {
      for (var j = 1; j <= oldCollection.length; j++) {
        if (objectEquals(newValues[i-1], oldCollection[j-1])) {
          matrix[i][j] = matrix[i-1][j-1]
        } else {
          matrix[i][j] = Math.min(matrix[i-1][j-1] + 1, // substitution
                                  matrix[i  ][j-1] + 1,   // insertion
                                  matrix[i-1][j  ] + 1)   // deletion
    }}}

    // See stackoverflow.com
    var x = newCollection.length
    var y = oldCollection.length
    var result = [];
    while (x >= 0 && y >= 0) {
      var current = matrix[x][y];
      var diagonal   = x - 1 >= 0 && y - 1 >= 0 ?
                       matrix[x - 1][y - 1] :
                       Number.MAX_VALUE
      var vertical   = x - 1 >= 0 ?
                       matrix[x - 1][y] :
                       Number.MAX_VALUE
      var horizontal = y - 1 >= 0 ?
                       matrix[x][y - 1] :
                       Number.MAX_VALUE

      if (diagonal <= Math.min(horizontal, vertical)) {
        x = x - 1
        y = y - 1
        if (diagonal + 1 == current) {
          result.add({ action:   "replace",
                       key:      y,
                       newValue: newCollection[x] })
```

```
49          } else {
50            // Nothing has to be done, elements were equal
51          }
52        } else if ((horizontal <= vertical && horizontal == current)
53                    || horizontal + 1 == current) {
54          y = y - 1
55          result.add({ action: "remove",
56                       key:    y })
57        } else {
58          x = x - 1
59          result.add({ action:   "add",
60                       key:      x,
61                       value:    newCollection[x],
62                       afterKey: y - 1 })
63        }
64      }
65    } else {
66      // Comparing Objects
67      foreach (key in oldCollection) {
68        if (!newCollection.contains(key)) {
69          result.add({ action: "remove", key: key })
70        } else if (!objectEquals(oldCollection[key], newCollection[key])){
71          result.add({ action: "replace", key: key,
72                       newValue: newCollection[key] })
73        }
74      }
75      var last
76      foreach (key in newCollection) {
77        result.add({ action: "add", key: key,
78                     value: newCollection[key], afterKey: last })
79        last = key
80      }
81    }
82    return result
83  }
```

Listing A.14: Levenshtein Algorithm to Determine Minimum Number of Changes

```
function add(expItNode, key, value, afterKey) {
  var newInstance = addInstance(expItNode, key, value, afterKey)
  foreach (child in expItNode.origin.getChildren()) {
    var newChild = initChild(expItNode, child, newInstance, key)
    expItNode.add(newChild)
  }
  initBinding(newInstance)
  refreshKeysAdded(expItNode, newInstance, afterKey)
}

function addInstance(expItNode, key, value, afterKey) {
  var newTemplate = expItNode.template.clone()
  // Adapt references to clone of template
  var newPlaceholders = expItNode.placeholder.clone()
  var newSockets = expItNode.sockets.clone()
  var newBinding = expItNode.binding.clone()

  // Renaming
  var bindingRenames = []
  bindingRenames.addAll(
    renameAllOwnVariables(expItNode.ownVariables, counter)
  )
  var entryIdRename = renameEntryId(expItNode.entryId, counter)
  var newEntryId = entryIdRename.newId

  var keyIdRename = renameKeyId(expItNode.keyId, counter)
  var newKeyId = keyIdRename.newId

  bindingRenames.addAll(entryIdRename, keyIdRename)
  bindingRenames.addAllIfExists(expItNode.instance.bindingRenames)
  doRename(newBinding, bindingRenames)

  // Inject entry and key
  if (entryIdRename) {
    bindingScope.set(entryIdRename.newId, value)
  }
  if (keyIdRename) {
    bindingScope.set(keyIdRename.newId, key)
  }

  // Insert template
  if (afterKey && expItNode.instances.length > 0) {
    var afterInstance = findInstanceByKey(expItNode.instances, afterKey)
    afterInstance.template.after(newTemplate)
  } else {
    expItNode.instance.placeholder[expItNode.placeholderIndex]
      .after(newTemplate)
  }
```

```
49
50    var newInstance = {
51      entryId: newEntryId,
52      keyId: newKeyId,
53      key: key,
54      template: newTemplate,
55      binding: newBinding,
56      placeholder: newPlaceholders,
57      sockets: newSockets,
58      bindingRenames: bindingRenames
59    }
60    expItNode.instances.add(newInstance)
61
62    // Call socket observer
63    callSocketInsertionObserverInstance(expItNode, newInstance)
64
65    return newInstance
66  }
67
68  function initChild(expItNode, plItNode, instance, key) {
69    var result = initExpandedIterationNode(plItNode, expItNode, ...)
70    plItNode.links.add(result)
71    result.instance = instance
72    if (instance.bindingRenames.renamed(result.sourceId)) {
73      result.sourceId = instance.bindingRenames.get(result.sourceId)
74    }
75
76    var sourceObserverId =
77      bindingScope.observe(result.sourceId, function () {
78        changeListener(result)
79      })
80    changeListener(result)
81    result.sourceObserverId = sourceObserverId
82    result.key = key
83
84    return result
85  }
```

Listing A.15: Adding *Iteration Instances*

```
function remove(expItNode, key) {
  var instance = findByKey(expItNode.instances, key)
  shutdownBinding(instance)

  foreach (child in expItNode.getChildren()) {
    if (child.instance == instance) {
      expItNode.remove(child)
      destroyChild(child)
    }
  }

  removeInstance(expItNode, key, instance)
  refreshKeysRemoved(expItNode, key)
}

function destroyChild(expItNode) {
  bindingScope.unobserve(expItNode.sourceObserverId)
  var collection = bindingScope.get(expItNode.sourceId)
  if (isReference(collection)) {
    collection = collection.getValue()
  }
  collection = collection ? collection : []

  var newCollection
  if (isBoolean(collection)) {
    newCollection = false
  } else {
    newCollection = []
  }
  bindingScope.set(expItNode.sourceId, newCollection)
  changeListener(expItNode)

  expItNode.origin.links.remove(expItNode)
}

function removeInstance(expItNode, key, instance) {
  callSocketRemovalObserverInstance(expItNode, instance)
  expItNode.instances.remove(instance)
  instance.template.detach()

  if (instance.entryId) {
    bindingScope.destroy(instance.entryId)
  }
  if (instance.keyId) {
    bindingScope.destroy(instance.keyId)
}}
```

Listing A.16: Removing *Iteration Instances*

A.3. Figures

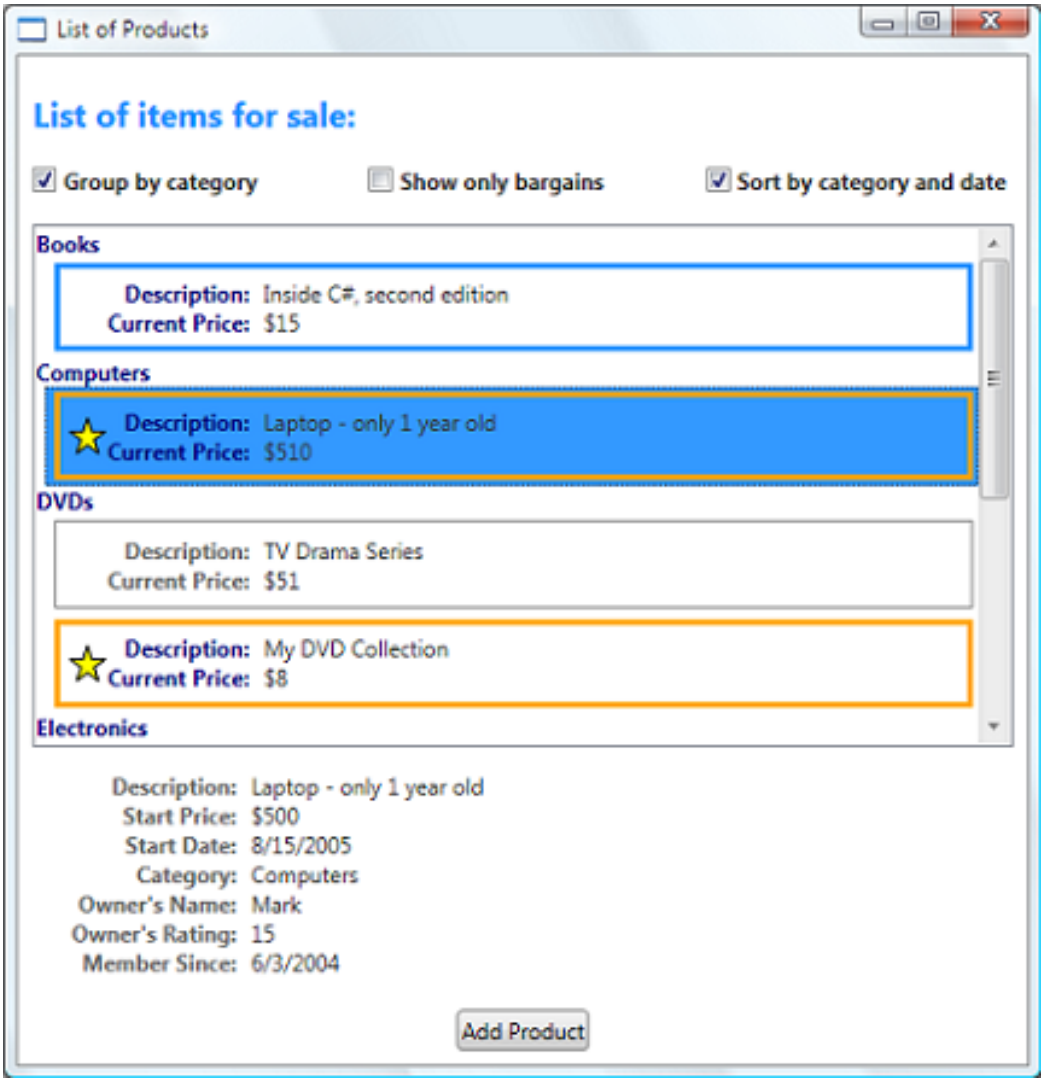


Figure A.1.: WPF Application Demonstrating Data Binding

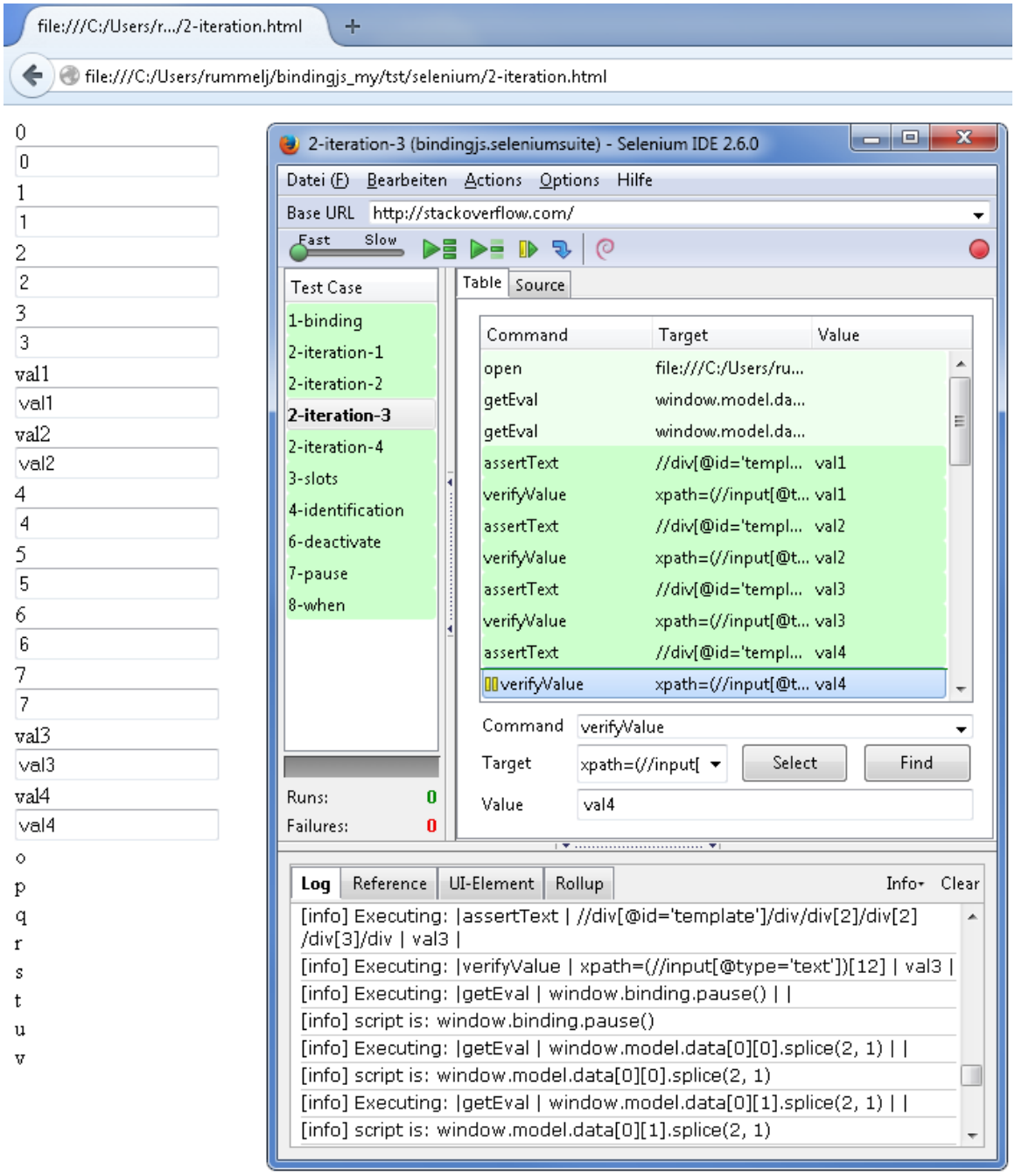


Figure A.2.: Selenium IDE Test Case Execution

B. Glossary

Adobe Flash Propietary software platform for programming and displaying interactive multimedia content. 1

AJAX Concept that describes how data is asynchronously transferred between a web browser and a server. It therefore allows to populate a web page with data from the server without reloading it. 23, 24, 181

AngularJS Open source JavaScript framework for creating browser-based single page applications following the Model-View-ViewModel pattern (see section 2.1.2). 1

API See Application Programming Interface. 8, 21, 32, 87, 88, 92–94, 97–99, 101, 102, 105, 115, 130, 131, 144

Application Programming Interface Specifies the possible ways of interacting with a software component. 181

Backbone.js Open source JavaScript library for creating browser based single page applications following the Model-View-Controller pattern (see section 2.1.1). 1

Big Ball of Mud Software without recognizable architecture. 36

Cascading Style Sheets Declarative language to define the appearance of web page elements (see [Lie94]). 181

Chrome Widely used web browser, that focuses on performance, reliability and security. 148, 181

Chrome OS Linux based operating system (OS) that makes minimal use of local resources and is primarily designed to run the web browser Chrome. 2

ComponentJS JavaScript library for hierarchically structuring the architecture of web applications. 2, 10, 11, 14, 19, 64, 148

CSS See Cascading Style Sheets. 1, 41, 58, 60, 102, 139

Dart Web programming language that aims to provide an alternative to and in the future replace JavaScript[1]. 1

Document Object Model Specification of an interface for accessing or manipulating HTML or XML documents. 182

dojo Open source library for writing JavaScript/AJAX based applications and web sites including support for language localizations and reusable user interface elements. 1, 2

[1] https://www.dartlang.org/

DOM See Document Object Model. 27, 28, 30, 32, 88, 90, 133, 147–149, 155, 183

double mustache notation Type of notation that distinguishes elements that are embedded into a host language by enclosing them in two curly brackets, like `{{expression}}`. The term is used, since curly brackets rotated by 90 degrees resemble a mustache. 14, 29, 32

Extensible Application Markup Language Language for describing the structure and appearance of user interfaces or workflows developed by Microsoft. 184

Extensible Markup Language Language for specifying hierarchically structured data. 184

facebook Internet application, regularly referred to as a social network that allows sharing and consuming information with and from other people[2]. 1, 27

Firefox Widely used open source web browser that offers a rich set of privacy and security measures. 148

GitHub Web-based service for hosting software projects and making their development visible to collaborators and interested people named after the revision control system Git. 91, 135, 151, 157

Google Docs Internet application for collaboratively creating and manipulating documents[3]. 1

Google Hangouts Internet application for holding online meetings including text-, video- and file-based communication[4]. 1

Google Maps Internet application for browsing the world map including satellite imagery, photos from street level, and navigation[5]. 1

Graphical User Interface Type of interface that in contrast to text based command line interfaces adds graphical elements that enable a user to interact with the application. 182, 183

GUI See Graphical User Interface. 2, 8

gzip Free application for compressing files that is available for virtually any operating system. 32

HTML See Hypertext Markup Language. 2, 8, 11, 19, 24, 30, 31, 36, 39, 44, 56, 60, 76, 87, 95, 101, 102, 142, 144, 145, 165, 181–183

HTML 5 Fifth revision of the HTML standard mainly adding multimedia functionality. 1, 8, 131

Hyperlink Usually a clickable reference to another web page inside a web page. 6

[2] http://www.facebook.com/
[3] http://docs.google.com/
[4] http://hangouts.google.com/
[5] http://maps.google.com/

Hypertext Markup Language Standard language to create web pages[6]. 182

i18n Numeronym (abbreviation) for *Internationalization* which is the task in software engineering making an application available in different languages. 47

IDE See Integrated Development Environment. 21

Integrated Development Environment Software that aids in developing software usually comprising a source code editor, build automation and support for debugging. 183

Internet Explorer Web browser for Windows operating systems. 148

Java Object oriented, imperative programming language that is, due to its execution in a virtual environment, platform independent. 21, 24, 183

Java EE Enterprise edition (EE) of Java specifying the architecture for a transaction based middleware solution that is primarily used in web applications. 6

JavaScript Web programming and scripting language that is executed mainly in web browsers to enrich web pages. 1, 6, 8, 11–13, 27, 29, 30, 32, 42, 52, 54, 87, 88, 90, 92, 105, 106, 131, 133, 137, 142, 144, 147, 160, 162, 181, 183, 184

JavaScript Object Notation Compact data format to transfer data between or inside applications that is defined with JavaScript, but is principally language independent with parsers in all dominant languages. 183

jQuery A client-side JavaScript library simplifying the manipulation of and interaction with HTML. 1, 88, 90, 92, 95, 102, 131, 134, 148, 154, 183

jQuery Markup jQuery plugin for populating and injecting templates into the DOM. 14, 16

jQuery UI Extension to jQuery that comprises solutions for designing and defining the functionality of Graphical User Interfaces. 2

JSON See JavaScript Object Notation. 7, 11, 39, 44–46, 48, 50, 87, 92, 101, 104, 126, 127, 129, 164

kB See kilobyte. 32, 33, 134, 150

kilobyte Unit for measuring amounts of data. For example to store 256 or 512 characters depending on the encoding, one kilobyte is required. 183

lint Tool that performs static analysis of source code to give developers hints about potentially error-prone passages. 90

Microsoft Developer Network Collection of software and documentation for Microsoft products and technology. 184

[6] http://www.w3.org/community/webed/wiki/HTML/Specifications

Microsoft Excel Spreadsheet application that is part of Microsoft Office featuring calculations, graphing tools, pivot tables and macro programming. 16

Microsoft Silverlight Extension for web browsers that allows the execution of rich internet applications. 1

minification The process of converting source code to make it as small as possible without changing its semantics. This is useful for interpreted languages, like JavaScript, where the source code has to be transfered over a network. 32, 90

Model-View-Presenter Design pattern that emerged from the Model-View-Controller pattern (see section 2.1.1) that aims to decouple model from view by connecting them with a presenter. 184

MSDN See Microsoft Developer Network. 24

msg systems AG German IT service provider based in Ismaning near Munich whose offer covers consulting, application development, and system integration. 11, 14, 142, 145

MVP See Model-View-Presenter. 6

Parsing Expression Grammar Formal grammar similar to context-free grammars. Its main difference is that it cannot produce an ambigious parse tree. 39, 77, 80, 106

StackOverflow.com Open internet platform that allows people to exchange experience with software development. 135, 151

Unified Modeling Language Graphical language to specify, construct and document software and system components[7]. 36, 92

Unix Timestamp Method for storing time information by saving the number of seconds or milliseconds elapsed since midnight, 1 January 1970. For example, `11:01:21 am, 27 June 2014` would be stored as `1403859681` seconds. 26

XAML See Extensible Application Markup Language. 24

XML See Extensible Markup Language. 24, 181

YouTube Open online video platform for sharing and consuming multimedia recordings[8]. 1

[7] http://www.uml.org/
[8] http://www.youtube.com/

C. Bibliography

[Bai] Derick Bailey. *Backbone.js Is Not An MVC Framework*.
URL: http://lostechies.com/derickbailey/2011/12/23/backbone-js-is-not-an-mvc-framework/ (visited on 06/06/2014).

[Bew] Chris Bewick. *HTML5 Custom Data Attributes (data-*)*.
URL: http://html5doctor.com/html5-custom-data-attributes/ (visited on 06/10/2014).

[Bro+03] Kyle Brown et al. *Enterprise Java Programming with IBM WebSphere*. 2nd ed. IBM Press, 2003.

[Cor] Robert Corvus. *Separation of Concerns Principle*.
URL: http://robertcorvus.com/separation-of-concerns-principle/ (visited on 06/13/2014).

[Cou87] Joëlle Coutaz. „PAC: an Implementation Model for Dialog Design“. In: *Proceedings of the Interact'87 conference*. 1987, pp. 431–436.

[Enga] Ralf S. Engelschall. *7-Layer Application Partitioning Architecture*.
URL: http://engelschall.com/go/EnTR-01:2014.01 (visited on 06/06/2014).

[Engb] Ralf S. Engelschall. *ComponentJS Architecture*.
URL: http://componentjs.com/architecture.html (visited on 06/10/2014).

[Engc] Ralf S. Engelschall. *User Interface Component Architecture*.
URL: http://componentjs.com/architecture/ui-component-architecture.pdf (visited on 06/10/2014).

[Engd] Ralf S. Engelschall. *User Interface Composition*.
URL: http://engelschall.com/go/EnTR-03:2013.12 (visited on 06/11/2014).

[For04] Bryan Ford. „Parsing expression grammars: a recognition-based syntactic foundation“. In: 2004.
URL: http://pdos.csail.mit.edu/~baford/packrat/popl04/.

[Fow] Martin Fowler. *Presentation Model*.
URL: http://martinfowler.com/eaaDev/PresentationModel.html (visited on 06/10/2014).

[FR10] Martin Fowler and David Rice. *Patterns of enterprise application architecture*. 16th ed. The Addison-Wesley signature series. Boston and Mass. [u.a.]: Addison-Wesley, 2010.

[Gam+95] Erich Gamma et al. *Design Patterns*. Reading, MA: Addison Wesley, 1995.

[Grea] Derek Greer. *Interactive Application Architecture Patterns*. URL: http://lostechies.com/derekgreer/2007/08/25/interactive-application-architecture/ (visited on 06/10/2014).

[Greb] Derek Greer. *The Art of Separation of Concerns*. URL: http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/ (visited on 06/13/2014).

[Har] John Harding. *Flash and the HTML5 <video> tag*. URL: http://apiblog.youtube.com/2010/06/flash-and-html5-tag.html (visited on 05/27/2014).

[Hua] Yi Ming Huang. *Find and resolve browser memory leaks caused by JavaScript and Dojo*. URL: http://www.ibm.com/developerworks/library/wa-sieve/ (visited on 06/13/2014).

[Lev66] VI Levenshtein. „Binary Codes Capable of Correcting Deletions, Insertions and Reversals“. In: *Soviet Physics Doklady* 10 (1966), p. 707.

[Lie94] Håkon Wium Lie. „Cascading Style Sheets“. PhD thesis. University of Oslo, Faculty of Mathematics and Natural Sciences, 1994.

[Lik] Jeremy Likness. *Model-View-ViewModel (MVVM) Explained*. Appendix B: Pre-existing MVVM Frameworks. URL: http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained (visited on 06/10/2014).

[Mak] Hayim Makabee. *Separation of Concerns*. URL: http://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/ (visited on 06/13/2014).

[Mar10] Andreas Martens. *Architekturbasiertes Vorgehensmodell zur Identifizierung und Lokalisierung von Architektur-Kriterien in Enterprise-Anwendungen. [Architecture based process modell for identifying and locating architecture criteria in enterprise applications].* University of Paderborn, 2010.

[McC] Brian McCallister. *Bulkheads*. URL: http://skife.org/architecture/fault-tolerance/2009/12/31/bulkheads.html (visited on 06/11/2014).

[Mey+] Leo A. Meyerovich et al. *Flapjax: A Programming Language for Ajax Applications*. Section 3.4: Propagation.

[Mod] Mitchell Model. *Model View Controller History*. URL: http://c2.com/cgi/wiki?ModelViewControllerHistory (visited on 06/06/2014).

[MSDN] Microsoft MSDN. *Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF*. URL: http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx (visited on 06/10/2014).

[Nil02] Jimmy Nilsson. *.NET Enterprise Design with Visual Basic .NET and SQL Server 2000*. Sams, 2002.

[Nyg] Michael Nygard. *Stability Patterns ...and Antipatterns*.
URL: http://cdn.oreillystatic.com/en/assets/1/event/79/Stability%20Patterns%20Presentation.pdf (visited on 06/11/2014).

[Pic] Sundar Pichai. *Introducing the Google Chrome OS*.
URL: http://googleblog.blogspot.de/2009/07/introducing-google-chrome-os.html (visited on 05/27/2014).

[Rm] *THE REACTIVE MANIFESTO*.
URL: http://www.reactivemanifesto.org/ (visited on 06/11/2014).

[Sha] Dharmesh Shah. *The Thin Client, Thick Client Cycle*.
URL: http://onstartups.com/tabid/3339/bid/161/The-Thin-Client-Thick-Client-Cycle.aspx (visited on 05/27/2014).

[Smi] Josh Smith. *Google Groups - Thought: MVVM eliminates 99% of the need for ValueConverters*.
URL: https://groups.google.com/forum/#!topic/wpf-disciples/P-JwzRB_GE8 (visited on 06/10/2014).

[Sto] Paul Stovell. *What is Reactive Programming?*
URL: http://paulstovell.com/blog/reactive-programming (visited on 06/11/2014).

[SW] Artem Syromiatnikov and Danny Weyns. *A Journey Through the Land of Model-View-* Design Patterns*.
URL: http://homepage.lnu.se/staff/daweaa/papers/2014WICSA.pdf (visited on 06/11/2014).

[Vaa13] Christian Vaas. *Pattern Guideline and Constraint Validation of Run-time Communication in User Interface Component Architectures*. 2013.

[Vog] Lars Vogel. *JFace Data Binding - Tutorial*.
URL: http://www.vogella.com/tutorials/EclipseDataBinding/article.html#databinding_overview (visited on 06/16/2014).

[Wal] David Walsh. *How JavaScript Event Delegation Works*.
URL: http://davidwalsh.name/event-delegate (visited on 06/20/2014).

[We] wealthfront Engineering. *Reactive.js: Functional Reactive Programming in Javascript*.
URL: http://eng.wealthfront.com/2013/04/reactivejs-functional-reactive.html (visited on 06/11/2014).

[Weba] *Application Specification — TodoMVC*.
URL: https://github.com/tastejs/todomvc/blob/master/app-spec.md (visited on 09/11/2014).

[Webb] Pew Research Center. *Cell Internet Use 2013 — Main Findings*.
URL: http://www.pewinternet.org/2013/09/16/main-findings-2/ (visited on 05/27/2014).

[Webc] *Commercial Rich client platform (RCP) applications*.
URL: https://www.eclipse.org/community/rcpcp.php (visited on 06/16/2014).

[Webd] *Data Binding Overview*.
URL: http://msdn.microsoft.com/de-de/library/ms752347(v=vs.110).aspx (visited on 06/18/2014).

[Webe] *Event delegation in JavaScript*.
URL: http://facebook.github.io/react/docs/reconciliation.html (visited on 06/20/2014).

[Webf] *Facebook React*.
URL: http://facebook.github.io/react/ (visited on 06/20/2014).

[Webg] *Facebook React — Getting Started*.
URL: http://facebook.github.io/react/docs/getting-started.html (visited on 06/20/2014).

[Webh] *Facebook React — Tutorial*.
URL: http://facebook.github.io/react/docs/tutorial.html (visited on 06/20/2014).

[Webi] Miniwatts Marketing Group. *Internet World Stats*.
URL: http://www.internetworldstats.com/stats.htm (visited on 05/27/2014).

[Webj] *Introduction to WPF*.
URL: http://msdn.microsoft.com/de-de/library/aa970268(v=vs.110).aspx (visited on 06/18/2014).

[Webk] *JavaServer Faces Technology*.
URL: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html (visited on 06/17/2014).

[Webl] *JFace Data Binding/The New Binding API*.
URL: http://wiki.eclipse.org/index.php?title=JFace_Data_Binding/The_New_Binding_API&oldid=358773 (visited on 06/16/2014).

[Webm] *KnockoutJS — Creating custom bindings*.
URL: http://knockoutjs.com/documentation/custom-bindings.html (visited on 06/20/2014).

[Webn] *KnockoutJS — The data-bind syntax*.
URL: http://knockoutjs.com/documentation/binding-syntax.html (visited on 06/20/2014).

[Webo] *KnockoutJS — The "foreach" binding*.
URL: http://knockoutjs.com/documentation/foreach-binding.html (visited on 06/20/2014).

[Webp] *KnockoutJS — The "with" binding*.
URL: http://knockoutjs.com/documentation/with-binding.html (visited on 06/20/2014).

[Webq] Knockoutjs.com. *MVVM and View Models*.
URL: http://knockoutjs.com/documentation/observables.html (visited on 06/10/2014).

[Webr] Mac Developer Library. *Cocoa Core Competencies - Model object*. URL: https://developer.apple.com/library/mac/documentation/general/conceptual/devpedia-cocoacore/ModelObject.html#//apple_ref/doc/uid/TP40008195-CH31-SW1 (visited on 06/06/2014).

[Webs] Mac Developer Library. *Cocoa Core Competencies - Model-View-Controller*. URL: https://developer.apple.com/library/mac/documentation/general/conceptual/devpedia-cocoacore/MVC.html (visited on 06/06/2014).

[Webt] *MVVM: ViewModel and Business Logic Connection*. URL: http://stackoverflow.com/questions/16338536/mvvm-viewmodel-and-business-logic-connection (visited on 06/10/2014).

[Webu] *Ractive.js — The diamond age of web development*. URL: http://www.ractivejs.org/ (visited on 06/20/2014).

[Webv] *Rich Client Platform/FAQ*. URL: http://wiki.eclipse.org/index.php?title=Rich_Client_Platform/FAQ&oldid=328452 (visited on 06/16/2014).

[Webw] Paul Sasik. *Why use MVVM?* URL: http://stackoverflow.com/questions/2653096/why-use-mvvm (visited on 06/10/2014).

[Webx] *Templates are not HTML valid — Ractive Issues*. URL: https://github.com/ractivejs/ractive/issues/185 (visited on 06/23/2014).

[Weby] *The Java EE 6 Tutorial — Managed Beans in JavaServer Faces Technology*. URL: http://docs.oracle.com/javaee/6/tutorial/doc/bnaqm.html (visited on 06/17/2014).

[Webz] *The Java EE 6 Tutorial — Referring to Object Properties Using Value Expressions*. URL: http://docs.oracle.com/javaee/6/tutorial/doc/bnahu.html#bnahx (visited on 06/17/2014).

[Webaa] Arjan Tijms. *Understanding JSF as a MVC framework*. URL: http://stackoverflow.com/questions/10111387/understanding-jsf-as-a-mvc-framework (visited on 06/17/2014).

[Webab] *UpdateTargetTrigger*. URL: http://stackoverflow.com/questions/13434675/updatetargettrigger (visited on 06/18/2014).

[Webac] *Usage statistics and market share of JQuery for websites*. URL: http://w3techs.com/technologies/details/js-jquery/all/all (visited on 09/03/2014).

[Wikia] Wikipedia. *Fat client — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Fat_client&oldid=600016640#Advantages (visited on 05/27/2014).

[Wikib] Wikipedia. *KnockoutJS — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=KnockoutJS&oldid=611805640 (visited on 06/10/2014).

[Wikic] Wikipedia. *Model–view–controller — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Model%C3%A2%C2%80%C2%93view%C3%A2%C2%80%C2%93controller&oldid=610228884 (visited on 06/06/2014).

[Wikid] Wikipedia. *Reactive programming — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Reactive_programming&oldid=612098085 (visited on 06/11/2014).

[Zak] Nicholas C. Zakas. *Event delegation in JavaScript*. URL: http://www.nczonline.net/blog/2009/06/30/event-delegation-in-javascript/ (visited on 06/20/2014).