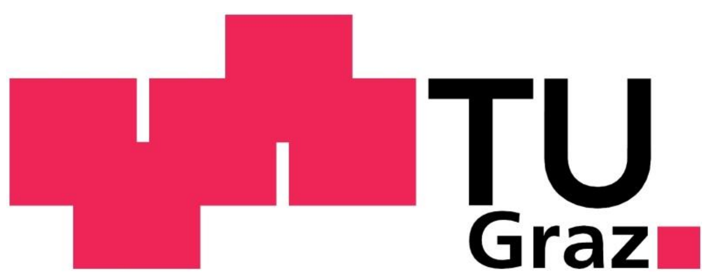
Masterarbeit

A New Approach to Web App Development Utilizing Behavior Driven Development Together with New Tools and Frameworks

Georg Kothmeier
georg.kothmeier@student.tugraz.at

Institut für Softwaretechnologie (IST)
Technische Universität Graz
Inffeldgasse 16B/II
8010 Graz, Österreich



Graz University of Technology

Begutachter und Betreuer: Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

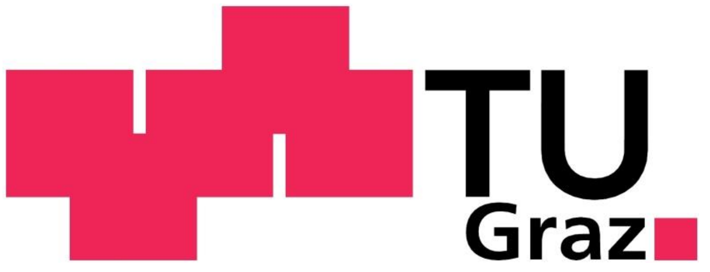
Graz, August 2014

Master's Thesis

A New Approach to Web App Development Utilizing Behavior Driven Development Together with New Tools and Frameworks

Georg Kothmeier
georg.kothmeier@student.tugraz.at

Institute for Software Technology (IST)
Graz University of Technology
Inffeldgasse 16B/II
8010 Graz, Austria



Graz University of Technology

Assessor and supervisor: Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Graz, August 2014

Zusammenfassung

Apps sind heutzutage allgegenwärtig. Allerdings werden mit Apps oft nur Smart Phone Apps in Verbindung gebracht. Diese Arbeit versucht einen neuen Blick auf Apps zu präsentieren. Dafür wird das Konzept von Web-Apps beschrieben und erklärt. Das Anwendungsfeld für Web-Apps ist riesig und erstreckt sich von simplen Apps welche für jede Plattform zugänglich gemacht werden sollen bis hin zu komplexen Anwendungen als „Software-As-A-Service“ welche über das Internet verfügbar sind. Da Webtechnologien (HTML, CSS, JavaScript etc.) von nahezu jedem Gerät unterstützt werden, eröffnen sich völlig neue Wege Software für alle Plattformen bereit zu stellen. Langezeit war dies nicht möglich, da die Engines und Standards dieser Technologien sehr unterschiedlich waren. Glücklicherweise haben sich diese Dinge immer mehr angenähert und somit bilden Webtechnologien eine gute Basis für professionelle Software Entwicklung. Die verbleibenden Probleme und Herausforderungen können oft mit Frameworks und Tools gelöst werden. Die Web Development Community ist riesig und bietet unzählige Libraries und Tools an, welche zur freien Verwendung zur Verfügung stehen. Da die Auswahl nahezu unbeschränkt ist, ist es sehr schwer das richtige Tool zu wählen, deswegen ist es unerlässlich die Wahl für jedes Projekt neu zu evaluieren. Im zweiten Teil dieser Arbeit werden einige dieser Tools vorgestellt. Das Wissen zu diesen Tools wurde durch das praktische Projekt über einen Zeitraum von 15 Monaten gesammelt. Weitere Informationen wurden aus Online-Ressourcen und Büchern zusammen getragen. Das Framework Angular.js vereinfachte das entwickeln des Frontend-Codes enorm. Aus diesem Grund erhielt Angular.js ein eigenes Kapitel.
Die App welche im praktischen Teil der Arbeit umgesetzt wurde, wurde als Web-App realisiert. Durch den Einsatz von Angular.js ist die App nach dem Single Page Application Muster aufgebaut. Dieses Muster ist im Grunde eine thin-server/fat-client Architektur. Dieses Konzept stellt allerdings einen Paradigmenwechsel im Web Development dar. Dieser Wechsel wird in Kapitel 4 detailliert beschrieben.
Aber auch die besten Tools und Konzepte führen nicht zwangsläufig zu erfolgreichen Projekten. Der Einsatz einer soliden Software Development Methode macht es sehr viel wahrscheinlicher ein Projekt erfolgreich abzuschließen. Aufgrund der enormen Wichtigkeit von Software Development Methoden liegt der Fokus des ersten Teils der Arbeit auf diesen. Da die Web Community und im Speziellen die JavaScript Community Behavior Driven Development intensiv nutzt, beschäftigt sich diese Arbeit intensiv mit dieser Methode. 2003 wurde Behavior Driven Development erfunden und trotzdem sind viele Bereiche noch immer nicht klar definiert und es gibt viele Diskussionen wie bestimmte Teile davon angewandt werden sollten. Dies geht sogar so weit, dass viele daran zweifeln, dass Behavior Driven Development eine vollwertige Software Development Methode ist. Die Nachforschungen führten auch in dieser Arbeit zu dem Schluss, dass Behavior Driven Development nicht als solche anzusehen ist. Zu viele Punkte bleiben offen und sind nicht definiert.
Das führte zu der Notwendigkeit neben Behavior Driven Development eine weit verbreitete Software Development Methode zu untersuchen. Es stellte sich heraus, dass Extreme Programming und Behavior Driven Development viele Gemeinsamkeiten haben. Deswegen könnte man diese beiden gemeinsam nutzen. Im Idealfall würde das zu einer Verbesserung beider Ideen führen. Ein Vorschlag wie diese Kombination bewerkstelligt werden könnte ist in Kaptiel 3.7 skizziert.
Nach 15 Monaten intensiver Arbeit am praktischen Projekt und während den Recherchen für diese Arbeit stellte sich heraus, dass für Web Development ein besonders intensiver Einsatz von Software Development Methoden erforderlich ist. Das ist auf die lebhafte Natur des Webs zurück zu führen. Anforderungen und Herausforderungen ändern sich ständig und es muss möglich sein auf diese Änderungen möglichst schnell zu reagieren. Mit Extreme Programming gibt es eine Software Development Methode mit welcher diese Herausforderungen bestens gemeistert werden können. Behavior Driven Development unterstützt das Team im Aufbau einer soliden Testsuite. Mit diesen beiden Ansätzen ist es sehr wahrscheinlich ein Projekt erfolgreich umzusetzen.
Für Behavior Driven Development gibt es eine Vielzahl an Tools (auch für JavaScript). Somit ist von Unit Tests bis Acceptance Tests und ausführbaren Spezifikationen in Form von Gherkin alles möglich. Nur weil es für jeden Aspekt von Behavior Driven Development ein Tool gibt, heißt das nicht, dass man alles umsetzen muss. Für jedes Projekt sollte abgewogen werden, ob der Nutzen dieses Tools wirklich den Aufwand rechtfertig. Die Pro und Kontras der einzelnen Aspekte von Behavior Driven Development werden in Kapitel 3.6 dargelegt. Kapitel 6.2.4 skizziert wie Behavior Driven Development in einem Web-App Projekt auf pragmatische Art und Weise umgesetzt werden kann.

Abstract

Apps are ubiquitous nowadays. Many people link the word app only with smart phone apps. This thesis presents a new view on apps and therefore the concept of a web app is described. The field of application of web apps is huge. It could be a simple app which should be accessible by all platforms and devices or it could be a sophisticated software as a service which is available over the web. Because web technologies (HTML, CSS, JavaScript etc) are supported by almost every device, web apps open the possibility to write a software which is available for every platform. For a long time this was not feasible because of diverse engines and standards in web technologies. These kinds of things have harmonized and now form a solid base for professional software development. For the remaining problems there are many frameworks and options to solve them. The web development community is vast and offers a wide range of third party libraries which are ready to use. This makes the choice difficult and has to be evaluated before every project. The second part of the thesis describes some of the tools available in more detail. The insights about these tools were gathered during a practical project over 15 months. Further information was collected from online resources and books. The choice of the tools turned out as a good one, especially Angular.js eased development of frontend code a lot. Therefore, this particular framework has its own chapter.

Angular.js helped immensely to build our web app. This app was realized as a single page application. The term single page application refers to an app which is implemented according to the thin-server/fat-client design pattern. This is a paradigm shift to traditional web development. The differences to the conventional model are outlined in chapter 4.

However, the best tools and technologies will not lead to successful projects; therefore, a sound software development methodology is indispensable. Because this is crucial for every software project, nevertheless if it is a web app or traditional, this topic is addressed in part one of this thesis. The focus in part one is also on web app development. The web community and especially the JavaScript community take heavy usage on behavior driven development. Therefore this methodology was investigated in more detail. Behavior driven development started to evolve in 2003 but about many parts there is still no common sense. It is unclear if behavior driven development is a full software methodology or just an extension of test driven development. This thesis comes to the conclusion that behavior driven development is more of the latter form.

This is the reason why widely adopted software methodologies were investigated. During this research it turned out that extreme programming and behavior driven development have several parts in common. If used together they can enhance each other. A suggestion how these two methodologies could play together is outlined in chapter 3.7.

After 15 month working on the practical project and after extensive researches for this thesis it turns out that web development needs special intense usage of a software methodology. This is due to the vivid nature of the web. Requirements and challenges are changing fast and therefore it must be possible to adapt to these changes as quickly as possible. With extreme programming there is a methodology which fits well to this environment and behavior driven development helps to create a powerful test suite. These two approaches often lead to successful projects.

There are plenty of tools to utilize to do behavior driven development, especially for JavaScript, where everything from unit testing to acceptance testing and executable specifications in the form of Gherkin are available. Despite the fact that there is a tool for every aspect of behavior driven development it has to be evaluated if everything is really needed for the particular project. Pros and cons of many aspects of behavior driven development will be described in chapter 3.6. How behavior driven development could be accomplished in a pragmatic way for a web app project is outlined in chapter 6.2.4.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz am: .... Unterschrift: ....

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Date: .... Signature: ....

List of Figures

Figure 1: Agile vs Waterfall, data source [Stan11, p. 25] 5
Figure 2: comparison of waterfall and iterative lifecycle. 5
Figure 3: outside in approach with behavior driven development (Source [Che10, p. 285]) 17
Figure 4: Comparison of traditional web app architecture with the single page app architecture 28
Figure 5: Comparsion of JavaScript the good parts vs JavaScript the definitive guide 35
Figure 6: results of a test run with the default Jasmine test runner 42
Figure 7: comparsion lines of code for Googlefeedback: GWT to Angular.js 49
Figure 8: Comparison lines of code at Localitys: Backbone.js to Angular.js 49
Figure 9: Batarang output of scopes of the simple example to demonstrate scope hierarchies 54
Figure 10: Mockup of a very simplistic user interface for the given example for Karma and Protractor 62

List of Listings

Listing 1: comparing TDD syntax in form of QUnit with BDD syntax in form of Jasmine 12
Listing 2: a quick intro to features written in Gherkin for Cucumber 14
Listing 3: Comparison of JavaScript for-in and PHP foreach 32
Listing 4: example of how easy it is to install a package with npm 33
Listing 5: example of a very simple express.js server listening on http port 3000 33
Listing 6: example of how to save and fetch data in MongoDB 33
Listing 7: Example of Jasmine describe/it 41
Listing 8: example of nested describe blocks in Jasmine 41
Listing 9: Defining a custom matcher in Jasmine 43
Listing 10: Using a custom matcher in Jasmine 43
Listing 11: Example of a production code for the described tournament management 45
Listing 12: Jasmine test code for a tournament management example app 46
Listing 13: The view of the “Hello World!” example in Angular.js 51
Listing 14: The business logic of the "Hello World!" example in Angular.js 52
Listing 15: change the view of the "Hello World!" example to show two-way-data binding 52
Listing 16: The view of a simple example to illustrate $scope hierarchy in Angular.js 53
Listing 17: The business logic of a simple example to illustrate $scope hierarchy in Angular.js 53
Listing 18: Shows a high coupling between two components 58
Listing 19: Example of constructor dependency injection 59
Listing 20: Example of setter dependency injection 59
Listing 21: Example of over complicated constructor dependency injection 59
Listing 22: Example how dependencies are annotated in Angular.js 60
Listing 23: Installing karma and karma plug-ins 61
Listing 24: Jasmine specs for the Protractor example 63
Listing 25: Possible implementation of the enrollment page object 64
Listing 26: Test code for Protractor to automate testing of the given example 65
Listing 27: A possible user interface for the Karma and Protractor example 66
Listing 28: Jasmine Specs for Karma for the AuthService 67
Listing 29: Jasmine Specs for Karma for the PlayersService 68
Listing 30: Jasmine Specs for Karma for the IntroController 69

Acknowledgments

I want to thank Dr. Wolfgang Slany for giving my colleague Paul Kapellari and me the chance to realize our idea as a web app. Beside the practical knowledge we gained we were able to learn a new software methodology and explore a new software development stack. Also Dr. Wolfgang Slany gave us many important inputs and supported us during the whole stage of the project and thesis.

It was also a pleasure to work together with my colleague Paul Kapellari during the whole project. We pushed each other and therefore we never lost the motivation to work on this project. It was great to use his flat as an office. Big thanks also to his family which supported us and always provided us with a great meal during working.

A very important person during the whole project was Valentin Robtisch from BSVÖ (Billard Sportverband Österreich). He was so fascinated by our idea that he put his trust in us. He made it possible to adopt our web app to the needs of BSVÖ. Also he was our customer with whom we practiced our software development methodology.

During my whole life my family supported me incredibly. They helped me in good and bad times. Also during this thesis they supported me and always trusted in me. Thank you for that. All this would not be possible without you!

Another acknowledgment goes to all my friends and special thanks go to Nick Hayes and Mason Sams for correcting my English.

Georg Kothmeier

Graz, Austria, August 2014

Table of contents

List of Figures ... iv
List of Listings ... v
Acknowledgments ... vi

1 Introduction ... 1
2 Software development methodologies ... 3
2.1 Traditional software development ... 4
2.2 Agile software development ... 5
2.2.1 Agile Manifesto ... 6
2.2.2 Extreme Programming ... 7
3 Behavior Driven Development ... 11
3.1 Unit testing in behavior driven development ... 11
3.2 Ubiquitous language ... 13
3.3 ATDD with behavior driven development ... 13
3.4 Specs vs features or both? ... 14
3.5 Behavior driven development in more detail ... 15
3.5.1 Project inception ... 15
3.5.2 The behavior driven development cycle ... 16
3.5.3 Outside in approach ... 16
3.5.4 Feedback stage ... 17
3.5.5 Stories ... 18
3.6 Summary behavior driven development ... 21
3.7 Behavior driven development and extreme programming ... 22
3.7.1 Planning the Game and Project Inception ... 23
3.7.2 Metaphor ... 23
3.7.3 Testing and refactoring ... 23
3.7.4 Coding ... 24
3.7.5 On-site customer vs customer writing gherkin ... 24
3.7.6 Conclusion ... 25
4 Web engineering ... 26
4.1 Web app development ... 26
4.2 Single page applications ... 26
5 JavaScript ... 29
5.1 The rise of JavaScript ... 29
5.1.1 Strengths ... 30

5.1.2 Problems 34
5.2 BDD in JavaScript..... 39
5.2.1 Jasmine vs “Mocha/Chai/Sinon” 39
5.2.2 Jasmine..... 40
6 Anuglar.js 48
6.1 Concepts 51
6.1.1 Templateing..... 51
6.1.2 Two way data binding..... 52
6.1.3 $scope 53
6.1.4 Dependency Injection 54
6.1.5 Directives 55
6.1.6 Code organization..... 55
6.2 Testing Angular.js 57
6.2.1 Dependency injection 58
6.2.2 Karma test runner 60
6.2.3 Protractor..... 61
6.2.4 Example of Karma and Protractor 62
6.3 Summary on Angular.js..... 70
6.3.1 Outlook 70
7 Conclusion..... 72

A Appendix 75
B Bibliography 90

1 Introduction

"Isn't there an app for that?"

"App" is a hype-word of the last few years. Basically an app is just software as we have been accustomed to for decades. There is only one different mechanism to install it, at least for Windows users. Linux has come with a package management system since its early days. This package manager makes installing software similar to the well known app stores from Google and Apple. One new aspect about apps in comparison to regular software is that they mostly run on mobile devices. Windows tries to bring the look and feel of these mobile apps also to the desktop with their new operating system Windows 8. Nowadays, apps can be installed almost anywhere from smart phones to desktops and even on TVs and cars.

In this way the potential for apps is huge, but with new apps there is a serious problem arising: the increasing cost of supporting all systems. There are iOS, Android, Windows Phone, and Blackberry (and some more operating systems with a smaller market share) only on the smart phone market. Besides that, it would be beneficial to the consumer and the provider if the app is also available for desktops and other devices. To develop an app natively means often rewriting it from scratch for every platform. Especially for small companies and start ups this is not an option.

In many cases web technologies could be used to implement software that is executable on many platforms. Almost every system is connected with the internet and therefore has a browser. This can be utilized to develop software which can run everywhere. Other tools have tried this already, but web technologies have the advantage of only needing a web browser to succeed rather than any additional plug-ins or runtime environment. A browser is enough to execute the software. The browser market is much more homogenized nowadays. This means that their engines do not differentiate as much as in the early days of web technologies together with new trends in HTML5, CSS3 and JavaScript it is feasible to develop web apps for many devices and platforms.

Frameworks like Twitter Bootstrap[1] or Foundation[2] ease developing of responsive web apps. New projects like Polymer[3] and x-tags[4] provide the reuse of user interface components. Google's material design components are already implemented in Polymer and AngularJS.

The shift from fat-servers and thin-clients to thin-servers and fat-clients also allows an implementation of reactive user interfaces. Therefore, the user experience becomes like the one in native apps. This is why software as a service becomes more and more popular. In addition to this trend there are many frameworks which help developers to overcome common problems. The architectural background will be discussed in chapter 4 and practical solutions are outlined in chapter 6.

Tools like Cordova[5] provide the possibility to "compile" web apps to smart phone apps. Cordova even provides access to devices like the camera, accelerometer, compass etc of a smart phone[6]. If a device is not available, there are many third party plug-ins to use.

---

[1] Detailed information about Twitter Bootstrap: http://getbootstrap.com/ (retrieved 18.07.2014)

[2] Further details about Foundation: http://foundation.zurb.com/ (retrieved 18.07.2014 12:45)

[3] Polymer's project website provides further details: http://www.polymer-project.org/ (retrieved 18.07.2014 13:12)

[4] Further reading about x-tags: http://www.x-tags.org/ (retrieved 18.07.2014 13:13)

[5] The Cordova website is available under: http://cordova.apache.org/ (retrieved 18.07.2014 12:56)

Web apps are a great way to reach a huge audience without rewriting the same app over and over again. Of course web apps do not fit for every project but there are a lot of tools and frameworks to to help accomplish many things. For example, asm.js which even allows the playing of 3D games in the browser (asm.js and other alternatives are presented in chapter Transpiler).

But developing web apps also have their own caveats. Designing one user interface for all devices brings more work to user interface and user experience designers. Furthermore, the changes in web technologies are very fast and many standards are still not stabilized. But the biggest downside is that there is no choice of client side languages for the browser. Every app has to be JavaScript sooner or later (to solve specific issues with JavaScript, transpilers can be used. This idea will be expounded upon in the Transpiler chapter). As JavaScript was not intended for developing large scale applications with thousands of thousands of lines of code, it is essential to choose a well suited software development methodology.

Because over the web there is a huge market available and there are many competitors, release cycles have to be even quicker than in traditional software development. Also new trends and technologies arise quickly which makes predicting even the near future almost impossible. For this environment, agile software methodologies are well-suited (they will be described in chapter 2.2).

The web development community, especially the JavaScript community, utilizes behavior driven development for their quality management. Therefore, chapter 3 will put a strong focus on behavior driven development and compare it with other methodologies.

To gain a broad practical knowledge about web technologies and web development a web app was implemented as the practical part of this thesis. At first it started as an idea of a generic blogging system to feed social media channels. It was developed together with Paul Kapellari. But soon a customer entered the project. The customer was Carambol Billard Association of Austria (BSVÖ). The following code examples in this thesis are written from the perspective of a sports association like BSVÖ. The examples are simplified versions of the production code to outline the ideas and theoretical background better. The project started in May 2013 and is still ongoing (withstanding of July 2014). We worked closely with web technologies during this time and therefore also have a broad practical knowledge. All these experiences lead to chapter 7. Many of the theoretical things described in chapter 2 and 3 were applied in practical using web technologies. Chapter 6.2.4 outlines behavior driven development with Angular.js and its testing tools.

The first part of this thesis focuses on the theoretical background concerning software methodologies and behavior driven development. The second part provides an overview of tools, frameworks, best practices etc and their usage to accomplish the things described in part one.

Nevertheless the focus is always on behavior driven development combined with web technologies. This of course led to heavy usage of JavaScript. As behavior driven development puts its main focus on readable code, basic programming skills should be enough to get the ideas and concepts when reading the code samples. It is not necessary to understand every little detail of them. Nevertheless to fully understand all code listings a basic knowledge of JavaScript is needed.

[6] A detailed list can be found on the Cordova website:
http://cordova.apache.org/docs/en/3.5.0/guide_support_index.md.html#Platform%20Support (retrieved 18.07.2014 13:01)

2 Software development methodologies

To successfully develop software it is important to plan, organize and structure the project. The origins of structured processes to develop software reach back as far as the mid 1970s. There are many different methodologies and even more opinions which methodology fits best. The answer is not easy and it is not possible to say methodology X is better than methodology Y and methodology Z is the best of all. There is no “one-size-fits-all” methodology. For different projects different setups and methodologies can be the best choice. This thesis will cover two software development methodologies which are common and often used today. After providing a brief overview agile software development will be discussed in more detail. This is primarily because agile software development often leads to successful projects (see Figure 1).

As the field of software projects grows, it is becoming increasingly more complex to assure quality. It is not easy to measure what the best tactics are to tackle the quality problem. There are many discussions already taking place on how to develop high quality software. Today many users are used to software that crashes as well as the struggle of bugs, but quality has recently increased[7] and user experience has since improved. Software development is still a young engineering discipline compared to others like mechanical engineering or electrical engineering. This is the reason why there is still a lot of progress going on and new insights coming up on a regular basis. Although software quality is essential for every IT project, it is not easy to define when software has high quality.

- Does software needs to be defect free? Yes.
- Should software be fast? Yes.
- Should software be easy to maintain? Yes.
- Should software be easy to extend? Yes.

Even if all these points are fulfilled it is not certain if the user will accept the developed software, mostly because it is not clear if the software solve the user’s problem. Technical correctness also does not imply a great user experience. Additionally there is no guarantee that the user has a business advantage using the software. [Nel13, pp. 125-126] All these factors sum up to good or weak software.

If you do not have a de facto monopoly bad software quality will often result in failing projects. Dr. Paul Dorsey states that one of the main reasons for failing software projects is to not use a sound methodology [Dor00, p. 5]. Charette goes one step further and claims that even sloppily practiced development approaches would lead in failing projects [Cha05]. The first step to develop high quality software is to find the right methodology. It is not easy to tell which methodology is the best, but in software development the agile methodology seems to be a good fit, especially for web engineering agile development is well-suited. The web is evolving very fast and it is unpredictable what will be the next big thing in five years.

The most extreme example is Facebook. It was launched in 2004 as small platform and grew to become the leading social media network with over 1 billon members. Such developments are not predictable and therefore a methodology which lets enough space to adapt to new requirements quickly is indispensable. It is not only about the factors of the project (like the growth of Facebook), but also factors outside, which are changing quickly in web engineering.

Technologies are changing quickly and evolving constantly. This is due to the fact that it is easy to collaborate worldwide over the web and share experiences. These new technologies enable totally new concepts to create modern web apps. Furthermore users are more demanding - a simple HTML page will not beguile anybody anymore. The competition on the browser market opens new possibilities and leads to new technologies. The ever-changing nature of the web and its technologies is why agile software development for web engineering should be considered as an option.

---

[7] At least Windows does not crash anymore while presented http://www.youtube.com/watch?v=IW7Rqwwth84 (retrieved 29.05.2014 11:51)

2.1 Traditional software development

Traditional software development methodology is mostly described as the waterfall model. This model has some derivations, but all have in common that the whole project is planed beforehand. There are clear defined steps and the next step can only be started after the step before was accomplished. While this process is suited for old and mature engineering disciplines this approach has drawbacks in software development.

As software development is a young engineering discipline with constantly changing technologies and requirements it is hard to plan a complete project in advance. In many cases it is too rigid for software projects to plan every detail in a long term before. On the other hand most customers do not have the technical expertise to specify their requirements clearly at the beginning of the project. But planning, analyzing and designing are the first steps in the waterfall model. Then the coding process starts and at the end, testing and deployment are performed. Therefore, the feedback of the customer only comes at the end of the process.

Changes at this stage are expensive and should be avoided as much as possible [Ken99, p. 26]. The costs of changes are assumed to increase exponentially after each step in the waterfall model. This is often referred as the Boehm's cost of change curve [And03, pp. 246, Figure 27-2]. In agile software development, the cost of changes is ideally like a 1-log(n) curve. This is also not always the case but the curve of change costs is definitely lower than in water fall projects [And03, pp. 247-250].

If there is only one deployment phase there is no possibility to get feedback from the customer during the development process. For some small projects [Pet09] or some very specialized projects the waterfall model can still be a good fit. The fact that testing is applied at the end of the development process makes the waterfall model less suited for web projects. As stated before, the web is constantly evolving and changing and without a test suite it is almost impossible to stay up to date. Refactoring or updating third party libraries becomes a nightmare without a test suite.

2.2 Agile software development

> *"The agile process is the universal remedy for software development project failure. Software applications developed through the agile process have three times the success rate of the traditional waterfall method and a much lower percentage of time and cost overruns" – [Stan11, p. 25]*

As one can see from this quote The Standish Group is captivated by agile software development. They gathered data from projects from 2002 through 2010 and came up with the following chart[8]:

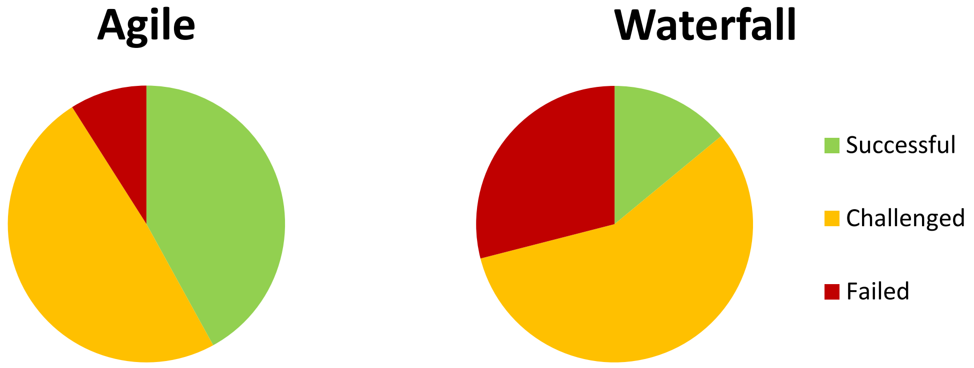


Figure 1: Agile vs Waterfall, data source [Stan11, p. 25]

If someone sees Figure 1 she or he could think that the waterfall model is totally useless. But basically the agile methodology does some kind of waterfall also. It packages the waterfall model into small iterations. It should be possible to deploy a release after iteration. The graphic from the book "The art of Agile Development" shows the differences and similarities quite well.

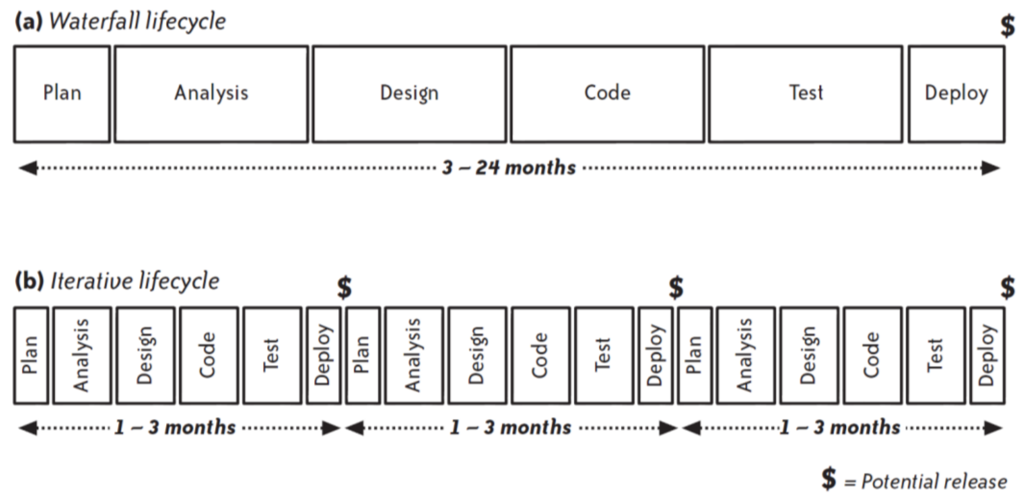


Figure 2: comparison of waterfall and iterative lifecycle.
Source: from the book "The art of agile development" [Sho07, p. 16]

[8] The Standish Group defines the project stages as follows:
"succeesfull: (delivered on time, on budget, with required features and functions);
challenged: (late, over budget, and/or with less than the required features and functions);
failed: (cancelled prior to completion or delivered and never used)." - [Stan11, p. 1]

Depending on which agile methodology someone chooses the iterative lifecycle will appear slightly different, but the main principles stay the same for every methodology. The iterative process is characteristic to all agile methodologies.

Agile software development aims to solve the main problems while building software. In literature there are many problems noted but four problems stand out of the rest [Nel13, p. 125], [Che10, pp. 125-126]:

- Finished too late or with too high costs
- The program does not do what the client was expecting
- The program is unstable and full of bugs
- Very hard to maintain

2.2.1 Agile Manifesto

The term “agile software development” was coined in early 2001 with the “Agile Manifesto”[9]. Long before the term agile software development was created there were many experienced developers who were advising to use new methodologies. They recognized that traditional project planning did not fit well for software projects. One for these visionaries was Kent Beck. He wrote the book “Extreme Programming – Manifesto” which was released 2000. [Bec00]

The start of agile software development was the “Agile Manifesto”. Therefore many smart developers came together to share their experiences about problems during the software development process. They came to the conclusion that all of them identified the same issues. So they wrote a short manifesto which outlined the main principles for agile software development. They are of course abstract, to leave space for every agile methodology to have their own focus and specialties. But no agile methodology should violate the basic rules:

> *“*
>
> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> ***Individuals and interactions*** *over processes and tools*
> ***Working software*** *over comprehensive documentation*
> ***Customer collaboration*** *over contract negotiation*
> ***Responding to change*** *over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more.*
> *” – Agile Manifesto – [Ken01]*

These basic rules are specified in more detail with twelve principles. These principles which found the base for the agile manifesto can be found online[10]. They give a more concrete explanation about the manifesto.

The four problems described earlier can be managed with agile software development quite well. This is mainly due to short release cycles and iterations (See Figure 2). The problem of shipping the software late can be handled better with small iterations, also over budget problems can be addressed easier because it is easier to prioritize features in a small project. Also it is possible to adapt to the customer’s needs quickly. The customer is involved strongly and sees results after every iteration. So the likelihood to develop the right tool for the customer rises dramatically. The shipped software will be more stable and easier to maintain. This is due to testing and the small release cycles [Che10, pp. 130-134].

---

[9] The manifesto can be read online: http://agilemanifesto.org/principles.html (retrieved 02.06.2014 10:45)

[10] The principles behind the Agile Manifesto: http://agilemanifesto.org/principles.html (retrieved 02.06.2014 11:09)

2.2.2 Extreme Programming

The extreme in the name does not mean programming 20 hours nonstop a day, alone in a dark room and hacking on some assembler code. In fact it is almost the total contrast to this flight of fancy. The "extreme" stands for taking the good things and bring them to extreme levels. Which means if something is good do it as often and as intense as possible. Kent Beck was the originator of the extreme programming methodology and defines the good things which should be taken to extreme as follows:

> "
> *If code reviews are good, we'll review code all the time (pair programming).*
> *If testing is good, everybody will test all the time (unit testing), even the customers (functional testing).*
> *If design is good, we'll make it part of everybody's daily business (refactoring).*
> *If simplicity is good, we'll always leave the system with the simplest design that supports its current functionality (the simplest thing that could possibly work).*
> *If architecture is important, everybody will work defining and refining the architecture all the time (metaphor).*
> *If integration testing is important, then we'll integrate and test several times a day (continuous integration).*
> *If short iterations are good, we'll make the iterations really, really short—seconds and minutes and hours, not weeks and months and years (the Planning Game).*
> *" – [Ken99, p. 7]*

It is easy to see that extreme programming is a pure form of agile software development. So it is the perfect methodology to outline the main principles (even if there are a lot of discussions[11] about some parts of extreme programming). Extreme programming is defined by Kent Beck as a lightweight methodology [Ken99, p. 9]. Typical paper work and administrative things should be reduced. The development process should be vivid and able to adapt quickly to new challenges.

The team

Decisions in the team should be made within a short time, ideally in less than a few hours. Therefore the structuring of the team is important. Extreme programming projects have less or even no hierarchy. All the members should be located close enough to each other to make communication fast and easy. It is important to make decisions quickly and to strengthen the team spirit. Beside the typical project team Kent Beck suggests to have an on-site customer available full time. He claims that this is essential to build the software the customer wants. The on-site customer is an integral part and more details follow in section On-Site Customer.

Planning the game

Planning the game is the first step. During this phase a project plan which shows approximate direction of the project is developed. This plan should give an overview of the next six month. It may also be less, but never more than a six month period as it is almost impossible to predict the different challenges and requirements in the long term. The whole team is involved in the planning process. The business people have to define scope and priority of features, the development team has to estimate theses wishes and elaborate on the consequences. With this knowledge and a schedule is arranged. After all project participants agree on the plan story cards are developed. These story cards are necessary for the iteration planning.

[11] Just google "extreme programming drawback" and see the flame wars

While developing the plan two things should be kept in mind:

- The goal is to maximize the value of the software
- This should be achieved with as little afford as possible

After “the game” is planed, iteration planning starts. Every iteration should not be longer than four weeks. Ideally each iteration is finished within a week. As mentioned before in the chapter 2.2, quick iterations are important to prevent that the project is heading in a wrong direction (which means it does not fulfill the customers’ expectations). For each iteration the team decides which story cards should be implemented. These story cards are prioritized and should be processed from important to less important. The story cards are split into task cards to minimize scope of every development step. The customer gets the chance to steer the development of the project after each iteration. Ideally the customer gets the product which fits her or his needs best. It is important to measure the results after each iteration. This means to verify if the estimations were correct. With these evaluations the team should be able to estimate its project speed correctly and define the scope in a right way.

Metaphor

The metaphor should represent the larger picture. This is likely what architecture is in traditional software development, but instead of a huge diagram with classes and interactions, the metaphor should tell a story. The story should be clear and remind all team members of the project’s main goal. A metaphor instead of an enormous large diagram is used because the main principle in the design phase is simplicity. The application should grow with each iteration and emerge from a small code base to a more complex one.

Testing

Extreme programming takes intensive usage of test driven development. The main paradigm is tested first. This means you have to write a test before you write your code. The test has to fail (if not, then some side effects make the test pass. This must be investigated), this is called red stage. Then the developer has to implement the code in the simplest way that the test passes, this is the green stage. If a developer finds a possibility to simplify the code she or he has to refactor it. This is the refactor stage. So there is a red/green/refactor cycle during development. These tests are important for every developer to get instant feedback if the incorporated code works and does not break existing functionality. Test driven development forms the basis for every agile software development method. Without testing refactoring would be impossible and without refactoring short release cycles are also impossible. Because only if refactoring works effectively it is possible to adapt design and architecture of an application dynamically. Tests which are written before the production code are called unit tests (in extreme programming). These tests should be written by programmers. Kent Beck suggests that functional tests should be written by the customer. These tests assure that some given feature or functionality really behaves as wanted. These kinds of tests are higher level and test not only one isolated component. In real world it turns out to be hard for customers to write tests (Behavior driven development tries to solve this issue with automated specifications, more on that in chapter Gherkin). To solve this issue, Kent Beck suggests having one team member as tester who helps the customer writing the functional tests.

Pair programming

Another central component of extreme programming is pair programming. Basically this means that two people are working on one computer. One person is writing code and the second person is thinking about the code the other is writing. The second person should think about possible tests, edge cases or ways to make code simpler. Pair programming should not be one person working and one person looking out of the window. There are several advantages if pair programming is executed well. The knowledge about the code is split across the whole team. There is no piece of code only one person knows. This reduces the risk of project failure because of just one single person. Also pair programming prevents developers from becoming

sloppy, meaning writing fewer tests or not sticking to coding conventions. Furthermore pair programming support the spread of knowledge in the team. The team members will always learn new things from each other.

Collective ownership

This means that the code which the team produces belongs to everyone. Therefore everyone is allowed to make changes everywhere. The only prerequisite is that a failing test is repaired as fast as possible. A failing test always has top priority and has to be fixed before anything else is done. Collective ownership leads to a more dynamic development process. Because everybody can refactor code and does not need to ask before altering the code somebody else has written. (The opposite of collective ownership would be individual ownership).

Continuous integration

To make all these things work, continuous integration is crucial. Without continuous integration it is impossible to quickly detect side effects introduced by refactoring. And because refactoring is an integral part of extreme programming it must be handled in a good way. The continuous integration tool should run the whole test suite over and over again. If some tests fail it should report this to the whole team. The failing test must be fixed as fast as possible.

40 hour week

It is important to not work more than 40 hours a week. If it is necessary the following week has to be 40 hours again. There should not be constant overtime. This avoids the circumstance that team members are stressed, tired and unmotivated. Only productive and highly motivated developers produce high quality code which brings relevant value to the project. Also the other parts of extreme programming are getting easier. Relaxed team members will communicate better with each other. Pair programming will work seamlessly and developers have the courage to do refactoring and improve code quality, instead of trying to solve a problem with some hack. If it is not possible to introduce a 40 hours week into the project there are certainly deeper problems which have to be resolved.

On-Site Customer

This means that a person from the customer is constantly available for the extreme programming team. It should be possible to request feedback and input from the on-site customer every time. This should avoid the risk of building something the customer does not need. The idea is that the on-site customer works at the same location as the extreme programming team. She or he can do regular work and should be just available for questions from the extreme programming team. Ideally the on-site customer is not disturbed in her or his daily work and only clarifies controversial issues. Kent Beck thinks that an on-site customer is extremely valuable and extreme programming without her or him is not as effective as it could be. Also he argues that the risk of building something wrong increases immensely without an on-site customer.

Coding standard

Besides all the quality assurance methods, like testing, continuous integration, pair programming it is indispensable to stick to a coding standard. This means there should be a common method for naming variables, classes, interfaces, methods and so on. Readable code is another mantra of extreme programming. It should be able to grasp what a certain part of the code does without the need of writing extra comments.

Summary

Extreme programming was taken as one example of agile software development. There are many others like Scrum or Kanban. But as said in the introduction, extreme programming takes the core principles to an extreme. This is why it is great to get an idea about agile software development. Furthermore it is widely adopted and accepted. Kent Beck says that extreme programming is not an all or nothing approach. Everyone can pick the things which fits her or his needs best, but he claims that it loses a lot of its strength if it is only applied partly. As behavior driven development is discussed in chapter 3 it must be stated that behavior driven development is not here to replace agile software development methodologies like extreme programming. Behavior driven development can be easily integrated into the existing methodologies. Behavior driven development tries to improve test driven development. As test driven development is an integral part of every agile software development methodology behavior driven development can enhance many projects. Behavior driven development can also be seamlessly integrated into extreme programming. A suggestion to combine it with extreme programming is made in chapter 3.7.

3 Behavior Driven Development

The term behavior driven development was created by Dan North 2003 and it still has no exact definition. Different literature tells you different definitions. But the main characteristics and the core idea are always the same. Solís and Wang conducted research about that in their paper "A Study of the Characteristics of Behaviour Driven Development" [Sol11].

Dan North thinks that many developers have problems getting started with test driven development. This is not because they are not able to write tests, it is more about getting the best out of test driven development. For people new to test driven development it is hard to understand how much testing is right, what parts must be tested and how a test suite should be structured. Dan North claims that this is due to the vocabulary used. Especially if a team needs to tackle a new problem it is hard to find tests beforehand. Therefore he suggests defining behavior instead of tests. The behavior of a component should be easier to specify as specifying a test [Nor06].

Especially if the team does not know how the problem will be solved, the word test pushes developers too much into verifying internals of objects. This is problematic because it can lead to flickering[12] tests or a hard to maintain test suite. If you need to change the internals of an object all tests which rely on these internal facts will fail. Even if the behavior of the object is the same and it still produces the expected output. Flickering test cases and hard to maintain test suites are annoying. This can lead to refusal of writing meaningful tests by the developers. Also the fact that the red/green/refactor cycle should drive the design is hampered by concentration on writing unit tests. The term unit test seems to be too vague for people new to test driven development. Many developers struggle to get the right amount of abstraction [Che10, pp. 23-24].

Many of the critics argue that it does not matter if someone uses the word test or the word behavior. And the behavior driven development folk around Dan North does not disagree. They often say that behavior driven development is test driven development done well[13] [Emr13, p. pos. 1766 eBook].

At the beginning of behavior driven development it was really just another way of doing test driven development. But nowadays it is much more as we will see in chapters 3.2 to 3.6. The argument that a different vocabulary and grammar makes it easier to head into the right direction is interesting and has a lot of truth to it. Dave Astels quotes the Sapir-Whorf hypothesis in his essay "A new look on test driven development" [Ast06]. This hypothesis is one of several which try to determine the influence of the used language on our thoughts. The Sapir-Whorf hypothesis states that the grammar and the vocabulary of a language influence the thoughts of a person. It even goes further and claims that people with different mother tongues cannot understand some of each other's thoughts because of the different vocabulary and grammar which is used in each language. This problem is not only appearing by people with different mother tongues. This also happens if customers, business analysts and developers talk to each other and try to define requirements. Behavior driven development addresses this problem with a ubiquitous language and a simple grammar to formalize specifications (more details as provided in chapter 3.2 and 3.5.5).

3.1 Unit testing in behavior driven development

But let us stay one more moment with the initial idea of behavior driven development which was to make it easier for developers to start with test driven development. Dan North suggests that naming of tests should be human readable. If someone reads the methods and class names they should tell a story also the test code (this is similar to the idea of readable code of extreme programming). Therefore a simple template for test

---

[12] Flickering tests are tests which occasionally fail

[13] To follow these vivid discussions just do a search in your favorite search engine

code is introduced. This template basically consists of describe/it blocks. There are many behavior driven development frameworks out and they have only small differences. For an example we will compare QUnit[14] and jasmine[15]. (QUnit is test driven development and jasmine is behavior driven development, more details about behavior driven development in JavaScript are provided in chapter 5.2).

```
// QUnit syntax
test('Lecture returns name', function() {
  var name = 'Introduction to BDD';
  var lecture = new Lecture(name);
  ok(lecture.getName()==name,'Lecture has a name');
});

test('Lecture returns num of enrolled students', function() {
  var name = 'Introduction to BDD';
  var lecture = new Lecture(name);
  var students = [ new Student('Max Mustermann'),
                   new Student('Martina Musterfrau') ];
  lecture.enroll(students);
  ok(lecture.getNumStudents() == students.length,
     'Lecture returned num of enrolled students');
});

// Jasmine syntax
describe("Lecture", function() {
  it("should have a name", function() {
    var name = 'Introduction to BDD';
    var lecture = new Lecture(name);
    expect(lecture.getName()).toEqual(name);
  });

  it("should return number of students enrolled", function() {
    var name = 'Introduction to BDD';
    var lecture = new Lecture(name);
    var students = [ new Student('Max Mustermann'),
                     new Student('Martina Musterfrau') ];
    lecture.enroll(students);
    expect(lecture.getNumStudents()).toEqual(students.length);
  });
});
```

Listing 1: comparing TDD syntax in form of QUnit with BDD syntax in form of Jasmine

The differences are subtle (especially in this toy example) but it should be easy to see how the jasmine syntax (and therefore the behavior driven development syntax) is telling the story better than the QUnit syntax. If you strip out all the key words like test and ok it is almost the same. But if someone is new to test driven development she or he will be distracted by these key words.

Another anti pattern which quite often occurs in test driven development is the duplication of design [Rad11, p. 19]. This means basically that a test is named after a method. Let's consider a class which fetches data from a database about a discussion. This class could have a method called fetchLastComments. In test driven development it would be ok to call the test "Test fetch last comments". (Although it is not a good idea) So what happens if we need to refactor the method and therefore give it a new name? At first the test would break. Then we fix the test and if we do not forget we would change the test name also. But at some point we will forget to change the test name and if we break the test, let's assume in six month, it will be confusing to read that testFetchLastComments fails because there is no such method name anywhere. Also this kind of

[14] QUnit is a project developed by the jQuery team, more about this project can be found online: http://qunitjs.com/ (retrieved 03.06.2014 13:25)

[15] Jasmine BDD Framework for JavaScript: http://jasmine.github.io/ (retrieved 16.06.2014 11:02)

naming makes tests uninformative, especially when they fail. Test driven development does not promote to do things like that. But there is also no mechanism to prevent a developer of doing it. Behavior driven development encourage the developer much stronger to specify the behavior instead of a test. The describe/it pattern is more likely to result in informative and readable tests.

3.2 Ubiquitous language

These first refinements of test driven development were designed for the developers. But behavior driven development covers a lot more. Another critical part of software development is the communication between stakeholders, business analysts and the development team. This is an important part because if there are misunderstandings there is no common sense what the final product should look like. This will lead to a failing project and a lot of troubles. Behavior driven development therefore tries to form a clear grammar and vocabulary to ease communication.

It is called ubiquitous language. The vocabulary should be defined by the domain experts and should be reflected from the requirements through the code. So the whole project should stick to this ubiquitous language. This could be imagined as a project jargon which everybody in the team and stakeholders agrees on and understands. To find the right vocabulary methods from domain driven design can be applied. There are also concepts for “Mapping Business Process Modeling constructs to Behavior Driven Development Ubiquitous Language” as Rogerio Atem de Carvalho, Fernando Luiz de Carvalho e Silva, Rodrigo Soares Manhaes describe in their same named paper [Car10]. This ubiquitous language is also used for describing the behavior of the system in a high level of abstraction. This makes it easy for all stakeholders and developers to define the requirements in a way which is understandable for all. Requirements are called stories in behavior driven development. These stories contain so called scenarios. Scenarios are events which belong to the story (more details about stories and scenarios are discussed in chapter 3.5.5)

3.3 ATDD with behavior driven development

In behavior driven development these stories and scenarios should be executable (details about what it means that stories or scenarios are executable are outlined in chapter 3.5.5). This has several reasons. It can be used as a proof to the customer that the system works as expected. All things the customer and developers agreed on (thanks to the ubiquitous language) can be checked automatically. So there is no need to click through an entire UI to verify the system. The stories are the acceptance criteria for the software. If all stories are fulfilled the software is considered as accepted.

On the other hand it is a living documentation. If the behavior of the system changes, some story or scenario will not be executable. Therefore the failing part has to be updated. Documentation in a non executable way (for example in a wiki) does not force the developers to update it when there are changes. As a result many documentations are outdated and not useful anymore. In practice there are several tools to execute stories. One of the most famous ones is Cucumber[16]. Cucumber uses Gherkin[17] as language for specifying a story. Unfortunately stories are called features in Cucumber. So stories and features are basically the same. To get a quick grasp of what a Cucumber feature looks like see the following listing (further details are explained in section Gherkin).

---

[16] An overview about all possibilities which Cucumber provides can be found on their website: http://cukes.info/ (retrieved 04.06.2014 11:40)

[17] A quick Gherkin introduction is available at the GitHub repository of Cucumber: https://github.com/cucumber/cucumber/wiki/Gherkin (retrieved 04.06.2014 11:41)

```
Feature: Login
   As a:    administrator,
   I want:  to be able to login
   so that: I can edit the data of our players

   Scenario: authenticate with correct credentials
      Given I am not logged in
       When I enter my correct username
        And I enter my correct password
       Then I should be redirected to the welcome page
        And I want to see the welcome message "Hello Admin"

   Scenario Outline: authenticate with wrong credentials
     ...
```

Listing 2: a quick intro to features written in Gherkin for Cucumber

All the key words and specialties will be explained later (in section Gherkin). As an introduction the example given in Listing 2 should be suffice. A feature consists of a title, a description and scenarios. As a description a free text can be used until the keyword scenario appears. Often it is suggested to use the Connextra format when writing the description. This means to use the template of "As a <role>, I want <desire> so that <benefit>". These keywords are highlighted green in Listing 2. This does not always make sense for example if an algorithm should be described or if words of the scenarios have to be described in more detail. A great example for this issue is provided in “The RSpec book” on page 47 [Che10, p. 47]. The Connextra format and its derivates have the advantage that customers, business analysts and developers have to stick to a tight framework. This should prevent misunderstandings and clarify things. Also it helps to make the benefit of a feature more visible.

3.4 Specs vs features or both?

Depending where the biggest challenges in a project are it can be decided to just use the describe/it syntax for replacement of test/assert or to use also ubiquitous language and executable stories/features. For example: if there are only developers on the team it could be overkill to define a ubiquitous language[18] and setup a whole testing framework for executable specifications. This could be the case if a team of developers decide to build an app and then release it. Then there are no real customers as long as the app was not published.

If there is a lot of communication going on between different types of people, like the customer, business analysts, developers etc then a ubiquitous language can be helpful. Also there will be a big benefit from executable stories. For this kind of projects the full spectrum of behavior driven development definitely makes sense.

But keep in mind, it is the same as with extreme programming, if you apply only some parts of a methodology it will lose a lot of its strength. Therefore we will take a more detailed look on how behavior driven development could be applied with using both aspects, executable specifications and new syntax for testing.

[18] Compare Biezemans’ answer to the question: “Can you describe for me some kind of problems that are a better fit for one tool than another? In what circumstances would someone use Cucumber.js, and when would they be better suited to using something else?” (http://www.infoq.com/news/2014/04/cucumberjs-bdd-biezemans, retrieved 05.06.2014 21:06)

3.5 Behavior driven development in more detail

To understand behavior driven development it is important to know its three main principles. These principles are the fundament of all decision.

- Enough is enough: planning and analyzing should not be too exhaustive. It is enough to plan and design just the first steps to start. But too little planning is also dangerous because there is no path to follow. It is essential to find a good balance. Developers often tend to over engineer things. This could be wasted time because it is almost impossible to plan the whole software project beforehand. Also during the implementation phase it is important to only work on the agreed features and scenarios. These are the only things which matter.
- Deliver stakeholder value: It only makes sense to do something which has a benefit. If nobody profits, stop doing it and do something different instead. Stakeholders in this case are not only customers. For instance: if someone implements a better build process the whole development team will profit.
- It's all behavior: therefore it is easy to use the same language throughout the whole project. This means from the code to the program and the conversations the same mindset can be used.

3.5.1 Project inception

Before every project is started there is the so called inception phase. During the inception phase all the people from the project come together and discuss what the product is about. This is done from a really general, birds-eye view. Technical details do not matter in this phase. After all participants have a common understanding about the idea and concept of the future product the ubiquitous language is defined. The vocabulary is set and ambiguous words are described and clarified. With this vocabulary themes will be defined.

Themes can be thought of the big parts of a product. For example: reporting, social media tools or content management. Every theme must be driven by a business outcome. Otherwise this theme does not fit to the principles "enough is enough" and "deliver stakeholder value". The themes are distilled into specific features.

These features should not contain too many technical details because of the likelihood of going off track. Often the whole planning process is hampered by technical details. Furthermore the details draw the attention away from the big view. It is often recommended to write the scenarios of every feature before iteration planning and not during the inception.

After the team defined the themes and features the risks for the project must be considered. Which risks there are outside of the project? For example: Is there a competitor who could implement a similar thing faster? Or are there some legal risks (licenses etc). Which risks are inside the project? For example: Is a certain technology the right one? Or can the team work together well? With the risks and the expected business outcomes the themes and features are prioritized. After inception phase the behavior driven development cycle starts.

One may think that the inception phase is like the "oldschool" top down planning but with the behavior driven development cycle there is no need to define everything beforehand. If new features or even themes are discovered they can be injected in the process easily. This is why also during the inception phase "enough is enough" should be the first premise. The goal during inception is to define the smallest marketable version of the product. All the other ideas and features should be included in the process only if they are still considered as useful after the smallest version is finished.

This strategy should avoid the problem that the release is delayed or in worst case never accomplished. This problem is often referred as YAGNI, which means “You Ain’t Gonna Need It”. YAGNI should be avoided at all cost. It is better to deliver a small software and enhance it from time to time as to never deliver a software because it has so many “great” features that it is never finished (in a releasable state).

3.5.2 The behavior driven development cycle

The behavior driven development cycles are basically the same as iterations in test driven development. Ideally one iteration is about one week long (but never longer than four weeks). Before the coding process starts, a business analyst and the stakeholders discuss which features are important and should be implemented as first. Therefore the features of the inception phase can be used but as the project goes on it is also possible to introduce new features. The business analyst helps the stakeholder to formulate her or his wishes as features. For this reason the business analyst also needs to have some basic understanding of behavior driven development. In small teams it is often the case that a special trained developer is business analyst and developer in one person.

After the stakeholders and the business analyst decided on the basic features the development team decides how much they can achieve during one iteration. This is of course only estimation and it is likely that they will not be correct for the first iterations. This is the reason why, it is essential to evaluate estimations after each iteration. This is the only way to get experience and improve estimations. The goal is that the development team learns to estimate their project velocity and select the right features to accomplish them in the given iteration period.

3.5.3 Outside in approach

With the set of selected features the development process starts. To lead developers into the right direction an outside in approach is used. This should avoid the question what to test and what not to test. Also it helps to assure that only the needed behavior is implemented and not more (remember YAGNI). At first the high level direction of the iteration is defined. Therefore scenarios of features have to be automated. This can be done for example with Cucumber. The automated scenarios are the acceptance criteria. If all scenarios pass their tests then the iteration is finished and the next one starts. But to make scenarios pass developers have to implement the actual code. This is doneby means of a test driven development. Therefore tools like RSpec[19] or Jasmine[20] can be used. The tests which are implemented in this phase are called specifications. They are more detailed about the behavior of a single object. In contrast features reflect the behavior of the whole system which means the behavior of many objects together. Specifications can be viewed as technical description of code objects and features are the desired behavior of the program (how the outside in approach is applied on a web app is outlined in chapter 6.2.4).

At first when a specification is written it will fail. This is because there is no code which fulfills the specification. Then the programmer starts to code. After each change the developer checks if the specification passes and if it passes she or he checks if the scenario passes. And this is what she or he does until all scenarios pass and therefore all features are fulfilled.

[19] RSpec is used for Ruby projects

[20] Jasmine is used for JavaScript projects

The following figure illustrates this outside in approach nicely:

Figure 3: outside in approach with behavior driven development (Source [Che10, p. 285])

The steps 1-7 of the implementation cycle are repeated until all scenarios passes. To get an idea about the steps they are briefly described next:

1. Take one feature and decided which scenario should be implemented. Only focus on this one scenario. ("enough is enough")
2. This leads to a failing scenario. To fix this, code must be implemented
3. Start to write a finer granular test, a so called specification with a behavior driven development toolkit, like RSpec for Ruby or Jasmine for JavaScript.
4. If implementation went well the specification should pass
5. If necessary do refactoring
6. Check if the scenario passes
7. When the scenario passes it is possible to do some refactoring
8. Steps 1-7 are repeated until all scenarios pass

It is clear to see why this is an outside in approach. You do not specify the behavior of some single object first and then try to form the behavior of the system. You start at a high level of abstraction and then go down to the technical details. This really eases the question of what to test and what not to test. Also it is unlikely that unnecessary things are implemented.

3.5.4 Feedback stage

After the iteration is finished business analysts, stakeholders and developers come together again. This is an important step and should not be neglected. Stakeholders should give detailed feedback if they like what developers implemented and if it fits to their expectations. If not the developers should modify the implementation and all people of the project have to reflect why the expectations were not met. This is also likely to happen during first iterations but after some time there will not be many misconceptions. This is the reason why the feedback round after each iteration is immensely important. If everything is fine the next

behavior driven development cycle can start. Business analysts and stakeholders select the next features and developers plan the iteration. Everything starts again from the beginning.

3.5.5 Stories

There are a lot of discussions going on about how to write good stories. “The RSpec Book” [Che10, pp. 145-148] the book “Instant Cucumber BDD How-to” [YeW13, pp. 38-46] and a blog post of Dan North [Nor14] give a good introduction into this topic. Though it is not a book about behavior driven development, chapter 2 of “Specification by Example” [Goj11, p. chapter 2] gives a good overview how to define stories. The ideas of these sources led to the following chapter.

Because communication between all participants of a project is enormously important in behavior driven development attention must be paid to writing good stories. Without good stories behavior driven development loses almost all its strength. Stories define the acceptance criteria and therefore are important for stakeholders and developers. Stakeholders know that their desired features work and developers know when the work is done.

Behind every story there should be a business outcome. But for non technicians it is often hard to define business outcome in a way that it is implementable. For instance the business outcome “reduce administrative costs by 4% with feature X” is to high level and abstract to start. Business analyst should guide customers to specify a story as a piece of functionality which is able to be demonstrated. This leads to more specific stories. On the other hand the business analyst has to hold back developers and testers to put too many technical details into a story. The amount of detail in a story should only be enough to get started and know the direction.

As described in the previous section (3.3) the Connextra syntax is a good template to get to the right amount of detail needed in the description of a story. There are also many derivates of this syntax but it is important that there is the benefit in the pattern. For example there is a variant which puts the benefit infront: "In order to <receive benefit> as a <role>, I want <desire>". The people promoting this variant argue that the different order puts stronger emphasis on the benefit. The benefit is an integral part of the story because it lets people think about why they want this feature. The “why” should be again driven by business outcomes. To find a good title to a story it is recommended to describe the title by an activity.

It is the business analyst’s and the stakeholder’s job to define a good title and description. With this outcome they specify together with a tester which scenarios are important. Again there should only be the basic scenarios. Nasty edge cases can be added when they are discovered. While writing a story and thinking about its scenarios it is not a good idea to focus on special and complex edge cases. The scenarios are written in the Given/When/Then syntax (as described in section Gherkin).

The title of the scenarios should be enough to distinguish each scenario from each other. With the fixed scenarios a technical developer discusses the story again with the others. The technical developer should come up with an estimation and eventually alternatives. This is important because customers can not always see the technical consequences of a desired feature. Therefore the technical developer should inform them about the costs, time and risks of the feature. With that in mind the story should be altered if necessary. Every story should be accomplishable during one iteration. If the technical developer thinks that the story is too big it needs to be broken up in several stories. The stories should not be broken up along technical lines. They should be broken up along business lines so that they are presentable chunks. For example: if a story is split into a user interface story and a database story there is no viewable result until both stories are accomplished.

After the discussion about the story everyone in the project should have a common understanding about what a certain feature is about. Therefore the scenarios form a common acceptance criterion which all participants of the project agreed on. With this amount of information it should be possible for developers to implement the right thing. However if there is a misunderstanding between customers and developers they will be

discovered quickly after each iteration (see feedback round section 3.5.4) and it is possible to quickly steer the project into the desired direction.

Cucumber

Cucumber is a tool which helps to automate stories. It understands the Gherkin language and maps the sentences from Gherkin to the actual code. There are several tools which can handle Gherkin for instance Yadda[21] for JavaScript or Spinach[22] which is an alternative to Cucumber for the Ruby community. But Cucumber is a widely adopted tool and ported for several programming languages. Initially Cucumber was written for Ruby but there exist ports for JavaScript, PHP, Java, C++ and even C# (with SpecFlow). So Cucumber is also a great tool if the same software has to be written in different programming languages. For instance: someone wants to develop a web app and smart phone app, with Cucumber can be assured that the app has the same behavior even if it was written in different programming languages. It is also possible to translate the English parts of the Gherkin language to another language. So it is possible to formulate stories in the natural language of the people on the project. For example there are translations for German, French or Spanish.

Gherkin

Gherkin is the language in which features can be written. As stated above there are many tools which can automate features in Gherkin. Therefore it is easy to formulate executable specifications and acceptance criteria. The basics of Gherkin were already outlined in Listing 2. Gherkin is not tied to any programming language. The mapping of the sentences is done by a tool like Cucumber. Especially in projects which needs to be duplicated in different programming languages this comes in handy. The behavior of an app should be the same on every platform no matter if it is a web app, an Android app or an iOS app. With Gherkin it is possible to use the same features and acceptance criteria. In the following chapter the Gherkin language will be discussed in more detail.

Keywords

Although Gherkin was not designed as a programming language there are several keywords. This is necessary for tools like Cucumber to make the features executable. Depending on which spoken language is used there are translations for the keywords [Way12, p. 28]. In English they are:

- Feature: the feature which should be implemented
- Background: the facts which have to be fulfilled before a scenario can be executed
- Scenario: a acceptance criteria for a special use case
- Scenario Outline: a acceptance criteria for several use cases
- Examples: keyword to specify examples of a scenario outline
- Given/When/Then: keywords to formulate a scenario
- And/But: keywords to refine Given/When/Then
- *: can be used instead of Given/When/Then and And/But
- @ and #: mark tags and comments

Feature

Not only keywords are important also a simple syntax has to be followed. At first the feature keyword has to be. This keyword is followed by the title of the feature. This title has to be on one line. The next lines are a description of the feature. This can be some narrative text or a description using the "as a" "i want" "so that"

[21] Their GitHub respository is available under https://github.com/acuminous/yadda (retrieved 06.06.2014 13:48)

[22] Spinach GitHub repository: http://codegram.github.io/spinach/ (retrieved 11.06.2014 10:42)

template (see chapter 3.3). The description stops when the next Gherkin keyword is discovered. This could be Scenario, Background or Scenario Outline. The last two are just simplifications to avoid writing duplicate code. This can be handy if the features are hard to read because always the same steps have to be done before checking for a certain state. Naming and description of a feature are crucial. The feature should be named after actions and summarize the expected behavior. Ideally the involved people grasp what is going on by reading the title without having to look at the description.

Scenario

As Background and Scenario Outline are just for simplification one must understand at first the concept of Scenario. A feature consists of several scenarios. Each scenario should be a real world example of different actions which belong to the feature. During planning there should only be basic scenarios and as discovered edge cases should be incrementally added. Every scenario has to be implemented by the developer. If implemented successfully the scenario passes and is fulfilled. If all scenarios are fulfilled the feature is also fulfilled and accepted. To define a scenario Given/When/Then are used. After each keyword there has to be a sentence about what is going on at the particular state. If the Given/When/Then vocabulary is not enough to define the behavior they can be combined with And/But. Every line after a keyword is called a step. It must be mentioned that the keywords are just for readability and does not make a difference for tools like Cucumber. Therefore there is also the star (*) to replace Given/When/Then and And/But. That the keywords are treated the same is due to the fact that every step has to be implemented by the developer. So it is up to the choice of the development team which keywords are used, but as Gherkin was designed to ease communication and specification it is a good idea to stick with the initial concepts [Way12, p. 32].

Naming of scenarios is even more important than naming of features, because there are always more scenarios than features. The name of a scenario should always summarize the steps and should be enough to know what's going on. The naming should not depend on the outcome because the outcome is likely to be changed. Instead it is better to describe the behavior.

Background

If there are some steps which are the same in every scenario they should be moved to one place. This can be accomplished by the background keyword. The background steps are always executed before each scenario, for example an authentication before visiting some restricted pages. The background is defined the same way as a scenario. If suddenly the steps in the background change the code and features only need to be adapted at one place. Another advantage is that the background steps do not pollute the scenarios. This means that the scenarios are still nicely readable and the common steps of all scenarios are not restated all the time. (Also the attention on them is put into background). A rule of thumb is to keep the background steps below 4 steps. The background should be simple enough to remember while reading all the scenarios. Also technical details like resetting a session or clearing some queue should not be mentioned in the background steps.

Scenario Outline

A scenario outline is used if multiple scenarios follow exactly the same steps just with different parameter values. For example log in on a web app. There could be several user credentials to test, but the scenario itself is always the same. Therefore an outline can be specified. This outline takes the values of an examples table. This is a table which is introduced by the keyword examples. A word of warning should be placed here. Do not use too many examples. This makes the test suite hard to maintain and slows down execution. Also it makes it hard to read scenarios. But readability is one of the most important things for behavior driven development. If all examples are necessary try to cover them in the underlying test suite like unit tests. Try to stick to key examples[23].

[23] As mentioned in the book Specification by Example. [Goj11, pp. 26-27]

Implementation

After all features are defined a developer has to implement certain scenarios of a feature. Cucumber treats every sentence after a Given/When/Then and And/But/* keyword as a step. The developer has to implement the behavior of every step. If every step is implemented and working a scenario will pass and therefore be marked as fulfilled. To get a scenario pass the developer should implement the code with unit tests. In behavior driven development they are called specifications. This is the inner cycle of the outside-in approach of behavior driven development (see Figure 3).

The inner cycle of the behavior driven development outside-in approach

While Gherkin and Cucumber are used to specify the high level behavior of a system, specifications are used to describe the behavior of single components. The comparison of acceptance tests[24] and unit tests[25] is often made with the following sentence:

> *"Building the right thing vs building the thing right"*

This is a great aphorism. For a high quality software both has to be assured. The product must meet the customers' expectations and it has to function well. The initial idea of behavior driven development was just to ease test driven development (compare with first paragraph of chapter 3). But with the problem of "building the right thing" there was more added. As Cucumber is a way to solve this problem, as was discussed in the previous section, it is time to have a look at the lower level components of behavior driven development.

As stated in the first paragraph of chapter 3 Dan North claims that the vocabulary of test driven development makes it hard for new adopters to make the most out of it. The concepts of test driven development are also used in behavior driven development but there are slight changes. The words test and assertion are radically stripped away. The focus while implementing specifications is also on readability. This is achieved by a describe/it syntax. This syntax is related closely to the Gherkin syntax. The describe-block should specify a behavior of a component. The it-block specifies a certain use case and expect-block specifies the outcome of a certain action performed on an object. While implementing a specification (in test driven development vocabulary this means making a unit test pass) there should be paid more attention to the technical details of the object which is under test. So these kinds of tests are more low level and the idea is that they help developers writing high quality of code. The specifications (unit tests) should assure that the thing is built right (a practical approach on this is explained in chapter 6.2.4).

3.6 Summary behavior driven development

Behavior driven development started as small improvement of test driven development and is now much more. It is not only about testing it is a lot about communication and specification. The focus on collaboration of all participants in a project (stakeholders, business analysts, developers) should be improved. This is due to the use of a ubiquitous language and the simple syntax in the used language. The Given/When/Then and the Connextra format templates aim to avoid ambiguities and misconceptions. The first premise is "to build the right thing" and the second premise is "to build the thing right". The second part is achieved by a traditional test driven development workflow but with different vocabulary. This tries to ease the first steps for developers which are new to test driven development. Also it should prevent common mistakes which often happen during test driven development.

---

[24] Automated features in behavior driven development

[25] Automated specifications in behavior driven development

A controversial aspect of behavior driven development is Gherkin. It is widely accepted that it make sense to define features in a ubiquitous language which minifies the risk of ambiguities. But on the other hand many people argue that it is overkill to use an automation tool for Gherkin like Cucumber. Their opinion is that a behavior driven development testing framework which supports the describe/it syntax is sufficient. Instead of adding another tool which must be maintained, learned and adopted by developers they suggest to stick to one tool and write specifications in it on the corresponding level of abstraction. For example the behavior driven development testing framework Jasmine can be used to define specifications at all levels of abstraction [Emr13, p. pos. 1904 eBook] (this will also be showed in chapter 6.2.4).

For Mocha (which is another behavior driven development testing framework for JavaScript, see chapter 5.2.1) there are approaches to gain the same results as when using fully fledged Cucumber [Amo13, p. slide 15]. Furthermore the critics of automated Gherkin argue that it is an over romantic assumption that customers will write features. Even if these features can be written in a simplified structured version of English (or any other spoken language). If the testing tools are used as described by these critics, the output reads like a Gherkin feature file. They believe that this is enough and that this output should be presented to business people and customers.

But automated Gherkin features also have benefits. One is that developers always have a feedback on how many features are finished and how many features are still waiting for implementation. It is not likely that a feature gets forgotten or omitted. But also for this issue the critics of Cucumber and Co provide solutions. The project Mocha-Gherkin[26] creates stubs for Mocha from feature files which produce the output in a Gherkin feature file style. Projects like this are not a replacement for Cucumber they are an alternative in certain situations. Most, if not the entire, complexity of tools like Cucumber are necessary. For instance these projects do not provide code reuse if step definitions are the same. Distilling test stubs from features can be helpful because there is instant feedback about how much work is already done.

Gherkin is handy for polyglot projects. This means projects which have to behave the same on different platforms and are implemented in different programming languages.

Even if Dan North claims that behavior driven development is a full agile methodology[27] [Nor09] there is no common sense about how this methodology is applied in real world projects. Many important aspects like team organization, code ownership, working hours etc are not defined and interpreted differently by different people [Emr13, p. pos. 1965 eBook]. It makes sense to incorporate behavior driven development into an existing software development methodology. For example it is easy to integrate behavior driven development into an extreme programming project.

3.7 Behavior driven development and extreme programming

This chapter provides a suggestion how the behavior driven development testing style can be incorporated into extreme programming. This makes sense because extreme programming is widely adopted and there are a lot of practical experiences. Furthermore extreme programming is well defined and the main principles are stabilized. In contrast behavior driven development is vivid and discussed. There is a lot of controversy about the main characteristics of behavior driven development [Sol11].

If someone wants to use behavior driven development as a full software development methodology there is no common sense about the core principles. So it leaves too much space for interpretation and this makes it

[26] Mocha-gherkin GitHub: https://github.com/mklabs/mocha-gherkin (retrieved 12.06.2014 13:20)

[27] “BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.“ - Dan North - [Nor09]

hard for a non-experienced team to get started with this methodology. The thing which is well defined and common in mostly all literature about behavior driven development is how to formulate specifications and how to organize and write tests. As test driven development is an integral part of extreme programming, behavior driven development can be incorporated quite well and in many use cases extreme programming can profit from this incorporation.

3.7.1 Planning the Game and Project Inception

These two steps are similar and can be merged together quite well without a lot of pain. The project inception of behavior driven development can be integrated into planning the game phase of extreme programming. Therefore the extreme programming parts can be extended by the behavior driven development parts. This means during "planning the game" the ubiquitous language could be defined. For an approximate project plan methods of behavior driven development could be used. These methods provide a good framework to avoid misunderstandings because of ambiguities in the spoken language. Defining stories in extreme programming and behavior driven development is similar (compare project inception with [Ken99, p. 73]). At this point the team has to decide if automated story testing should be applied. If yes the team can take full advantage of all the features which Gherkin provides. The story cards from extreme programming could be written as Gherkin feature files. But using automated features provides extra overhead and it has to be considered if the effort is worth it.

3.7.2 Metaphor

Behavior driven development lacks the definition of a metaphor. A metaphor is important in extreme programming and should not be neglected. It helps developers to stay focused and to not lose the track. Also for the customer it is valuable because it helps keeping the big picture in mind and do not get confused by technical terms. Metaphors could make use of the ubiquitous language defined with behavior driven development methods. The system metaphor could even be created while finding the ubiquitous language.

3.7.3 Testing and refactoring

This part is the part where extreme programming and behavior driven development have the most differences. If a team decides to incorporate behavior driven development into an existing methodology like extreme programming it makes sense to use the concepts of behavior driven development. Otherwise the whole behavior driven development paradigm will be blurred. And a sloppy applied methodology will not lead to the expected results (compare with chapter 2). The main distinction comes from the outside-in approach (therefore see the behavior driven development cycle in chapter 3.5.2).

Furthermore behavior driven development uses a different vocabulary to define its tests, but the main proposal behind these tests stays the same. The tests should drive development and design. The nice side effect is that the tests assure quality and provide an automated way to see if the system works as expected. With outside-in development it should be assured that the "right" thing is built. This means that the system solves the problems of the customer. But in behavior driven development also the phrase "building the thing right" is addressed. Therefore the inner cycle of the behavior driven development cycle is used. At this stage behavior driven development and test driven development are similar.

The behavior driven development community argues that the tests written by the outside-in approach are more valuable than the tests written by the inside-out approach. They think this comes from the different focus. At first there is the feature and then there are the technical details of it. They believe that this prevents developers from writing very technical tests and only thinking about nasty edge cases. Also they claim that

the outside-in approach is easier for developers new to test driven development and also easier if a team needs to implement a system with a new technology.

Executable features definitions are special for behavior driven development. Although this concept is controversial (bigger overhead, more abstraction, more layers in the tool chain) there are several benefits. One thing which should be additionally stated here is the fact that executable features could be a form of progress indicator. It gives developers instant feedback how much is already done and how much has still to be implemented.

Refactoring is an integral part of extreme programming and behavior driven development so there is no discussion on how and why to use refactoring. Both see a huge benefit in refactoring and try to encourage developers to do as much refactoring as possible. The confidence for refactoring is provided by a good test suite. Since both methodologies promote the test-first pattern the basis for effective and confident refactoring is there.

3.7.4 Coding

All the suggestions concerning coding are similar in behavior driven development and extreme programming. The differences are subtle. Often the same things are referred in different terms. For example simple design in extreme programming is covered in behavior driven development by the term “enough is enough”. Principles like YAGNI[28], KISS[29], DRY[30] apply for extreme programming and behavior driven development. These concepts are immensely important for both methodologies. While behavior driven development gives a great coding guideline for tests and specifications it lacks a suggestion for coding standards. This part is covered by extreme programming in detail so it makes sense to use these proposals. Practices like pair programming, collective ownership, continuous integration and working hours are not addressed in behavior driven development ether. This is why many people do not consider behavior driven development as a full software development methodology [Emr13, p. pos. 1965 eBook]. These parts are discussed in extreme programming in detail and can be adopted directly.

3.7.5 On-site customer vs customer writing gherkin

These two concepts are a good idea but in reality both are not always possible. To have a customer always on-site would be great for development but especially for projects with a smaller budget this is problematic. The extreme programming community argues that an on-site customer is not that expensive and that it is essential for project success. The behavior driven development community also addresses the problem of communication between customer and development team. To tighten cooperation between these two parties they suggest that the customer writes also features, preferably in Gherkin. Also this assumption has a great intention but rarely happens in reality. Because these are integral parts of these methodologies there are concepts to have a backup plan. Interestingly both solve this with the same mechanism. They suggest making small iterations and giving feedback as fast and often as possible to the customer. This works so well that project success does not only depend on an on-site customer or a customer writing gherkin.

---

[28] YAGNI: “you ain’t gonna need it”, http://en.wikipedia.org/wiki/YAGNI (retrieved 07.07.2014 15:49)

[29] KISS: “keep it simple, stupid”, http://en.wikipedia.org/wiki/KISS_principle (retrieved 07.07.2014 15:49)

[30] DRY: “don’t repeat yourself”, http://en.wikipedia.org/wiki/Don%27t_repeat_yourself (retrieved 07.07.2014 15:51)

3.7.6 Conclusion

Although behavior driven development is not a complete software methodology it definitely makes sense to investigate this development style. It tries to improve test driven development. This is obviously the main field where it makes sense to use behavior driven development. How to apply it as a whole methodology is not clear and it leaves too much space for interpretation. Therefore combining behavior driven development with a well known agile methodology like extreme programming is a viable option. Especially because many core principles are similar, so there is not a big danger of opposite opinions. Also communication and feedback are promoted heavily by both. In this area behavior driven development even goes one step further and provides a way to ease communication between all stakeholders, for example the ubiquitous language and language templates for defining specifications (features). Behavior driven development and extreme programming think that communication is crucial. Behavior driven development also argues that it is important how all participants communicate. It is important that everybody has the same understanding of what was defined and agreed on. Language ambiguities should be abstracted as much as possible.

Behavior driven development and extreme programming do not exclude each other. In contrast they can enhance each other. Because extreme programming is more stabilized, clearer defined and covers a bigger spectrum as behavior driven development it is a good choice to adopt extreme programming and refine it with parts from behavior driven development.

4 Web engineering

Web engineering is a new software development discipline which tries to build complex software which is available online. Because of the new trends in web technologies it is possible to implement feature rich, interactive and real time applications with it. This chapter will give a brief overview about what web engineering is and which tools can be used to develop web apps.

4.1 Web app development

At first it is important to distinguish the terms web app and website. Because these two terms serve different needs there are diverse requirements. The main purpose of a website is, to present content to a user. Let's consider a website of some sports club. This website should for example present the list of players, next events and the history of the club. It is built around the concept of providing information to the user and not around the interaction with the user.

In contrast a web app can be viewed as software running in a browser. It helps a user to solve some problem. This sounds very generic so let's consider some examples. Google provides many web apps which can be used as a fully functional office suite. All this apps are running in the internet browser. For text processing, spreadsheets, presentations and online forms there are apps available.

Another impressive example is Cloud9 IDE[31]. This application is a full IDE running in the browser. The look and feel is similar to all the well known IDEs which have to be installed on the computer. Cloud9 IDE makes it possible to program from everywhere on the world and from every computer.

Because web apps can replace some desktop applications users often compare these two. Therefore a web app has to be more reactive and snappy than a traditional website. Furthermore user interfaces and especially user experience are getting more and more important and have to be fascinating.

The traditional architecture of web apps does not fit to today's needs anymore [Mik13, pp. 3-9]. To process everything on the server and then send it to the user is too slow. Not because of the servers, the bottleneck is the roundtrip time over the network. To reduce the network delay there is a shift to single page applications. These kinds of applications only require one big initial page load and after that only small packages are requested from the server. As much program logic as possible is transferred to the client.

4.2 Single page applications

Web apps use the HTTP protocol to communicate with the server. The HTTP protocol is stateless, but in every application there will be a certain state after some user interaction. Let's consider text processing software. The user starts it, so the software is in its initial state. Then the user types something and the software is in another state. If the user changes the view the application hits the next state and so on and on. The first web apps tracked this state on the server. So the server had to do all the heavy lifting. This architecture is called thin-client/fat-server [Cou01, p. 39].

The client only sends a HTTP packet to the server and receives processed HTML code. The server uses a session cookie on the HTTP request to identify the user. The state of the application is stored according to every user. Together with the HTTP request the server calculates the state of the application and compiles all

[31] https://c9.io/ (retrieved 21.05.2014 10:05)

things together to one single HTML file. This approach has the advantage that the client is not in possession of the application's program code.

This has two main advantages. It is easier to secure the application and the developer does not need to care about different browsers. (This is true for the program logic, not for the rendering of HTML/CSS). But the thin-client/fat-server model does not fit the need of modern web apps well.

The disadvantage is that for every interaction the whole HTML page has to be recalculated and resent from the server to the client. So the network delay increases and a reactive user interface, as someone is used to from desktop applications, is almost impossible to implement.

Therefore the architecture shifted to fat-client/thin-server. This shift was driven by the fact that modern browsers support AJAX and WebSockets. A single page application does one initial page load from the server. Mostly this is when the user opens the web page the first time. With this page load all the needed code is transferred from the server to the client. The application state is handled by the client and for every user interaction the client processes the event.

If necessary, the client sends a request to the server and the server response only with the needed data. It is not necessary any more to send always the whole page from the server to the client. So the network delay decreases and a more reactive application can be implemented.

It is common to implement the server as REST[32] API. The client communicates with the server over HTTP and as data exchange format JSON is often used. Nowadays it is common that the client code is in JavaScript. This is due to the fact that almost every browser supports JavaScript and the incompatibilities between the different JavaScript engines are getting less.

With single page applications there appear new challenges and someone will soon recognize that it is not possible to switch the whole business logic from the server to the client. Especially security relevant parts have to stay at the server for example authentication[33] and validation of the data which is sent to the server. Figure 4 should give an idea how the architecture of a traditional web app and a single page application differs. The thickness of the rectangle should approximately indicate the amount of code needed for each part.

---

[32] REST means "Representational state transfer", it uses the HTTP request methods GET, POST, PUT and DELETE and is build around the concept that every URL corresponds to a resource. For a quick intro checkout Wikipedia: http://en.wikipedia.org/wiki/Representational_state_transfer (retrieved 10.07.2014 16:46)

[33] Also for authentication there is a technique to minimize the effort on the server. It is called "token based authentication".

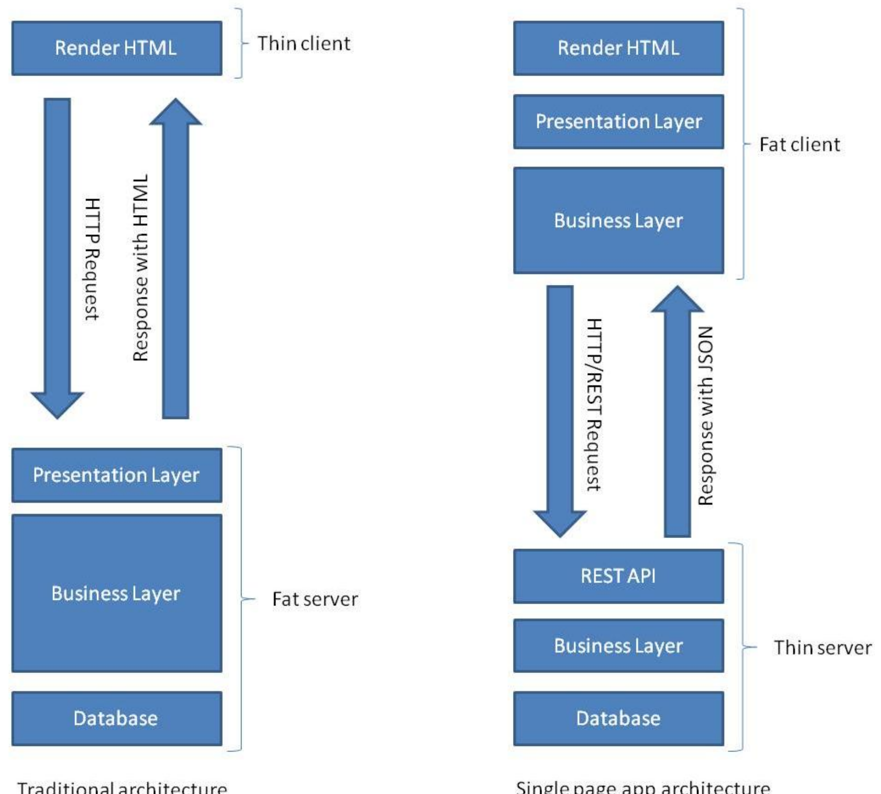


Figure 4: Comparison of traditional web app architecture with the single page app architecture (inspired by[34] [35])

For many single page apps it is enough to accomplish the initial page load and then only request data from the server. This also frees the server from many computational intense tasks. With a simple serve it is possible to process more requests at a given time. This is immensely important for cloud base services. This reduces the costs of hosting such a service because often the payment is calculated in computation time and resource consume.

The current trend of cloud based services and software as a service will increase the amount of single page applications fast. Single page apps are new compared to other technologies and so there is a lot going on at this topic at the moment. It is a very exciting time for web developers right now.

[34] [Kni13, p. retrieved 21.05.2014 12:54]

[35] [Mik13, p. 8]

5 JavaScript

Because JavaScript is the only language which is understood by almost every web browser it is the most important language for web development. JavaScript has strength and weaknesses as we will see in the following chapter. Because the community is huge there are a lot of tools to address JavaScript's weaknesses but also to leverage its strengths.

5.1 The rise of JavaScript

For developing web apps JavaScript is ubiquitous nowadays. Interactive applications especially rely on JavaScript. JavaScript was long considered as toy language and was not taken seriously. Many people still think this way. But JavaScript was the only language which survived the so called "first browser war"[36] between Netscape's Navigator and Microsoft's Internet Explorer. During the early 2000s the main purpose was to enhance the GUI with little effects. JavaScript was mostly used in a "decorative" way. This was due to the fact, that the JavaScript engines of the browsers were very diverse. It was not state of the art to implement business critic logic in JavaScript. Many developers used technologies which were based on browser plugins, for example Java applets or Flash. But they also did not turn out as great as desired. So the majority of the web developers stayed with JavaScript.

2004 Google launched Gmail and 2005 Google Maps. With these web apps Google demonstrated the possibilities of JavaScript. This inspired many other companies and developers to implement highly interactive applications also with JavaScript.

To overcome the browser incompatibilities there were written many different libraries. The most famous one is jQuery[37]. This library was immensely important for the spread of JavaScript because it helps developers to abstract the differences between the browsers and gives them a convenient way of accessing elements of the DOM. Even for non technical users it's quite easy to write small scripts in jQuery. The selectors to access the DOM are based on CSS selectors; this makes it easy for a broad audience of people.

Also around this time the term AJAX appeared (Jesse James Garrett was one of the first who coined the term[38]). AJAX is a technology to make asynchronous calls to a server. This give a JavaScript program the possibility to fetch data from a server after the HTML page was loaded. AJAX was an important step on the rise of JavaScript. With AJAX it became possible to shift more and more programming logic from the server to the client.

The next big step forward was Google's V8 JavaScript Engine[39]. It gave JavaScript a performance boost. V8 is released with under BSD license, so it opens new opportunities to use JavaScript in different places like on the server (for example node.js[40]). JavaScript is also used for databases for example mongoDB[41] or couchDB[42]. These databases store data as a JSON format and provide a JavaScript API. This makes

[36] http://en.wikipedia.org/wiki/Browser_wars#The_first_browser_war (retrieved 20.05.2014 11:27)

[37] http://jquery.com/ (retrieved 20.05.2014 11:46)

[38] http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ (retrieved 20.05.2014 11:55)

[39] https://code.google.com/p/v8/ (retrieved 20.05.2014 12:00)

[40] http://nodejs.org/ (retrieved 20.05.2014 12:17)

[41] http://www.mongodb.org/ (retrieved 20.05.2014 12:44)

[42] http://couchdb.apache.org/ (retrieved 20.05.2014 12:45)

JavaScript available for the whole development stack. From the database to the server and the client code, everything can be written in JavaScript.

The rise of JavaScript is driven by the fact that JavaScript is the de facto standard to write code for the browser. Despite the fact JavaScript was not designed for the things it is used now, and therefore has also a lot of weaknesses, it is one of the fastest growing programming languages. The paper “Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects” [Bis13] comes to the conclusion that JavaScript is one of the most important and popular languages at the moment. Also JavaScript has one of the biggest communities of all programming languages. Stephen O'Grady from RedMonk[43] did an analysis [OGr14] about this and compared the interactions on Stackoverflow[44] and the projects on GitHub[45]. This publication also claims that JavaScript was one of the most important programming languages during the last years. In the next years the language will maybe become even more important, as the trend to software as a service and cloud computing is still ongoing. Most of these upcoming services will be written as browser application and so it has to be JavaScript in the end. Because JavaScript is so widespread there emerged also some “odd” projects. For many people it sounds awkward to use JavaScript for hardware projects, but Espruino[46] and Tessel[47] make JavaScript also available for micro controllers. The community around JavaScript is vivid and it is exciting which new projects and technologies using JavaScript will emerge in the future.

5.1.1 Strengths

JavaScript is the language of web browsers; therefore it is widely spread. The community is huge and provides many libraries and frameworks. Most of them are hosted on GitHub and everybody is invited to collaborate. Because there is no choice at the client side, all developers have to implement their applications in JavaScript (if they use a Transpiler, this is slightly different, but the resulting code is always JavaScript). In contrast on the server side there are many languages to choose from. This makes JavaScript interesting because people with different backgrounds have to collaborate in the same language, so people familiar with PHP, Ruby, Java, Asp.net etc bringing their knowledge into JavaScript projects. There is a huge interchange of knowledge going on in JavaScript. Because there are so many people writing JavaScript code, best practices and the workflows are constantly evolving.

Furthermore JavaScript is easy to get started with. Someone who wants to learn JavaScript just needs a browser. There is no need to setup a compiler and tool chain. The feedback, if the code works as expected, is fast and motivates beginners to try out new things and play around. So very little prevents people from writing in JavaScript code and so many people can start to use it. This enables the community to grow and a big and active community is essential for every programming language.

It is controversial if the loose typing of JavaScript is beneficial or not. On one hand strong typing would make the entrance level to JavaScript higher but on the other hand code could be better maintainable with strong typing. For web development the dynamic nature of JavaScript is not always bad. It avoids type casts, which can be quite annoying when dealing with form data. Furthermore it is not necessary to build huge class hierarchies. There are pros and cons but for small projects the benefits of loose typing seems to overweight the disadvantages. The dynamic nature allows writing less boilerplate code and therefore lets developers get work done quickly. For bigger projects this question has to be answered from a broader view.

---

[43] http://redmonk.com/ (retrieved 20.05.2014 14:33)

[44] http://stackoverflow.com/ (retrieved 20.05.2014 13:41)

[45] https://github.com/ (retrieved 20.05.2014 13:42)

[46] http://www.espruino.com/ (retrieved 20.05.2014 14:47)

[47] https://tessel.io/ (retrieved 20.05.2014 14:47)

Loose typing also means less tooling support and bigger challenges on the maintainability side (but there are also possibilities to solve these issues, and one of them is described in section TypeScript).

Functions are first class citizens in JavaScript. This is unusual for many developers but powerful. Things like lambdas and closures have been well known in JavaScript for a long time and many other programming languages start to adopt these features[48]. Prototypal inheritance may seem awkward to classical object-oriented trained programmers but it fits good the dynamic nature of JavaScript. It is flexible and powerful and allows a lot of new design patterns. With the current standard the syntax of the inheritance mechanism is not very well and considered problematic. The next standard (ECMAScript 6) will harmonize and improve this syntax. These features can be used also today (therefore see chapter Traceur Compiler and TypeScript).

JavaScript is asynchronous which makes it great for GUI development. It eases the programming of user interfaces also for non experienced programmers. They do not need to learn complex threading models; they just need to understand the asynchronous nature of JavaScript. This language feature is also used by projects like node.js. They rely heavily on the asynchronous functions, especially on I/O based operations.

Another strength of JavaScript is its data and object marshalling format which is called JSON[49]. The definition of JSON is the same as a JavaScript object, this makes it easy to consume JSON data and use it in the application code. JSON is often used as format to exchange data between web services. For this purpose JSON has the advantage over XML that it uses less overhead. But JSON is not a replacement for XML. JSON is just a data exchange format and XML is a fully fledged markup language. Because of the wide spread adoption of JSON there exists almost for every modern programming language[50] a way to import JSON data and make it accessible.

Full stack JavaScript

A new trend is to use JavaScript as full stack language. This means that JavaScript is at every layer of the application. For the client, server and database code JavaScript can be used. Even the whole tool chain and workflow can be automated with JavaScript and test automation is of course also available for JavaScript. One programming language at the whole stack eases development a lot. Programmers do not need to switch their mind all the time.

---

[48] For example PHP, which adopted Closures and Lambdas with version 5.3

[49] JSON: JavaScript Object Notation

[50] For example PHP and json_decode (http://www.php.net/manual/de/function.json-decode.php, retrieved 08.07.2014 16:02)

For example, for-in loop in JavaScript and foreach loop in PHP. They are similar as Listing 3 shows and a developer could get confused quickly.

```
// JAVASCRIPT FOR-IN LOOP
var obj = { hello: true,
            world: true,
            end: false };
for(var key in obj) {
  if(obj[key]) {
    console.log(obj[key]);
  }
}
// PHP FOREACH LOOP
$obj = new stdClass();
$obj->hello = true;
$obj->world = true;
$obj->end = false;

foreach($obj as $key => $value) {
  if($value) {
    echo $key;
  }
```

Listing 3: Comparison of JavaScript for-in and PHP foreach

For small teams and startups this could be a big problem. Because they do not have the capacities to split their team into special task forces which only program in a particular language. If a team decides to use the LAMP stack it has to manage four different environments. For development the M and P acronym are interesting. M means MySQL and P refers to PHP. On the backend the team has to learn SQL and PHP, for the front end they need to learn JavaScript. This means the developers have to be fluent in three different languages. Additionally often XML is used as data exchange format and this also has to be understood by the programmers.

If your team is split on the boundaries of the languages there are also disadvantages. For example the ability to reuse already existing code is lost. An example would be data validation. For a snappy user experience the validation has to be performed in JavaScript at the client. For security reasons the data has to be validated again on the server. Therefore the same logic has to be implemented twice, once in the language of the server and once in JavaScript. Only if server and client use the same language this code can be shared and reused. Tools like Browserify[51] help to achieve this goal.

MEAN

One popular full stack JavaScript project setup is MEAN. This acronym stands for MongoDB, Express.js, Angular.js and node.js. Projects like mean.io[52] help scaffolding and additionally provide ready to use packages.

Node.js

The revolution of using JavaScript on the server was driven mostly by node.js. Node.js is often referred as a JavaScript-server but it is much more. It is an entire ecosystem built on JavaScript. It is possible to write any kind of application with it. There is also a node package manager which is called npm and makes it easy to install third party libraries and frameworks. Npm also takes care of package management. There are over

[51] http://browserify.org/ (retrieved 08.07.2014 16:42)

[52] http://mean.io/ (retrieved 08.07.2014 16:56)

80.000 packages[53] available for almost every use case. This frees developers from always implementing the same things again. They can focus on the business value of the application they are working on.

Node.js makes use of Google's V8 JavaScript engine and therefore is very fast. For using node.js as web server the asynchronous nature of JavaScript is ideally. Most of the computation time is lost on I/O operations. These operations are asynchronous in node.js and therefore they do not block the whole process [Rod12, pp. 11-12].

Now let's consider the E in MEAN. It refers to express.js. This is a framework especially designed to develop a web server using node.js. There are several alternatives but express.js is a widely adopted framework. To use it node.js has to be installed. With npm express.js is loaded and afterwards usable for programming.

```
# Install express.js:
$ npm install express --save
```

Listing 4: example of how easy it is to install a package with npm

To start up the express.js server the following code is enough:

```
var express = require('express');
// REQUIRE IS A NODE.JS WAY OF MANAGING JS MODULES

var app = express();
app.get('*', function(req, res) {
  res.contentType('text/html');
  res.send(200, 'Hallo World');
});
http.createServer(app).listen(3000);
```

Listing 5: example of a very simple express.js server listening on http port 3000

This server is not doing much. It is just a toy example to illustrate how easy it is to get started. For further details there are tons of examples on the internet.

The M stands for MongoDB. MongoDB belongs to the movement of NoSQL[54] databases. They try to solve the problems which arise with huge web apps in a different way than SQL databases. Also the approaches inside the NoSQL universe are different and reaching from simple key value stores to document stores. MongoDB is a representative of the latter one. Documents in MongoDB are JSON objects; this makes it easy to use the stored data in a JavaScript application. Calls to the database can be made via JavaScript and also queries are defined as JSON objects.

```
// Saving a new user
db.users.save({name: "georg", password: "secret" });
// fetch users with name georg
var user = db.users.find({name: "georg" });
// The object fetched from DB looks as follows:
// {
//   "_id" : ObjectId("53bc14a01cdcaf0728738090"),
//   "password" : "secret",
//   "name" : "georg"
// }
```

Listing 6: example of how to save and fetch data in MongoDB

[53] Amount of packages at 08.07.2014 source: https://www.npmjs.org/ (retrieved 08.07.2014)

[54] NoSQL means not only SQL, http://en.wikipedia.org/wiki/NoSQL (retrieved 08.07.2014 17:25)

As one can see in Listing 6 syntax and data format of MongoDB integrate seamlessly into JavaScript. This is why MongoDB is often the first choice when considering a full stack JavaScript development environment.

The A in MEAN stands for Angular.js. It is a front end JavaScript framework which helps to built single page applications. There are several different choices like backbone.js, ember.js, knockout.js and more but Angular.js stands out of the crowed. Because this thesis focuses on front end development Angular.js is discussed in detail in chapter 6.

Conclusion about MEAN

The MEAN stack is a trendy setup at the moment. But there exist plenty of alternatives for each layer of this stack. MongoDB could be exchanged with CouchDB, express.js by geddy.js. For Angular.js there are many other options as mentioned before. Only for node.js there is no serious competitor as a JavaScript runtime environment.

It does not have to be the MEAN stack if a team decides to use JavaScript on the whole development stack, there are many choices out there. But MEAN is popular and there are many generators and project templates which one can use. This reduces the effort to get the project started and minimizes boilerplate code. There are several sources for these templates but tools like yeoman.js[55] help scaffolding new projects. Projects scaffold with generators often play nicely with workflow tools like Grunt.js[56] or Gulp.js[57].

Project setups like MEAN do not try to blur the border between server and client. In contrast they try to shift as much logic as possible to the client and keep the server side code as small as possible. There are also project like Meteor.js[58] which try to make the gap between server and client totally transparent. If this approach will displace the others is not clear. This has to be observed during the next years.

As one can see the JavaScript ecosystem grew up and offers a wide range of choices for every development need. It is important to evaluate every third party project before using it. This comes from the vivid nature of the JavaScript community. What's hyped today could be possibly dropped tomorrow. It's better to investigate and make your own choice than to follow every hype.

5.1.2 Problems

Even if the ecosystem around JavaScript grew up and offers many tools to develop high quality software, still not everything is perfect. One of the main reasons for problems is the fact that the language was not designed for being a multipurpose language which can be used for browser, server, database and even hardware. It also was not intended to be used for huge software projects with thousands and thousands of lines of code. Especially maintainability and working in teams can become difficult quickly. So testing turns out to be even more essentially in JavaScript projects than in other projects.

---

[55] Yeoman.js is a scaffolding tool for JavaScript projects, more information on http://yeoman.io/ (retrieved 09.07.2014 12:31)

[56] Grunt.js is a JavaScript task runner, https://github.com/gruntjs (retrieved 09.07.2014 12:33)

[57] Gulp.js is an alternative to Grunt.js, for more details visit their GitHub repository: https://github.com/gulpjs/gulp/ (retrieved 09.07.2014 12:25)

[58] Meteor.js is a full stack JavaScript development setup which tries to make the gap between server and client totally transparent, more information on their website https://www.meteor.com/ (retrieved 09.07.2014 12:43)

Also defining a coding guideline and sticking with it is indispensable (as this is suggested by extreme programming, see chapter 3.7.4). Big companies like airbnb published their guidelines to the community and asked for feedback. The guidelines can be found on their GitHub repository[59].

Douglas Crockford wrote the book “JavaScript: The Good Parts” [Cro08] to emphasis the strength of JavaScript and how to avoid its pitfalls.

Figure 5: Comparsion of JavaScript the good parts vs JavaScript the definitive guide (Source: http://www.laurencegellert.com/2012/03/javascript-the-good-parts-review/, retrieved 09.07.2014 14:12)

Figure 5 illustrates drastically how much of JavaScript is considered as good by Douglas Crockford. This is not because he dislikes JavaScript. In contrast, he really likes it but he thinks that it is better to stick to the basic concepts and only use language feature which can not cause unexpected side effects.

To prevent developers from experiencing strange behavior in JavaScript programs he developed the tool called JSLint[60]. This tool checks a given source code for “as bad considered” parts. These checks are performed as static syntax analysis. The errors are then presented to the developer. There is also a fork of JSLint which is called JSHint[61]. It also performs static code analysis but it is not as opinionated as JSLint and can be configured.

So to assure quality there are powerful tools for JavaScript, which are testing and linting/hinting. Because JavaScript is loosely typed it is hard to implement tooling. There is a lot of progress going on at this area but we are still in an early stage. So it is still not trivial to write large maintainable JavaScript programs. Most of the time, the advice is, to not write huge applications and split it into several small projects instead. This is not always possible and better tooling is enormous important for JavaScript. The main IDEs are getting better and better and in the near future tooling will be maybe satisfying. To solve some of the problems which come with JavaScript there are a lot new projects arising. There are transpilers like CoffeeScript or TypeScript and other projects like Dart which want to replace JavaScript on the long run. Because there are many different ways to cope with the weaknesses of JavaScript some people consider JavaScript as the assemble language of the web [Han13]. Before starting to write huge software in JavaScript it is essential to consider also the weaknesses of JavaScript and how to address them. There are many different ways and there is no common best practice. For every project there are different requirements and therefore everything from vanilla JavaScript to high abstractions like Google Web Tool Kit could be a viable alternative.

Transpiler

A transpiler is a program which translates one programming language into another programming language. There are many “to-JavaScript” transpilers out there. A detailed list can be found on GitHub on the following

---

[59] https://github.com/airbnb/javascript (retrieved 20.05.2014 14:58)

[60] http://www.jslint.com/ (retrieved 20.05.2014 15:20)

[61] http://www.jshint.com/ (retrieved 20.05.2014 15:19)

repository[62]. Depending on one's needs, one has to select the right tool. The different alternatives will be discussed in brief. The rest of the thesis will concentrate on vanilla JavaScript. Because if someone understands the concepts it is easy to use some transpiler for her/his project. Especially in conjunction with Angular.js it is quite simple to introduce a transpiler to the workflow. Also there is an enormous community which is always willing to help.

CoffeeScript

CoffeeScript is heavily influenced by Ruby, and Python. The main goal is, to provide the developer a different syntax. CoffeeScript translates to regular JavaScript and uses common JavaScript design patterns. With CoffeeScript it is possible to write more compact code than in vanilla JavaScript. Also it tries to harmonize the different possibilities of doing the same thing in JavaScript. For example there is one standardized way of creating classes. In JavaScript there are more than 3 ways. It is also easy to convert an existing JavaScript project to CoffeeScript. There is a converter called js2coffee which can be utilized for that. CoffeeScript makes most sense for teams which are already writing code in Ruby or Python, because CoffeeScript is more similar to these languages than JavaScript.

Traceur Compiler

ECMAScript defines the standard of a client side language. There are several implementations of this standardization and the most famous one is JavaScript. Because the developers' community wants new features there is a new standardization ongoing. It is called ECMAScript 6 and should bring these new features. Furthermore ECMAScript 6 should remove weaknesses of the older standards. The aim of ECMAScript 6 is to ease the development of large scale apps. Therefore there is a new syntax for classes and modules. Because standardization is a time consuming task and it is hard to predict when the standard is finished it could take a long time to have all these new features available. Also the adoption of browsers takes a lot of time and so it is not possible to use an ECMAScript 6 implementation for a web app at the moment. But if a development team wants the features now there is a compiler called Traceur. Traceur is a project by Google and can be found on GitHub[63]. What this compiler does, in basic terms, is translate JavaScript written in ECMAScript 6 to JavaScript which follows the ECMAScript 5 standard. The compilation can be included in the workflow or can be done in the browser. Therefore Traceur Compiler has to be included in the web page. The disadvantage of this approach is that the code needs to be interpreted twice. Once by Traceur and then by the browser.

TypeScript

TypeScript is a relatively new project powered by Microsoft. The 1.0 version was released 2nd of April 2014 but minor versions appeared already in 2012. The lead architect of the language is Anders Hejlsberg. He was also the inventor of C#, Delphi and Turbo Pascal. This makes the language quite promising. Microsoft published TypeScript under the Apache License 2.0. So the language is open source and is not tied to Microsoft. TypeScript defines itself as a superset of JavaScript, which means that every JavaScript program is also a TypeScript program. TypeScript adds optional static typing to JavaScript and harmonizes the class syntax. Building class hierarchies is similar to the concepts of classical object oriented languages. This decision was made because many developers have problems with prototypical inheritance. Also there are more developers who are trained to the classical object oriented concepts than developers trained to prototypical inheritance. All these adoptions of TypeScript try to make building large JavaScript applications easier and more maintainable. Especially for large teams static typing can be a great help. This feature makes tooling better and more efficient. Things like code completion, type hints, and compile time errors are helping developers. There is TypeScript support in Visual Studio, Sublime, Webstorm and Eclipse. With these IDEs it is possible to leverage the full strength of TypeScript. For many of the popular JavaScript libraries there are type definitions. Several projects by the community provide types to various libraries. One

---

[62] https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS (retrieved 20.05.2014 16:25)

[63] GitHub repository of Traceur Compiler: https://github.com/google/traceur-compiler (retrieved 26.05.2014 16:30)

of the biggest of its kind is DefinitelyTyped[64]. With these definition files TypeScript integrates seamlessly into the existing JavaScript world. This makes it interesting, because it is possible to use all the existing work which the JavaScript community already did. TypeScript is clearly not a replacement for JavaScript but it could be useful for large scale web app development. TypeScript is compiled to regular JavaScript. The code is aligned to JavaScript best practices and design patterns. Therefore the produced JavaScript code is executable in every modern web browser. No plugins or something else is needed for the user. Of course a compile step has to be added to the build process. TypeScript wants to stay as close as possible to ECMAScript 6 and align their language to the ECMA standard[65].

Asm.js

In contrast to TypeScript asm.js is a subset of JavaScript. So every script written in asm.js is JavaScript but not every JavaScript is asm.js compatible. Mozilla created the asm.js project. The main purpose is to make very computational and resource intense applications available for the browser. This is used for example to port 3D games to the web browser. Every language which can be compiled to LLVM[66] can be used to produce asm.js compatible JavaScript. Therefore many C/C++ applications can be ported to asm.js. One famous example is the Unreal Engine 3 which was ported to asm.js and is now fully executable in the browser utilizing JavaScript and WebGL [Bru13]. The asm.js JavaScript code is highly optimized code and will run fast in every browser. But if the browser knows that the script is written in asm.js and the browser supports asm.js the performance is even better. This performance gain is made due to the design of the asm.js syntax. With this syntax the JavaScript code does not need to be reinterpreted. It can be translated to assembly directly. The asm.js implementation of Firefox is only two times[67] slower than the same program written in native C. Right now asm.js is interesting for graphic applications, because the WebGL interface can be used well. For conventional web apps there is a lack of DOM interaction [Res13].

Dart

Google invented Dart as an alternative programming language for web development. It is designed to be used at the server and at the client. Therefore the Dart team developed a virtual machine. Right now the Dart VM is only available in an experimental version of Chrome called Dartium. But the Dart VM should come to Chrome as soon as possible (according to the statements of the Dart team[68]). Google wants to achieve two main goals with Dart. On one hand they try to speed up web apps and on the other hand they want to enhance developers' experience. Google claims that it is not possible to improve performance of web apps written in JavaScript significantly anymore. (Of course better algorithms in an app will always improve performance but this applies to every program no matter what language is used). With the Dart VM they hope to bring bigger performance gains to web apps. Because there is no big adoption of the Dart VM and it is unsure if other browsers than Chrome will ever implement Dart VM there is a tool called dart2js[69]. With this tool Dart code can be compiled to regular JavaScript and therefore every app written in Dart is executable in all modern browsers. The dart2js tool applies optimizations and the resulting JavaScript code in some cases is

[64] For more details visit: https://github.com/borisyankov/DefinitelyTyped (retrieved 23.05.2014 10:40)

[65] To follow the aligment process visit: http://typescript.codeplex.com/wikipage?title=ECMAScript%206%20Status (retrieved 23.05.2014 10:58)

[66] Low Level Virtual Machine, for more details visit: http://en.wikipedia.org/wiki/LLVM (retrieved 23.05.2014 11:33)

[67] To see a benchmark for Box2D visit: http://www.j15r.com/blog/2013/04/25/Box2d_Revisited (retrieved 23.05.2014 11:54)

[68] FAQ question: "Will the Dart VM get into Chrome" https://www.dartlang.org/support/faq.html#q-will-the-dart-vm-get-into-chrome (retrieved 26.05.2014 10:25)

[69] The documentation of dart2js can be found here: https://www.dartlang.org/tools/dart2js/ (retrieved 26.05.2014 10:35)

even faster than handwritten JavaScript code[70]. The other main goal Dart is trying to achieve is to ease the development of large scale web apps. Therefore Google provides Dart developers with a good tooling infrastructure. Because of the feature "optional typing" the tooling should work better than with vanilla JavaScript. Also the class syntax and the possibility of building class hierarchies was made more similar to classical object oriented languages like Java. Google puts a lot of effort into Dart but it is still uncertain if Dart will come to browsers natively or if it will be just another option of JavaScript transpiler. The project is ambitious and shows how web app development could be. But history told us that new web technologies (especially on the browser) are only successful if all of the major vendors support it. Otherwise it is just another plugin with lose adoption. So the future of Dart is unpredictable.

Google Web Tool Kit

For the sake of completeness Google Web Toolkit has to be described here also. This thesis will not go into too much detail about Google Web Toolkit because this is another huge topic. The Google Web Toolkit enables the developer to write the whole front end code with Java. The Java-Code is then transpiled to JavaScript and is also executable on every modern web browser. The main advantage is that it is possible to use all the tooling build for Java. These tools can help to increase speed of development and assure quality. Also it is easier to find developers which are trained on Java and classical object oriented concepts than JavaScript developers. The strict typing of Java makes working on large teams easier but it also has the drawback that you need to write a lot of boilerplate code. There is a huge discussion if Google Web Toolkit is a good technology for future projects especially since Google announced Dart. Many developers who were working on Google Web Toolkit were shifted to the Dart project[71]. Also Google left Web Toolkit as leading authority and change it to a fully open source project[72]. Many commentators judge these two steps as sign of death of the Google Web Toolkit but this scenario is definitive over dramatic. Vaadin made a survey and stated that there is a strong and active community. Also Google Web Toolkit is used widely in enterprise applications with more than 500.000 lines of code. These companies have a huge interest in keeping Google Web Toolkit alive and improve it further. A code base of 500.000 lines cannot be ported to some other technology easily. Also Google Web Toolkit developers are happy with this tool and feeling productive [Vaa12][73]. So Google Web Toolkit is still a great option for building large scale web apps.

Conclusion on JavaScript alternatives

There is still no real alternative to JavaScript in the browser but for the development process there are a lot of different options. At the moment they all just compile to regular JavaScript and therefore it makes sense for every developer to understand at least some basics of JavaScript. How far the abstraction of JavaScript should be cannot be answered generally. This depends heavily on the project and on the developers available. If your team does not have specialized and well trained JavaScript developers a layer of JavaScript abstraction can help the project succeed. Despite the hype around Node.js and many blog post suggesting that it is no problem to write large JavaScript apps it is not that easy in the real world. For small projects and prototypes vanilla JavaScript is great but for large scale application one should consider some of the

---

[70] A detailed explanation is available at the Dart FAQ:
https://www.dartlang.org/support/faq.html#q-how-can-dart2js-produce-javascript-that-runs-faster-than-handwritten-javascript (retrieved 26.05.2014)

[71] Therefore see the blogpost:
http://googlewebtoolkit.blogspot.co.at/2011/11/gwt-and-dart.html (retrieved 26.05.2014 11:25)

[72] More details available on the blog:
http://googlewebtoolkit.blogspot.co.at/2013/07/gwt-news.html (retrieved 26.05.2014 11:27)

[73] The report is available online:
https://vaadin.com/blog/-/blogs/the-future-of-gwt-report-2012 (retrieved 26.05.2014 11:44)

JavaScript transpiler mentioned above or from the almost complete list which can be found at GitHub[74]. For large applications which are built for long term some kind of typing is definitely helpful. Also when there are many developers typing makes sense. Because typing forces the team to agree to some specified interfaces. Especially for long term applications tooling should not stay unconsidered. A very opinionated suggestion would be to choose one out of these three:

- TypeScript: pragmatic and easy to integrate with existing JavaScript code
- Dart: very new and exciting
- Google Web Toolkit: mature, many proofs of concept

As mentioned above this is just an opinionated suggestion about the experiences I made during the practical part of this thesis and during many other web projects. If you ask somebody else she or he will tell you something different. But that's natural to discussions about technology and programming languages. Because Angular.js is so popular there is almost for every transpile option a port or a tutorial how to use Angular.js for the chosen transpiler, e.g. Angular.dart[75], AngularGWT[76] or Angular with TypeScript[77].

5.2 BDD in JavaScript

Despite the methods of abstracting JavaScript it is essential to do testing. Not only because of the paradigm of test driven development. JavaScript programs need a special high amount of testing. This comes from the nature of JavaScript as a dynamic typed language. Because it is dynamic there is no compiler which can warn a programmer about possible errors. Furthermore the different JavaScript engines can be causing problems. Fully functioning code in browser A can be buggy in browser B and vice a versa.

Testing with the behavior driven development syntax is widely spread in the JavaScript community. Not every project uses the whole spectrum of behavior driven development as described in chapter 3. Many projects use Jasmine or Mocha/Chai/Sinon as testing framework. These testing frameworks implement the basic syntax of behavior driven development. There is also a library called Cucumber.js[78] which brings the whole cucumber methodology to JavaScript.

5.2.1 Jasmine vs "Mocha/Chai/Sinon"

There are many JavaScript testing frameworks available on the web. But for behavior driven development two stand out of the crowd. They are Jasmine[79] and the compound of Mocha.js[80], Chai.js[81] and Sinon.js[82].

---

[74] List of JavaScript transpiler:
https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS (retrieved 26.05.2014 12:28)

[75] AngularDart is a compelte port of Angular.js to Dart. There are two teams, on Angular.js and one Angular.dart team. These teams collaborate and try to enhance both projects. https://angulardart.org/ (retrieved 26.05.2014 13:03)

[76] AngularGWT: https://github.com/cromwellian/angulargwt (retrieved 26.05.2014 13:00)

[77] One of the many Angular.js TypeScript seed projects:
https://github.com/seanhess/angularjs-typescript (retrieved 26.05.2014 13:04)

[78] More details can be found on their GitHub repository:
https://github.com/cucumber/cucumber-js (retrieved 26.05.2014 12:57)

[79] The project and documentation can be found at the following URL:
http://jasmine.github.io/ (retrieved 28.05.2014 14:50)

[80] More details about Mocha.js: http://visionmedia.github.io/mocha/ (retrieved 28.05.2014 14:55)

Jasmine comes as full bundle and almost everything needed is built in. So it requires not much configuration. Therefore many people believe that Jasmine is inflexible and bloated. This is the reason why another framework came up.

Mocha.js is only a testing framework and needs to be configure. For instance it can be configured to traditional test driven development style or to behavior driven development style. Also there are no assertions built in, so it has to be combined with an assertion library. In most cases therefore Chai.js is used. To use stubs, mocks and spies another library has to be included. This is often done with Sinon.js.

This thesis will focus on Jasmine because it is mature, and supported by Angular.js by default (see chapters 6.2.2, 6.2.3, 6.2.4). Because it comes as full bundle is has less configuration effort. Furthermore there exists a lot of literature about Jasmine. These were the main reasons to choose Jasmine, but they are subjective and Mocha.js could also be a good fit. Jasmine vs Mocha.js is just about personal preferences. Especially if Mocha.js is used in behavior driven development mode it is similar and many of the explained techniques in this chapter will be convertible to Mocha.js without much effort.

5.2.2 Jasmine

As Jasmine is a behavior driven development framework for JavaScript it makes use of the describe/it syntax. Jasmine is not a Gherkin tool so it is better compared with RSpec from the ruby community. Normally Jasmine is used in the inner cycle of the behavior driven development cycle (remember Figure 3). But Jasmine is not restricted to this layer. It could also be used for higher level testing and therefore also be utilized for Gherkin. Tools like Yadda can help doing so. In this chapter the focus will be on the initial idea of Jasmine. This is to provide a testing framework for behavior driven development which can be used from unit to end-to-end testing. Therefore the describe/it syntax is used.

To make developers adopt testing it has to be made as comfortable as possible. As Jasmine is widespread in the JavaScript community there are many tools to automate testing. Test runner like Karma (see 6.2.2) ease configuration and task runners like Grunt.js make it possible to execute tests every time a developer updates a file (which means when she or he saves a file). This might seem to be an aggressive approach but once developers get used to it they will like the instant feedback which comes with that workflow. But this can only work if these tests are fast. Therefore tools like Phantom.js are essential. Let’s assume these facts as given in this chapter.

As Jasmine is a behavior driven development framework everything is built around behavior and the idea of readable test code. The whole vocabulary of behavior driven development is reflected in Jasmine. Tests are called specifications and are organized in “spec”-files[83]. The describe/it functions provide the way to document the specs with a string. Let’s consider the following example:

[81] More details about Chai.js: http://chaijs.com/ (retrieved 28.05.2014 14:56)

[82] More details about Sinon.js: http://sinonjs.org/ (retrieved 28.05.2014 14:57)

[83] In fact they are just plain JavaScript files, but to distinguish between production files and test files they are called spec files.

```
describe("A shopping cart",function() {
    it("should be initially empty",function() {
        var cart = new Cart();
        expect(cart.isEmpty()).toBeTruthy();
    });
    it("should not be empty if a product is added",function() {
        var cart = new Cart();
        cart.add(new Product("JavaScript - The good Parts"));
        expect(cart.isEmpty()).not.toBeTruthy();
    });
});
```

Listing 7: Example of Jasmine describe/it

Because it is obligatory to provide a string for “describe”- and “it”-blocks the specs are readable[84]. Of course this does not imply meaningful descriptions. Someone could also just specify the strings as “test” and “testcase1”. But behavior driven development provides a nice suggestion which often leads to meaningful spec descriptions. At first both strings should be readable as a sentence when concatenated. Second the “describe-string” should specify the object which is tested and the “it-string” has to start with the keyword “should”.

This verbal description of specs should help developers to understand what a spec is doing. But the main goal of these descriptive tests is to create a living documentation [Emr13, p. pos 1306 – 1309 eBook]. The great thing is that this documentation is always in sync with the actual code because it is tied to the implementation. Depending on the test runner the output can be formatted to read as strings.

```
describe('A Product', function(){
  it('should have a price', function() {
    // DO SOMETHING, THIS CODE IS HIDDEN TO KEEP THE EXAMPLE SHORT
  });
  it('should be orderable if on stock', function(){
    // DO SOMETHING, THIS CODE IS HIDDEN TO KEEP THE EXAMPLE SHORT
  });
  describe('which is ordered as present', function(){
    it('should cost  0,5 more than regular',  function(){
      // DO SOMETHING, THIS CODE IS HIDDEN TO KEEP THE EXAMPLE SHORT
    });
    it('should have a nice packaging',function(){
      // DO SOMETHING, THIS CODE IS HIDDEN TO KEEP THE EXAMPLE SHORT
    });
    describe('for christmas', function(){
      it('should be shipped before 17.12', function() {
        // DO SOMETHING, THIS CODE IS HIDDEN TO KEEP THE EXAMPLE SHORT
      });
    });
  });
});
```

Listing 8: example of nested describe blocks in Jasmine

Listing 8 shows a simple spec for an object called product. The exact implementation of the spec is left out on purpose to focus on the narrative parts of the specs. The default test runner of Jasmine would produce the following output for the code in Listing 8:

[84] If one doesn’t let herself or himself be distracted by the JavaScript syntax (which leads to a lot of parentheses)

```
Jasmine 2.0.0                                    finished in 0.002s

• • • • •

5 specs, 0 failures                              raise exceptions

  A Product
    should have a price
    should be orderable if on stock

    which is ordered as present
      should cost € 0,5 more than regular
      should have a nice packaging

      for christmas
        should be shipped before 17.12
```

Figure 6: results of a test run with the default Jasmine test runner

This toy example should demonstrate the narrative power of behavior driven development with Jasmine. The results of the test run can be easily read and understood (even from non technical people). Even the nested "describe"-blocks do not disturb readability. In contrast, they form logical encapsulation of behavior which belongs together. But as rule of thumb there should not be more than three nested describe blocks. Otherwise it gets confusing to read the JavaScript code also for JavaScript experts. If it is necessary to nest more than 3 describe blocks a separate spec file should be considered.

If Jasmine is also used for higher level specification the output ideally is also presentable to the customer and business people.

But let's consider some other interesting aspects of Listing 7. The keyword "expect" acts like "assert" in test driven development. Jasmine also tries to improve readability of these code lines. The "expect" statements can be read also like a sentence. This should help developers especially if a specification does not pass anymore. The "expect" function is implemented as fluent interface[85] which makes it possible to chain assertion. The second "expect" statement in Listing 7Listing 18 shows how the "not" keyword is chained into the expectation.

Jasmine provides a lot of ready to use matchers for "expect", like toBeTruthy, toBeNull, toMatch etc[86]. But often these matchers do not read as nicely as wanted. For this purpose Jasmine provides the possibility to define custom matchers. Also custom matchers make it easy to encapsulate complicated expectation logic into one single function which can be reused throughout the whole project.

For example: we want to write a custom matcher to check if a player is in an expected league. The league in which a player is allowed to participate depends on her or his points. The more points a player has, the higher the league she or he has to play. To illustrate how a custom matcher could be defined consider Listing 9. Beside the effect of more readable specification files another advantage arises. With a custom matcher also custom messages can be defined. This should lead to meaningful error messages if an expectation is not fulfilled.

[85] A fluent interface provides the possibility to chain method calls. For a quick introduction follow: http://en.wikipedia.org/wiki/Fluent_interface (retrieved 14.07.2014 13:30)

[86] All of these matchers are listed on their website http://jasmine.github.io/2.0/introduction.html (retrieved 14.07.2014 13:36)

The examples in Listing 9 and Listing 10 are reduced to the minimum and special error handling is not used. This is due to the fact that these two examples should only illustrate the idea and not give a completely waterproof copy & paste code snippet.

```
beforeEach(function() {
  jasmine.addMatchers({
    toBeInLeague: function() {
      return {
        compare: function(act, expected) {
          var mapLeagueToPoints = function(points) {
              if(points > 91) {
                return 1;
              } else if(points < 91 && points > 80) {
                return 2;
              } else if(points < 81 && points > 70) {
                return 3;
              }
              return 4;
          };
          var result = {
            pass: mapLeagueToPoints(act.getPoints()) == expected;
          };
          if (result.pass) {
            result.message = "Player has " + act.getPoints() +
                             "points so she/he is allowed
                              to play in league "+expected;
          } else {
            result.message = "Player has " + act.getPoints() +
                             "points so she/he is not allowed to
                              play in league "+expected+" expected
                              league would be " +
                              mapLeagueToPoints(act.getPoints());
          }
          return result;
        }
      };
    }
  });
});
```

Listing 9: Defining a custom matcher in Jasmine

How this custom matcher is used is outlined in Listing 10.

```
describe("A player", function() {
  it("should be in league 1 if he has > 90 points", function() {
    var player = new Player();
    player.setPoints(93); // SET POINTS HERE FOR SIMPLYFY EXAMPLE
    expect(player).toBeInLeague(1);
  }

  it("shouldn't be in league 3 if he has > 80 points", function() {
    var player = new Player();
    player.setPoints(93); // SET POINTS HERE FOR SIMPLYFY EXAMPLE
    expect(player).not.toBeInLeague(3);
  }
});
```

Listing 10: Using a custom matcher in Jasmine

Listing 10 shows how a custom matcher improves readability of the “expect” statement.

As a project grows also the size of the test suite gets bigger and bigger. Therefore it is important to think about how to organize the specs and test code. Because Jasmine has a strong emphasis on readability there

should be a balance between DAMP[87] and DRY[88]. Repetition in test code is more acceptable than in production code. This is due to the fact that test code often tests almost the same thing over and over again only with slightly different parameters. Also repetition is mostly appearing in one single spec file. Abstracting away these kinds of repetition would lead to harder to read test code [Emr13, p. pos. 746 eBook].

As mentioned at the beginning of this chapter tests should be executed fast. This means that developers get instant feedback about the state of the code. Task runners like Grunt.js or Gulp.js provide the possibility to execute some action if a file changed. Because Jasmine has a big community there are also plugins for these task runners to execute Jasmine specs on a change. With these tools in place it is possible to create a fast feedback loop. After every change the developer is informed if everything is still working correctly and that the new code does not have any special side effects. The feedback loop eases the use of classical test driven development practices like baby steps or triangulation. If running the test suite would be complicated and tricky these techniques would be impossible to use [Emr13, p. pos. 627 eBook].

But a good testing workflow is not enough. In a real world scenario the tested object will interact with different other objects. If these other objects are "slow" an instant feedback is impossible. These objects could be a database abstraction layer, an object which makes calls via HTTP etc. To prevent slow execution because of external dependencies these object have to be replaced by test doubles[89].

Jasmine provides functionality to create stub and mock objects. Jasmine calls these test doubles spies. Covering the whole spectrum of spies is too broad for this thesis. Following the basic concepts will be described. Details how to mock Angular.js objects will be given in chapter 6.2.4.

Let's assume we are programming a tournament management system for a sports association. Clubs can enroll their players to tournaments. Because these tournaments are added, updated and deleted on regular basis by the administrator the tournaments have to be fetched every time again from the server. Therefore a HTTP call is necessary. This would slow down test execution so this call has to be mocked. Another use case is that the nomination fee has to be added to the club's member dues if this club successfully enrolls a player. To accomplish this, another HTTP request has to be performed. This request should also be mocked. The following Listings show how this could be accomplished with Jasmine (these listings are simplified so that it easier to understand the idea. Therefore no modules, dependency injection framework etc are used. Also the object definitions are made on global scope):

---

[87] DAMP: descriptive and meaningful phrases, [Emr13, p. pos. 764 eBook]

[88] DRY: don't repeat yourself, more information http://en.wikipedia.org/wiki/Don%27t_repeat_yourself (retrieved 14.07.2014 14:47)

[89] This is true for the inner cycle of behavior driven development, which is unit testing in test driven development. End-to-end testing is a whole other story; there you explicitly want to test the whole stack.

```
function Club (id) {
  this.id = id;
}

function Player(firstname, lastname) {
  this.name = firstname + ' ' + lastname;
}

function HttpBackend() {

}

HttpBackend.prototype.getPlayers = function(clubId) {
  // DO FANCY HTTP REQUEST
};

HttpBackend.prototype.fetchPlayers = function() {
  // DO FANCY HTTP REQUEST
};

HttpBackend.prototype.addFee = function(clubId) {
  // DO FANCY HTTP REQUEST
};

HttpBackend.prototype.enroll = function(clubId) {
  // DO FANCY HTTP REQUEST
};

function TournamentManagement(httpBackend,club) {
  this.backend = httpBackend;
  this.club = club;
  this.players = [];
}

TournamentManagement.prototype.fetchPlayers = function() {
  this.players = this.backend.getPlayers(this.club.id);
};

TournamentManagement.prototype.enrollPlayer = function(player) {
  if(this.backend.enroll(player.name)) {
    this.backend.addFee(this.club);
    return true;
  }
  return false;
};
```

Listing 11: Example of a production code for the described tournament management

```
describe('Tournament Management', function(){

  var mgmt, httpBackend, club;

  beforeEach(function() {
    httpBackend = new HttpBackend();
    club = new Club(42);
    mgmt = new TournamentManagement(httpBackend, club);
  });

  it('should fetch all players of a club',function() {
    var players = [
          new Player('Max','Mustermann'),
          new Player('Maria', 'Musterfrau') ];
    spyOn(httpBackend, 'getPlayers').and.returnValue(players);
    mgmt.fetchPlayers();
    expect(mgmt.players).toBe(players);
  });

  it('should enroll player Maria Musterfrau',function() {
    spyOn(httpBackend, 'addFee');
    spyOn(httpBackend, 'enroll').and.returnValue(true);
    expect(mgmt.enrollPlayer(new Player('Mary', 'Maier'))).toBeTruthy();
    expect(httpBackend.addFee).toHaveBeenCalled();
  });

  it('should not add fee is wrong player is enrolled',function() {
    spyOn(httpBackend, 'addFee');
    spyOn(httpBackend, 'enroll').and.returnValue(false);
    expect(mgmt.enrollPlayer(new Player('Evil','Man'))).toBeFalsy();
    expect(httpBackend.addFee).not.toHaveBeenCalled();
  });

});
```

Listing 12: Jasmine test code for a tournament management example app

Listing 11 shows the production code of the Jasmine specs which are stated in Listing 12. To mock a method of an object it is sufficient to call the spyOn method of Jasmine. If this spyOn is applied on a method this method is never executed. If a method has a return value this value can be specified by the "and" and returnValue method. This can be seen in the first "it"-block of Listing 12.

Because everything which deals with money is critical it should be assured that the TournamentManagement object interacts correctly with the backend[90]. This is why in the second "it"-block a spy on addFee is registered. To check if TournamentManagement calls addFee the expectation matcher toHaveBeenCalled can be used. The third "it"-block describes how toHaveBeenCalled can chained with the not method.

This simple example should give an idea of how easy it is to use mocks with Jasmine. But also here it is important to find a balance between mocking and using the real objects. Time consuming tasks are always a good case to mock out. Things which alter the state of the whole system should also be mocked (like operations on the file system, database updates etc) furthermore things like sending e-mails and posting messages on third party services are also good candidates to mock [Emr13, pp. pos. 1712 - 1751 eBook].

Because Jasmine is built for behavior driven development there are also tools to accomplish the outside-in approach. On one hand mocks and stubs can be used via the spy utility and on the other hand, specs can be x'ed-out. This means a spec can be defined as pending (which means it still needs to be implemented). This

---

[90] Due to security reasons it would be better to perform the addFee method on the server in a real world project, but this example should outline the usage of spyOn and toHaveBeenCalled of the Jasmine framework

feature is mostly used for the high level specs and therefore in the outer cycle of behavior driven development. To x-out a spec only a x has to be prefixed before the “it”-method or the “describe”-method. So a xit or xdescribe will not result in a failing spec it will report a pending spec. Marco Emrich thinks that high level and low level specs serve different needs. The first one improves defect awareness and the latter one eases defect localization [Emr13, p. pos. 1626 eBook]. He argues that it make sense to use both especially if a team is using the outside-in approach.

As one can see Jasmine is a sophisticated behavior driven development testing framework for JavaScript. There is a lot more to discover with Jasmine like asynchronous specs, code coverage etc. To learn more the website[91] of the Jasmine project is a great starting point also there exist a lot of literature, blog posts and online tutorials.

[91] http://jasmine.github.io/2.0/introduction.html (retrieved 14.07.2014 20:37)

6 Anuglar.js

As mentioned in chapter 4 there is a paradigm shift from traditional web apps to single page web apps. With this shift there comes a lot of new challenges. This is due to the fact that many browsers are not adapted to these new concepts and that JavaScript was not designed to be used for hundreds of thousands of lines of codes. Especially if a team is new to the topic of single page web apps it makes sense to use one of the existing frameworks. This frees the developers from solving common problems, e.g. browser history, routing (needs to be done on the client side), search engine optimization, authorization and many more. There are several single page app frameworks to choose from. There is not only Angular.js even if there is the biggest hype around Angular.js at the moment. Some of the most well known Angular.js alternatives are:

- Backbone.js (was one of the first JavaScript frameworks for building single page apps)
- Knockout.js
- Ember.js
- React.js (a project from Facebook[92])

It is the same with JavaScript transpiler. There is no right answer to the question: “Which one is the best?” It depends on the preferences of the team and the scope of the project. Angular.js seems to be a very interesting and active project. It also has a huge community and the main principles on which Angular.js is based on fitted to the project which was done during the practical part. The choice of Angular.js was subjective but based on the following facts:

- Build around testing [App13]
- Dependency injection system
- Test runner for unit tests (Karma, more details in section 6.2.2)
- Test runner for end-to-end testing (Protractor, more details in section 6.2.3)
- Declarative UI
- Two-way data binding
- Very active and large community
- Many existing components which are ready to use
- It is easy to use existing libraries
- Great “Get started tutorial”
- Powered by Google

For the practical project an important part was to write high quality code. This can only be achieved by a great testing infrastructure. So this was the main reason why Angular.js was chosen.

Also the fact that there is a lot of attention for the Angular.js project and there is a huge community led to the clear decision that Angular.js is a perfect fit for the practical project.

Angular.js started as a side project of the Google employee Miško Hevery. He claimed that he could rewrite the whole GoogleFeedback project within 2 weeks with Angular.js. At this stage GoogleFeedback was developed over 6 months by 3 people with Google Web Tool Kit. The Java Code had more than 17.000 lines of code. Miško Hevery was not able to rewrite everything within 2 weeks but after 3 weeks he finished the project. The code base shrunk from 17.000 to 1.500 lines of code [Hev14].

[92] More details about React.js: http://facebook.github.io/react/ (retrieved 26.05.2014 13:19)

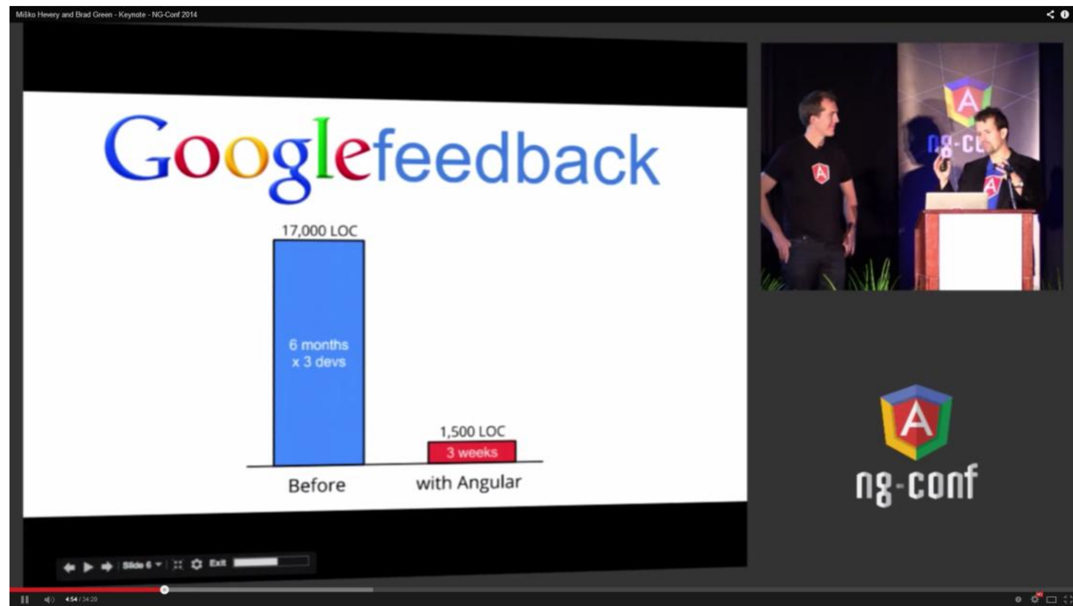


Figure 7: comparsion lines of code for Googlefeedback: GWT to Angular.js
(Source: ng-conf keynote https://www.youtube.com/watch?v=r1A1VR0ibIQ#t=285)

This results impressed many people at Google so they tried to rewrite another project with Angular.js. Therefore they chose doubleclick[93]. The development team was fascinated that they were also able to shrink the code base to $^{1}/_{10}$ [Goo12]. These impressive results draw a lot of attention to Angular.js.

A community began to grow and also non Google projects used Angular.js and took advantage of Angular.js' new architecture. Many other development teams reported similar results like the ones which the doubleclick and GoogleFeedback teams made.

For example Localitics switch from Backbone.js to Angular.js. They were able to shrink the code base by almost 50%.

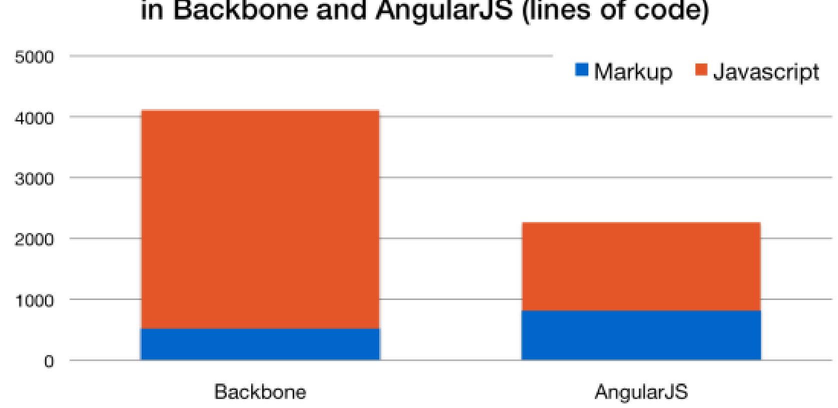


Figure 8: Comparison lines of code at Localitys: Backbone.js to Angular.js
(Source: http://localytics.com/wp-content/uploads/2013/04/amp-angular-backbone-550x339.png)

[93] doubleclick is a online marketing tool which was bought by Google 2007. As incomes from ads are still Google's biggest cash cow doubleclick is an immensely important project for Google. More info about doubleclick are available at their website: http://www.google.com/doubleclick/ (retrieved 10.07.2014 13:10)

Miško Hevery thinks that the reduction of code is due to the two-way data binding and the declarative user interface [Hev14]. A smaller code base has multiple advantages:

- Less code means less space for bugs
- Less code means more time for other things, e.g.: intense testing
- Less code means quicker development cycles (good for TDD or inner cycle of BDD)
- Less code means easier maintainable projects

The achievements made by Angular.js are really impressive and this is why a huge community came up around Angular.js. In January 2014 there was the first Angular conference held in Salt Lake City[94] (Utah - United States of America). The first Angular conference in Europe[95] will take place in Paris (France) at 22nd and 23rd of October 2014.

2013 Angular.js ranked as top 5 GitHub repository with the most contributors and third on repositories with the most star ratings [Git14]. The success of Angular.js led to many ready to use components like Angular UI which is a port of Twitter Bootstrap's Components to the Angular.js world.

At the Google I/O a new design guideline was presented which is called Material Design[96]. There are also ready to use components. For these components there already exists a native Angular.js port[97]. This shows how quickly the community reacts to new trends.

Also for behavior driven development there were written many tools. For example Karma test runner (more details see 6.2.2) and protractor end-to-end test runner (more details see 6.2.3). Karma can make use of jasmine or Mocha/Chai/Sinon and for protractor there even exists a cucumber.js plug-in[98].

Angular.js provides the developers with specialized mock objects to their core components. In combination with Angular.js' dependency injection system these tools and features enable developers to apply behavior driven development in each of its facets.

Many IDEs integrated Angular.js support and there is even a specialized debugging tool called Batarang[99]. The "Dart fork" of Angular led to many new insights. These new knowledge is merged into Angular.js. The Dart fork and the original version are pushing each other to new levels. The near future of Angular seems to be great. For the version 2.0 they set ambiguous goals (they will be discussed in 6.3.1). Another promising announcement was made by Rob Eisenberg. He joins the Angular team to converge his framework Durandal[100] into Angular 2.0 [Eis14]. Angular is an exciting project and it is interesting to follow all the development going on around Angular.

---

[94] ng-conf conference website: http://ng-conf.org/ (retrieved 26.05.2014 16:37)

[95] ng europe conference website: http://ngeurope.org/ (retrieved 26.05.2014 16:37)

[96] More details about "Material Design":
http://www.google.com/design/spec/material-design/introduction.html (retrieved 15.07.2014 12:04)

[97] This port can be found online under: https://material.angularjs.org/ (retrieved 15.07.2014 12:05)

[98] The plug-in can be downloaded via npm:
https://www.npmjs.org/package/protractor-cucumber (retrieved 26.05.2014 15:53)

[99] Angular.js Batarang debugging tool: https://github.com/angular/angularjs-batarang (retrieved 26.05.2014 15:59)

[100] Durandal.js http://durandaljs.com/ (retrieved 10.07.2014 13:16)

6.1 Concepts

This chapter is inspired by chapter 1 of the book “Angular.js” from Brad Green and Shyam Seshardi [Gre13, pp. 1-6] and our own learning during the practical project. Beside the fact that Angular.js is built around testing, there are many other concepts which are followed by Angular.js. At first Angular.js follows the well known MVC[101] pattern. Although MVC is wide spread there is no absolute clear definition what MVC really is. There are many derivates like MVVM[102] or MVP[103]. The team around Angular.js is pragmatic and calls Angular.js a MVW pattern, which means Model View Whatever. The “Whatever” stands for the different options which are available. Because the distinction between Controller, ViewModel, Presenter or anything else is not always clear and discussed a lot the Angular.js team does not want to promote one single concept. Everybody who uses Angular.js can decide how to use the framework [Koz13, p. 12].

6.1.1 Templateing

Angular.js tries to enhance web browsers by teaching them how to understand custom defined html tags and attributes. This makes it possible to use client side templates. This is essential for single page apps because we do not want full page refreshes. Furthermore as much logic as possible should be shifted from server to client. The server only needs be provide accessibility to a given template and the client takes care of the rest. To outline the ideas a short example is given (obviously the most famous one in programming “Hello World!”):

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
        <script src="angular.js"></script>
    <script src="app.js"></script>
  </head>

  <body ng-app="IntroApp">
    <div ng-controller="HomeController">
      <h1>This is a quick intro to Angular.js</h1>
      <p>{{ hello }} World!</p>
    </div>
  </body>

</html>
```

Listing 13: The view of the “Hello World!” example in Angular.js

Listing 13 shows a typical view template. This template is rendered by Angular. The first thing to notice are the new HTML attributes. They are called directives and enable declarative programming on the view. Of course these directives are no valid HTML and if someone wants it to validate the data-* or x-* prefix can be used. The ng-app directive shows Angular which part of the view belongs to which Angular module. With ng-controller a controller can be assigned to a certain part of the view. And the double curly brackets are

[101] MVC: Model View Controller, http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller (retrieved 15.07.2014 12:35)

[102] MVVM: Model View ViewModel (http://en.wikipedia.org/wiki/Model_View_ViewModel, retrieved 15.07.2014 12:39)

[103] MVP: Model View Presenter (http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter, retrieved 15.07.2014 12:38)

used as data binding. Everything in these double curly brackets is evaluated by Angular and rendered into the template.

```
var app = angular.module('IntroApp', []);

app.controller('HomeController', [ '$scope', function($scope){
    // viewModel is just picked as random name, it has nothing to
    // do with angular. It is just good practice to put the model
    // on an own JavaScript object
    $scope.viewModel = {};
    $scope.viewModel.hello = 'Hello';
}]);
```

Listing 14: The business logic of the "Hello World!" example in Angular.js

The according JavaScript is illustrated in Listing 14. At first an Angular module has to be defined. This module is bootstrapped with the ng-app directive. Then it is possible to assign different types of objects to the module. For this example we just need a controller. The first parameter is the name of the controller. The rest is the definition of the controller (here the dependency injection syntax is used, more on that in chapter 6.2.1).

6.1.2 Two way data binding

One could think this is a lot of boilerplate code just to display “Hello World!”, but let us extend the example slightly:

```
<body ng-app="IntroApp">
  <div ng-controller="HomeController">
    <h1>This is a quick intro to Angular.js</h1>
    <p>
      What's your name <input type="text" ng-model="viewModel.name" />
    </p>
    <h2>{{ viewModel.hello }} {{ viewModel.name || "World" }}!</h2>
  </div>
</body>
```

Listing 15: change the view of the "Hello World!" example to show two-way-data binding

In Listing 15 an input element is introduced. The according business logic does not need to be changed and stays the same as in Listing 14. The ng-model directive establishes a two way data binding. This means all changes performed in the view are reflected directly in the model.

This reduces the amount of code needed to react to changes in the view to a minimum. No getter or setter are needed, no registering of event handler has to be performed. All this actions are done by Angular. Angular therefore has a so called digest cycle[104] and does dirty checking on the bindings[105].

[104] Digest cycle: is performed if a certain event is triggered, during the digest cycle the dirty checking mechanism is executed

[105] Dirty checking: on every digest cycle a comparison is performed if the value changed. This leads to performance concerns but in most cases browsers are fast enough. Mostly the limiting factor is not the performance of the dirty checking, it is the ability of a user to react to hundreds or thousands of data bindings. The Angular team did a lot of improvements on this issue. It is also possible to deactivate two way data binding and use it only as one way binding. Also things like Object.observe will increase performance additionally.

6.1.3 $scope

In Angular.js the glue between the View and the model data is the so called $scope object. It is flexible and almost every JavaScript construct can be attached to it. All available properties of this attached construct are then available in the View. Because it is possible to assign everything to the $scope object it is easy to utilize Angular.js for a MVC, MVVM, MVP etc pattern. But a good advice is to keep things attached to the $scope as simple as possible.

Via the $scope object the two way data binding is realized. A thing which can be quite tricky is the $scope hierarchy. This hierarchy is similar to the prototypal inheritance of JavaScript. One important thing to remember is the prototype chain. This means if a property is not found on an object the next object in the prototype chain is inspected and so on. Only if the property is found nowhere an error is thrown [Koz13, pp. 15-18].

To illustrate the concept of scope hierarchies the “Hello World” example of Listing 13, Listing 14 is extended.

```
<body ng-app="IntroApp">
  <div ng-controller="HomeController">
    <h1>This is a quick intro to Angular.js</h1>
    <p ng-controller="InputController">
      What is your name
      <input type="text" ng-model="viewModel.name" />
    </p>
    <h2>{{ viewModel.hello }} {{ viewModel.name || 'World' }}!</h2>
  </div>
</body>
```

Listing 16: The view of a simple example to illustrate $scope hierarchy in Angular.js

Now InputController is under HomeController. In terms of Angular this means that HomeController is in a parent HTML tag of InputController. Everything exposed on $scope belongs to the $scope chain.

```
var app = angular.module('IntroApp', []);

app.controller('HomeController', ['$scope',function($scope){
  // viewModel is just picked as random name, it has nothing to
  // do with angular. It is just good practice to put the model
  // on an own JavaScript object
  $scope.viewModel = {};
  $scope.viewModel.hello = 'Hello';
  $scope.viewModel.setHell = function() {
    $scope.viewModel.hello = 'Welcome to hell ';
  };
  $scope.viewModel.unsetHell = function() {
    $scope.viewModel.hello = 'Welcome to heaven ';
  };
}]);

app.controller('InputController', ['$scope', function($scope){
  $scope.$watch('viewModel.name', function(newVal,oldVal){
    if(newVal===oldVal) {
      return;
    }
    if(newVal==='Georg') {
      return $scope.viewModel.setHell();
    }
    return $scope.viewModel.unsetHell();
  });
}]);
```

Listing 17: The business logic of a simple example to illustrate $scope hierarchy in Angular.js

The InputController should react to the changes on the input, therefore the method $watch on $scope is used. The great thing about $watch is that it is not necessary to think about which event triggered a change. If the value changes and is different to the old one it should be checked which person has entered her or his name. After that the greeting should be altered. The methods setHell() and unsetHell() are not defined on InputController but they are available because InputController is a sibling of HomeController and therefore has access to all properties and methods exposed on the $scope object.

Nesting of $scopes can be powerful but also dangerous (in the sense of introducing bugs and unwanted behavior). This is the reason why the Angular debugging tool Batarang pays special attention to this topic. There is a dedicated tab for this problem. For beginners this tool is great because it helps understanding the scope hierarchies.

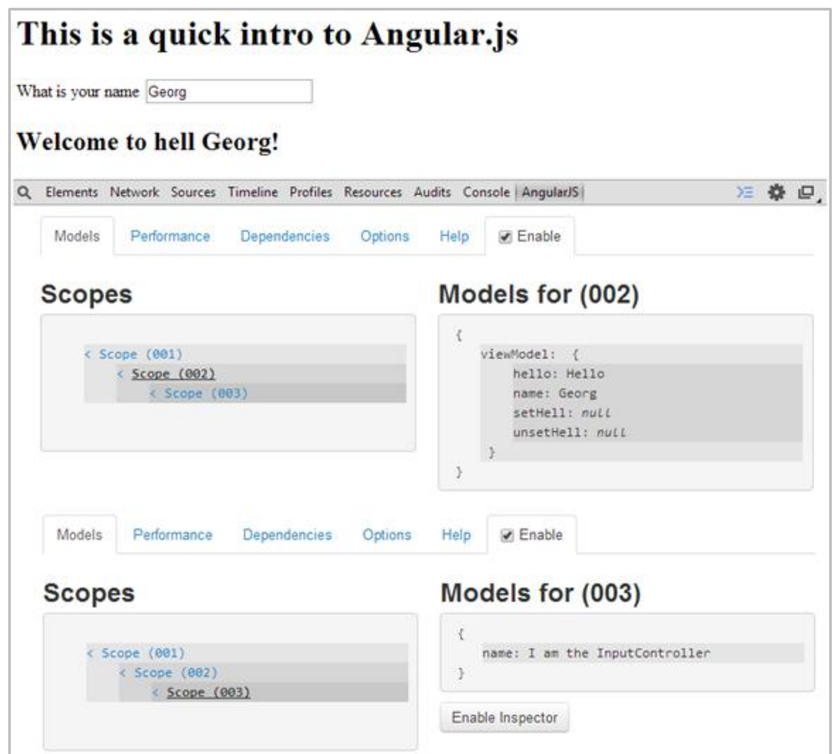


Figure 9: Batarang output of scopes of the simple example to demonstrate scope hierarchies

The black, bold and underlined scope is the scope which is displayed under the Models section. There are three scopes but in the example are only two scopes requested. This is due to the fact that there is always the so called rootScope. This scope is often used to enable communication between controllers via broadcasts or message emits. But this would go too far into detail. The Angular documentation is a good starting point if someone wants to know more about this topic[106].

6.1.4 Dependency Injection

Dependency injection is a design pattern which allows to better decuple objects from their dependencies. Dependency injection is a core concept in Angular.js and has a lot of benefits for building modern web apps. The Angular.js injector takes care of all the dependencies and therefore the order of defining JavaScript

[106] Documentation of $rootScope, $broadcast, $emit and more can be found online at https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$emit (retrieved 15.07.2014 14:45)

objects does not matter. This is enormously handy especially if the code is split across multiple files. If the JavaScript files are included in separate script tags in HTML it does not matter which file arrives first at the client. Also if all files are concatenated to one big file it makes no difference in which order they are concatenated. This takes a lot of organizational work away from developers and eases development of large web apps drastically.

Dependency injection also simplifies testing. This is due to the fact that dependencies can easily be replaced by mock objects or stubs. Angular therefore provides own mock objects.

The dependency injection system became so popular that it will be a separate project in Angular 2.0. Therefore it is possible to use the dependency injection mechanism also in non Angular projects[107].

Detailed information about dependency injection is provided in chapter 6.2.1.

6.1.5 Directives

As already seen in many Listings, like Listing 13, directives are Angular's way to extend the HTML vocabulary. The idea is to create highly reusable components with it. Angular is already shipped with many directives but it is also possible to create own directives. Because the community around Angular is very active there are a lot of ready to use directives, from drag and drop, to file uploads over to date picker or gantt views[108]. The idea of directives is inspired by web components. Directives intersect with another Google project which is called Polymer. Maybe these two projects will converge in this area and enhance each other. But Polymer is too young to estimate in which direction it is heading.

6.1.6 Code organization

Code organization is a big topic in JavaScript. This is due to the fact that JavaScript in the current version has no modules or packages. Furthermore there are no imports and namespaces.

Because of Angular's dependency injection system it is easy to split the source code across multiple files. Before deployment a built step has to be performed which concatenates all files to a single one. This is not a big deal with tools like Grunt or Gulp. Especially when developing in a team separate files are essential.

[107] The new DI system is available on GitHub, but it is written in ES6 so it is not ready for production until browsers implement ES6: https://github.com/angular/di.js (retrieved 16.07.2014 11:20)

[108] Gantt diagram: is a diagram to visualize tasks of a project in a certain form. An Angular adoption can be found on GitHub https://github.com/Schweigi/angular-gantt (retrieved 15.07.2014 14:56)

Angular also provides the possibility to define own modules. On a module you can register the business logic of it. This is smart because then the business logic of a module does not pollute the global namespace[109]. Angular provides different types of objects which can be registered. These are:

- **Controllers**: interact with the view, this is done via the $scope object
- **Services**: this is one of the most controversial types of objects because it splits into three subgroups which are hard to distinguish. It is even worse because services are one of the most important part in an Angular app. Often in services a lot of complicated business logic is accomplished. Another thing to mention is that angular services are always singletons. No matter which type you choose. This is on purpose and if this is not desired behavior there are workarounds.
    - **Factory**: is a JavaScript function which returns an JavaScript object
    - **Service**: is a JavaScript constructor function[110]
    - **Providers**: are like factories but the can be configured through the config phase of a module. To understand this fully a deeper look on Angular is necessary. These differences are often described in online resources or in books like the “ng-book” [Ler13, pp. 157 - 172].
- **Directives**: objects which extend the HTML vocabulary
- **Constants**: values which does not change
- **Values**: values which are allowed to change
- **Animations**: these type of objects are only available if the ngAnimate module is loaded, otherwise they do not do any effect
- **Filters**: these objects can be used to filter data, for example a list which should be filtered according to a query string. Often filters are also used for internationalization like reformatting dates and time.

Recommendation

It is highly recommended to use Angular’s function to register JavaScript into an Angular module. Do not define global objects and use them in the Angular functions, although this is often described in tutorials on the internet. This destroys the concept of modules and brings the danger of naming conflicts.

As Angular’s dependency injection mechanism takes care of object dependencies it is recommended to split the JavaScript code into single files and concatenate them. There should be one object in one file[111].

There are several suggestions about the folder structure and often it is recommended to put all controllers into one folder named controllers, all directives into a directives folder and so on [Gre13, pp. 47 - 50]. The experience we made in the practical part revealed this approach as not scalable. It is great if the application is small but it becomes quickly a mess if the application grows. We agree with the opinion of Cliff Meyers that it makes more sense to group files by topic. Which means to put everything which belongs together into one folder and do not care if the object is a controller, a service or directive [Mey13]. The new Google style guidelines for Angular.js are also the same opinion[112].

---

[109] Beware of online tutorial which promote to put controllers on the global namespace! This destroys the idea of using modules also for namespacing.

[110] Constructor functions are a JavaScript-idiom to mimic object-oriented behavior. Constructor functions are used for prototypal inheritance. Since this is very common in JavaScript the service method provides the possibility to use already existing Classes.

[111] One Angular method call of registering a JavaScript object

[112] Guidelines: https://google-styleguide.googlecode.com/svn/trunk/angularjs-google-style.html#testing (retrieved 17.07.2014 14:07)
Directory structure: https://docs.google.com/document/d/1XXMvReO8-Awi1EZXAXS4PzDzdNvV6pGcuaF4Q9821Es/pub (retrieved 17.07.2014 14:08)

Also we do not see any chance to scale the first approach to huge applications where lazy loading[113] of code is needed. The grouping by type of objects makes it difficult to define modules which can be lazy loaded.

But again (as always) there is no definitive best solution. Every developer has to evaluate each solution and use the one which fits best for her or his situation.

6.2 Testing Angular.js

“Angular is written with testability in mind” – [Ang14]

As stated in chapter 6 Angular is built around testing. That the Angular team takes testing really seriously is also noticeable in their tutorial[114]. Testing is a first class citizen there and introduced from the first steps. For each chapter they show how to setup tests. So testing is prominent and nobody who worked through the official tutorial can overlook or ignore testing in Angular. It is great that the Angular team also puts a focus on testing in their tutorial. The aim is to show that it is easy to integrate testing into your regular workflow. This was not always the case for web developers.

Long time web developers were skeptic about testing, because there was a lack of good automation tools. But nowadays this is not true especially not with test runners and frameworks like Angular.js. The old school way was to write the code and then test the resulting app by hand. This step had to be repeated all the time and in various browsers. Furthermore everything had to be repeated after an update of the software. To escape this mess automated testing was created. Therefore a written specification has to be made.

This specification is executed by a test runner. Ideally the specifications are executed every time a change on the code happens. The developer gets instant feedback if the code she or he is working on is ready[115] and if the new code did not break some functionality elsewhere.

Despite testing is easy to integrate many developers argue that their projects are too small for testing. They are completely wrong. At least when they start refactoring their own code or updating some third party library they will appreciate every single test case they have. Furthermore you never know what will happen to an app. An app designed just for a small use case can become popular and needs to be refactored to fit large scale requirements. Tests are a living documentation which is another plus.

As Angular makes is easy to test your code and there are many great test runners out there, there is no excuse for not testing your code. Sooner or later every project will benefit from testing [Kno13, pp. 35-37].

The Angular team distinguishes basically two types of testing. To test the code of your components they suggest unit tests written in behavior driven development syntax. These unit tests are executed with their test runner Karma[116]. The other kind of test they mention is end-to-end testing. Therefore the test runner Protractor[117] was implemented.

---

[113] Lazy loading in the frontend JavaScript world means to load a script file only when it is needed. This is not officially supported by Angular but there are several modules which help to accomplish it like ocLazyLoad, https://github.com/ocombe/ocLazyLoad (retrieved 16.07.2014 12:52)

[114] The official Angular tutorial can be found on their website: https://docs.angularjs.org/tutorial (retrieved 27.05.2014 10:11)

[115] Ready in terms of red/green/refactor as described in TDD

[116] Karma test runner website: http://karma-runner.github.io/0.12/index.html (27.05.2014 12:30)

[117] More details to Protractor: https://github.com/angular/protractor (27.05.2041 12:31)

End-to-end testing means to test the whole software stack over the graphical user interface. This is more like an acceptance test, because instead of a single piece of code the whole functionality is tested. Unit tests and end-to-end test do not exclude each other they complement each other. A great testing infrastructure has both a good unit test suite and a good end-to-end test suite.

6.2.1 Dependency injection

To make testing of complex applications manageable dependency injection is essential. In mature enterprise frameworks like Java Spring or Java EE dependency injection has already been in use for many years. But the JavaScript world did not use dependency injection for a long time. Maybe this resulted from the fact that "monkey patching"[118] is easy in JavaScript and therefore the interest for creating complex dependency containers was not big enough. As the size of JavaScript applications grow there was a need for better testing infrastructure. Especially if an application relies on different third party libraries "monkey patching" can become a nightmare. Changing behavior of code during run time can always cause unexpected behavior but changing third party code at runtime will always result in an unpredictable mess. Therefore "monkey patching" is not the way to go to build a solid testing suite. Angular.js implements an easy to use dependency injection mechanism. But before the Angular.js mechanism is discussed, the basic ideas behind dependency injection will be described.

Let's consider a JavaScript object called FormHandler. This object can perform several actions but before an action is executed the FormHandler asks a security object if the authentication is correct and the user is allowed to call the action. Listing 18 is a pseudo JavaScript code demonstration how the described behavior could be implemented without dependency injection.

```
function FormHandler() {
  this.securityService = new SecurityService();
  this.doAction = function() {
    // ...
    if(securityService.authentication().hasError()){
      // ...
      throw new Error('There was an error with the authentication
                       of the User');
    } else if(!securityService.user().isAllowed()) {
      // ...
      return false;
    }
    // ...
    return true;
  };
};
```

Listing 18: Shows a high coupling between two components

Obviously it is hard to test all possible execution paths of the FormHandler object. Therefore the authentication state and the user state have to be manipulated. Another way would be to overwrite the SecurityService definition somewhere but this is also not a clean solution. Especially if SecurityService comes from a third party library nobody knows what could happen if this service is overwritten. To loosen the tight coupling of FormHandler and SecurityHandler dependency injection could be used.

[118] Monkey patching describes a technique to change behavior of an object during runtime

There are many different styles of dependency injection but the three most important ones are [Fow04]:

- Dependency injection by constructor
- Dependency injection by setter
- Dependency injection by interface

Because there are no interfaces in JavaScript we will focus on the first two. Listing 19 shows injection via constructor and Listing 20 outlines a setter injection.

```
function FormHandler(securityServiceInjected) {
  this.securityService = securityServiceInjected;
  // THE REST IS THE SAME AS IN THE FIRST EXAMPLES
  };
// assemble objects
var formHandler = new FormHandler(new SecurityService());
```

Listing 19: Example of constructor dependency injection

```
function FormHandler() {
  this.securityService = null;

  this.setSecurityService = function(securityServiceInjected) {
    if(securityServiceInjected === null) {
      throw new Error('Malformed SecurityService! Can not inject');
    }
    this.securityService = securityServiceInjected;
  };
  // THE REST IS THE SAME AS IN THE FIRST EXAMPLES
};
// assemble objects
var formHandler = new FormHandler();
formHandler.setSecurityService(new SecurityService());
```

Listing 20: Example of setter dependency injection

Now it is easy to mock out the SecurityService object while testing. When the objects are assembled a mock object can be used instead of the real SecurityService implementation. The mock object can then be configured to make all execution paths of FormHandler available during testing without altering some global state or the behavior of a real object. Therefore isolated tests are possible and tests can be written in small pieces without a huge setup and tear down phase. But dependency injection can also become complicated. Let’s consider a more complex example:

```
var obj = new FormHandler(new SecurityService(new AuthService(),
              new User('username')),
              new Http(new ResponseHandler(),
                       new ServerExceptionHandler()),
              new ValidationHelper($, new ThirdPartyLibObject()));
```

Listing 21: Example of over complicated constructor dependency injection

As one can see in Listing 21 assembling of objects can become quite complex and complicated in a real world example. To get out of this mess dependency injection containers were invented. Angular.js implements an easy and ready to use dependency injection container[119]. Listing 22 outlines how the code from Listing 21 could be rewritten using Angulars injection system.

[119] The Angular object to perform dependency injection is called $injector and internal details can be found in the Angular Docs (https://docs.angularjs.org/guide/di retrieved 28.05.2014 12:46)

```
var nmcModule = angular.module('nmcModule',
            [/** other modules nmcModule needs */]);
var nmcModule.factory('FormHandler',
  ['$http', 'SecurityService', 'ValidationHelper',
      function($http, SecurityService, RenamedNobodyKnowsWhy){
        // IMPLEMENTATION OF FORMHANDLER
        $http.get('/form/member').success(function(response){
          // ...
        }).error(function(error) {
          // ...
        });
      }
  ]
);
```

Listing 22: Example how dependencies are annotated in Angular.js

There are several ways of defining dependencies in Angular.js[120]. Every method has its pros and cons. During this thesis we will only use the annotation system described in Listing 22 to avoid confusion. Also it is the most robust way Angular provides to list dependencies. The first two lines of Listing 22 are just the instantiation of an Angular module. After that we define a factory which is called FormHandler. The array syntax is used to define the dependencies. The strings tell the injector which object to inject into the FormHandler. This is important to be able to minify and obfuscate this piece of JavaScript. During minification function parameters are renamed to shorter identifiers. The renaming of the parameter makes it impossible for Angular to find the specific dependency. To solve this issue the array syntax with the string annotations was created. The last part in the array has to be a function into which the dependencies are injected. This is a form of constructor dependency injection. The name of the parameter is unessential as hinted with the parameter name "RenamedNobodyKnowsWhy". Only the order is important. The first dependency will be injected into the first parameter and so on. For the sake of simplicity normally the parameters are named the same way as the dependency is called but there could be also other conventions. The Angular injector takes care of the sub dependencies, which for instance ValidationHelper has. (Compare it to Listing 21Listing 21) [Kno13, pp. 29-34]

6.2.2 Karma test runner

During the development of Angular the team developed the Karma test runner (they renamed the project from Testacular to Karma). The main goals of Karma were to provide a test runner which executes tests very fast, runs tests in real browsers and headless browsers. Furthermore the barrier of writing tests should be lowered [Kno13, pp. 47-50]. Nowadays there are Karma plugins for every major browser and tests are executed simultaneously in all registered browsers (which means all browsers which are registered in the Karma config file).

Karma is developed as command line tool so it integrates nicely with existing continuous integration solutions[121]. The command line interface is quite handy but for the big IDEs there are also plug-ins to run Karma. For example Webstorm provides great Karma support.

Karma is test framework agnostic so it is possible to use every JavaScript testing library available. For the most common libraries there are plug-ins and Jasmine, Mocha or QUnit can be used without installing

[120] All possibilities to define dependencies in Angular can be found in their documentation: https://docs.angularjs.org/guide/di#dependency-annotation (retrieved 28.05.2014 13:54)

[121] There are several karma reporters, which can be utilized: https://www.npmjs.org/browse/keyword/karma-reporter (retrieved 17.07.2014 14:16)

something else. It is also possible to integrate code coverage. The default code coverage plug-in uses Istanbul.js[122] to create a coverage report. Karma also provides the possibility to register preprocessors. This can be useful if the code is written in another language which transpiles to JavaScript. Additionally there can be installed many more plug-ins which can be useful for many different use cases. Karma relies on npm[123]. Karma is installed via npm and all the plug-ins and preprocessors are also installed via node packages.

```
# Install Karma:
$ npm install karma --save-dev

# Install plugins that your project needs:
$ npm install karma-jasmine karma-chrome-launcher --save-dev
```

Listing 23: Installing karma and karma plug-ins taken from http://karma-runner.github.io/0.12/intro/installation.html

Special attention should be paid to the "--save-dev" flag. This flag adds an entry to the package.json file. This is useful to organize the project. With the package.json file it is clear which dependencies this project has. This not only helps for the deployment process it also eases the introduction of new developers.

To configure karma with all the plug-ins, preprocessors and parameters there must be written a config file[124]. The syntax is similar to grunt.js and therefore many developers should be familiar with it.

Beside all the other configuration options the parameter autoWatch is interesting. This parameter should be used while developing your app. This causes Karma to re-run all the tests if the developer saves a file. This feature helps enormously, because the developer does not need to switch to the browser and hit the refresh button all the time when she or he wants to check the tests. Also it eases the instant feedback during the development cycle [Koz13, pp. 51-74].

6.2.3 Protractor

Protractor was created as an end-to-end testing tool. It replaced Karma Scenario Runner[125]. Protractor stands on the shoulders of Selenium and WebDriver and adds functionality useful for Angular.js applications. For example it recognizes when a certain view is loaded and rendering is done or when a route changed successfully.

This is useful because the developer does not need to think about when to trigger an assertion. This prevents tests from being flickering[126] just because of timing issues. Elements of the DOM can be access by Angular features like bindings, model, repeater etc.

The default library for writing tests is also Jasmine, so the end-to-end tests feel similar to the unit tests written for Karma. Also installation and configuration feels familiar if you are used to Karma. Protractor can be also installed via npm and configured by a config file. Nevertheless the Angular team thinks it is necessary to split Karma and Protractor into two separate projects.

---

[122] More details to istanbul.js: http://gotwarlost.github.io/istanbul/ (retrieved 27.05.2014 11:38)

[123] Node Package Manager – package manager of node.js

[124] An example can be found a the projects website: http://karma-runner.github.io/0.12/config/configuration-file.html (retrieved 27.05.2014 12:19)

[125] Karma FAQ Question: „Can I use Karma to do end to end testing?" (http://karma-runner.github.io/0.12/intro/faq.html, retrieved 27.05.2014 12:35)

[126] A flickering test means a test which occasionally fails or not

End-to-end testing should be as realistic as possible and therefore use native events, unit testing like Karma does should be fast and deliver instant feedback. They believe that these two principles are orthogonal and therefore they created Protractor. Also they think that WebDriver provides more flexibility than the old Karma Scenario Runner [Ral13]. Even if Protractor is still a young project the Angular team recommends using it instead of the old scenario runner[127]. Also for Protractor there are many plug-ins which can be used. For example there exists a plug-in for Cucumber.js and reporters for Continuous Integration.

6.2.4 Example of Karma and Protractor

This section should give a brief example of how to use Karma and Protractor. This example is not production code it is simplified code to demonstrate quickly some features of both test runners. All the unit tests or specs written for Karma could be seen as the inner cycle of behavior driven development and the specs for Protractor can be used for the outer cycle of behavior driven development. As we use an outside-in approach we will start with Protractor. But at first we need to define the example and define the specifications (the whole code of the example is available in Appendix.

Consider a sport association where clubs can enroll their players to the current tournament. Therefore an “Enrollment page” should be created. After a club authenticates to a server the “Enrollment page” should present the players of the club. The best player should be listed separately. All players can be enrolled to the current tournament and enrollments can be withdrawn again. For a better understanding Figure 10 provides a mockup of a very simplistic user interface to the given example.

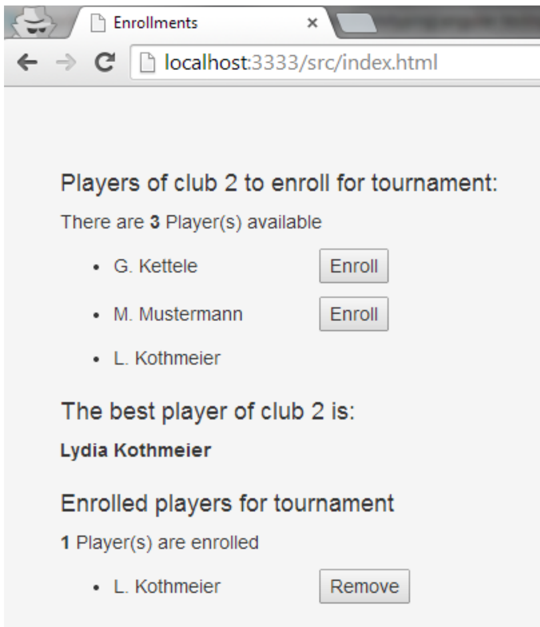


Figure 10: Mockup of a very simplistic user interface for the given example for Karma and Protractor

Because this example is not integrated into a continuous integration and the behavior driven development approach should be emphasized the output of Karma and Protractor is formatted by a behavior driven

[127] See the side mark at the Angular.js documentation page https://docs.angularjs.org/guide/e2e-testing (retrieved 27.05.2014 13:41)

development runner[128]. Furthermore the code coverage tool Istanbul.js is used for a coverage report of Karma test runs.

The first step is to configure Karma and Protractor. This is quite straight forward and there are a lot of tutorials online, therefore this will not be covered here.

Because outside-in approach is used we start with defining the user interface's behavior.

- It should list all players
- It should be possible to enroll all players
- It should be possible to enroll all players and withdraw them again
- The enroll button should be hidden if a player is enrolled

As Jasmine specs this could be rewritten as follows:

```
describe('Enrollment page', function () {
  it('should have the title "Enrollments"', function () {
    // TEST CASE
  });

  it('should list all players', function () {
    // TEST CASE
  });

  it('should enroll all 3 players', function () {
    // TEST CASE
  });

  it('should enroll all 3 players and withdraw them again', function () {
    // TEST CASE
  });

  it('should hide enroll button if a player is enrolled', function () {
    // TEST CASE
  });
});
```

Listing 24: Jasmine specs for the Protractor example

[128] For Karma karma-spec-reporter is used (https://www.npmjs.org/package/karma-spec-reporter, retrieved 17.07.2014 15:30) and for Protractor jasmine-spec-reporter is used (https://github.com/bcaudan/jasmine-spec-reporter, retrieved 17.07.2014 15:28)

Now these specs have to be filled with the actual expectations. To keep the test code simple and readable it is recommended to create a page object[129]. This page object could be implemented as follows:

```
module.exports = function () {
  this.nameInput = element(by.model('yourName'));
  this.greeting = element(by.binding('yourName'));

  this.get = function () {
    browser.get('http://localhost:3333/src/index.html');
  };

  this.getEnrollButtons = function () {
    return [
      element(by.repeater('player in viewModel.players').row(0))
        .element(by.css('button')),
      element(by.repeater('player in viewModel.players').row(1))
        .element(by.css('button')),
      element(by.repeater('player in viewModel.players').row(2))
        .element(by.css('button'))
    ];
  };

  this.getWithdrawButtons = function () {
    return [
      element(by.repeater('player in viewModel.enrollments').row(0))
        .element(by.css('button')),
      element(by.repeater('player in viewModel.enrollments').row(1))
        .element(by.css('button')),
      element(by.repeater('player in viewModel.enrollments').row(2))
        .element(by.css('button'))
    ];
  };

  this.getNumEnrollments = function () {
    return element(by.binding('viewModel.enrollments.length')).getText();
  };

  this.getNumPlayers = function () {
    return element(by.binding('viewModel.players.length')).getText();
  };
};
```

Listing 25: Possible implementation of the enrollment page object

As Protractor makes use of node.js it is necessary to assign the page object to the module.exports property. Then this file can easily be imported with node.js' require function which is shown later. The functions browser, element, by, binding, getText() etc are provided by Protractor and can be used to automate the browser.

As the page object and the stubs of the Jasmine specs are ready the actual test code has to be written. This code could look as follows:

[129] More details about page objects are available at the Protractor GitHub repository: https://github.com/angular/protractor/blob/master/docs/page-objects.md (17.07.2014 15:15)

```
var EnrollmentPage = require(__dirname+'/pages/EnrollmentPage');
describe('Enrollment page', function () {
  var page, enroll, withdraw;

  beforeEach(function () {
    page = new EnrollmentPage();
    page.get();
    enroll = page.getEnrollButtons();
    withdraw = page.getWithdrawButtons();
  });
  it('should have the title "Enrollments"', function () {
    expect(browser.getTitle()).toEqual('Enrollments');
  });

  it('should list all players', function () {
    expect(page.getNumPlayers()).toEqual('3');
  });

  it('should enroll all 3 players', function () {
    enroll[0].click();
    enroll[1].click();
    enroll[2].click();
    expect(page.getNumEnrollments()).toEqual('3');
  });

  it('should enroll all 3 players and withdraw them again', function () {
    enroll[0].click();
    expect(page.getNumEnrollments()).toEqual('1');
    enroll[1].click();
    expect(page.getNumEnrollments()).toEqual('2');
    enroll[2].click();
    expect(page.getNumEnrollments()).toEqual('3');
    // Always zero because always withdraw first one
    withdraw[0].click();
    expect(page.getNumEnrollments()).toEqual('2');
    withdraw[0].click();
    expect(page.getNumEnrollments()).toEqual('1');
    withdraw[0].click();
    expect(page.getNumEnrollments()).toEqual('0');
  });

  it('should hide enroll button if a player is enrolled', function () {
    enroll[0].click();
    expect(enroll[0].isDisplayed()).toBeFalsy();
  });

});
```

Listing 26: Test code for Protractor to automate testing of the given example

To make use of the enrollment page object this file has to be required. This is done in the first line and is node.js specific. The rest is Jasmine and some function calls to Protractor's API. Of course all these tests will fail because no production code is written. At first the user interface will be defined.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Enrollments</title>
      <link type="text/css" rel="stylesheet" href="..">
  </head>
  <body ng-app="IntroApp">
    <div ng-controller="IntroController">
      <h2>Players of club {{ viewModel.clubId }} to enroll for tourn</h2>
      <p>There are <strong>{{ viewModel.players.length }}</strong> Player(s) available</p>
      <ul ng-repeat="player in viewModel.players" id="playerList">
        <li><span>{{ player }}</span>
          <button ng-click="viewModel.enroll(player)"
                  ng-hide="viewModel.isEnrolled(player)">Enroll</button></li>
      </ul>
      <h2>The best player of club {{ viewModel.clubId }} is:</h2>
      <p><strong>{{ viewModel.bestPlayer.firstname }}
                 {{ viewModel.bestPlayer.lastname }}</strong></p>
      <h2>Enrolled players for tournament</h2>
      <p><strong>{{ viewModel.enrollments.length }}</strong>
         Player(s) are enrolled</p>
      <ul ng-repeat="player in viewModel.enrollments">
        <li><span>{{ player }}</span>
          <button ng-click="viewModel.withdraw(player)">Remove</button></li>
      </ul>
    </div>
    <!-- consider using a CDN for Angular in a real world project -->
    <script src="../bower_components/angular/angular.js"></script>
    <!-- concatenate these files in a real world project -->
    <script src="app.js"></script>
    <script src="AuthService.js"></script>
    <script src="PlayersService.js"></script>
    <script src="IntroController.js"></script>
  </body>
</html>
```

Listing 27: A possible user interface for the Karma and Protractor example

Now the outer cycle of the behavior driven development process is accomplished and the inner cycle has to be started. Therefore Karma specs need to be written. To keep the example not too simplistic three objects should be created. One object which is doing authentication to a server, one Angular service which fetches the players and provides business logic on the data and last but not least the controller to interact with the view.

At first the authentication component should be implemented. A possible spec file is shown in Listing 28. Of course it is not possible to write the entire spec beforehand. The real world workflow would be to write one spec check it and write the next one and so on. But Listing 28 presents the result of this workflow.

```
describe('Service Auth', function () {

  var httpBackend, auth;

  beforeEach(function () {
    module('IntroApp')
    inject(function ($httpBackend, Auth) {
      httpBackend = $httpBackend;
      auth = Auth;
    });
  });

  afterEach(function () {
    httpBackend.verifyNoOutstandingExpectation();
    httpBackend.verifyNoOutstandingRequest();
  });

  it('should be available', function () {
    httpBackend.expectGET('/auth').respond(200, {id: 1});
    expect(auth).not.toBe(null);
    httpBackend.flush();
  });

  it('should perform authentication if instantiated', function () {
    httpBackend.expectGET('/auth').respond(200, {id: 1});
    auth.isAuthenticated().then(function (result) {
      expect(result).toBeTruthy();
    });
    httpBackend.flush();
  });

  it('should reject isAuthenticated if server response with 401', function () {
    var err = {msg: "Unauthorized"};
    httpBackend.expectGET('/auth').respond(401, err);
    auth.isAuthenticated().then(function (result) {
      expect(result).toBeFalsy(); // IS NEVER CALLED THROW AN ERROR IF
    }, function (error) {
      expect(error).toBe("Unauthorized");
    });
    httpBackend.flush();
  });
});
```

Listing 28: Jasmine Specs for Karma for the AuthService

The function inject makes use of Angular's dependency injection mechanism. This can be used to inject mock objects. $httpBackend is a mock object which is provided by Angular. It is used to fetch HTTP calls and respond with a certain reply. The code should be quite self explanatory. One thing to mention is that the method isAuthenticated returns a promise[130]. This is due to the fact that this method has to wait for the server response and therefore is asynchronous. The flush() call on the httpBackend object creates an digest cycle and then all promises are resolved. Without this call the promise would never be resolved and no expectation would be checked. After the specs are defined and the auth service is implemented, the outer cycle (Protractor specs) have to be checked. Of course they still fail. The production code of the auth service is listed in Appendix.

So let's hit back to the inner cycle. As next step a service for fetching players should be implemented. The spec file could be as follows: (There are only three sample specs outlined, the rest can be found in Appendix)

[130] A promise is a design pattern in JavaScript which deals with asynchronous code. There are several different implementations of promises. Angular uses the Q library. Further information about $q can be found on the Angular documentation https://docs.angularjs.org/api/ng/service/$q (retrieved 17.07.2014 15:54)

```
describe('Service Players', function () {

  var httpBackend, players, club, data;

  beforeEach(function () {
    module('IntroApp')
    inject(function ($httpBackend, Players) {

     // ARRANGE DATA THIS IS SKIPPED HERE, FULL EXAMPLE IN APPENDIX
      httpBackend = $httpBackend;
      httpBackend.when('GET', '/auth').respond(200, club);
      httpBackend.expectGET('/auth');

      players = Players;

    });
  });

  afterEach(function () {
    httpBackend.verifyNoOutstandingExpectation();
    httpBackend.verifyNoOutstandingRequest();
  });

  it('should reject dataAvailable if server reponse with 403', function () {
    var err = { msg: 'Forbidden' };
    httpBackend.expectGET('/players/' + club.id).respond(403, err);
    players.dataAvailable().then(function () {
      expect(false).toBeTruthy(); // THIS SHOULD NEVER BE CALLED
    }, function (error) {
      expect(error).toBe('Forbidden');
    });
    httpBackend.flush();
  });

  it('should return short name forms, first letter of firstname and full lastname',
     function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, data);
    players.dataAvailable().then(function () {
      expect(players.getPlayersShort()).toEqual([ 'M. Musterfrau', 'S. Maier',
                                                  'P. Kapellari' ]);
    });
    httpBackend.flush();
  });

  it('should return the best player, which is the player with most points', function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, data);
    players.dataAvailable().then(function () {
      expect(players.getBestPlayer()).toEqual(data[1]);
    });
    httpBackend.flush();
  });
// FULL EXAMPLE AVAILABLE IN APPENDIX
});
```

Listing 29: Jasmine Specs for Karma for the PlayersService

After the players-service is implemented (code is available in Appendix) again the Protractor specs are checked. These tests will fail again, so back to the inner behavior driven development cycle again. The last step is to implement the controller which is interacting with the view. The controller specs could be as follows:

```
describe('IntroController', function () {

  var httpBackend, $scope, $rootScope, players, club, data;

  beforeEach(function () {
    module('IntroApp')
    inject(function ($injector) {
      // ARRANGE DATA AND HTTPBACKEND, FULL CODE AVAILABEL IN APPENDIX
      $rootScope = $injector.get('$rootScope');
      $scope = $rootScope.$new();
      players = $injector.get('Players');
      var $controller = $injector.get('$controller');
      createController = function () {
        return $controller('IntroController', { '$scope': $scope });
      };
    });
  });

  afterEach(function () {
    httpBackend.verifyNoOutstandingExpectation();
    httpBackend.verifyNoOutstandingRequest();
  });

  it('should set the viewModel correctly', function () {
    createController();
    players.dataAvailable().then(function () {
      expect($scope.viewModel.bestPlayer).toEqual(data[1]);
      expect($scope.viewModel.players).toEqual([ 'M. Musterfrau', 'S. Maier' ]);
      expect($scope.viewModel.clubId).toEqual(3);
    });
    httpBackend.flush();
  });

  it('should enroll player G. Kothmeier', function () {
    createController();
    var player = 'G. Kothmeier';
    $scope.viewModel.enrollments.push(player);
    expect($scope.viewModel.isEnrolled(player)).toBeTruthy();
    httpBackend.flush();
  });

  it('should enroll player G. Kothmeier and withdraw him again', function () {
    createController();
    var player = 'G. Kothmeier';
    $scope.viewModel.enroll(player);
    expect($scope.viewModel.isEnrolled(player)).toBeTruthy();
    expect($scope.viewModel.withdraw(player)).toBeTruthy();
    expect($scope.viewModel.isEnrolled(player)).toBeFalsy();
    httpBackend.flush();
  });
// FULL EXAMPLE AVAILABLE IN APPENDIX
});
```

Listing 30: Jasmine Specs for Karma for the IntroController

The test code follows the same core idea as with the services. The only difference is in the beforeEach() method. A new $scope is created which is injected into the controller. Because only methods and variables available on the $scope are “public” it is sufficient to test them[131]. The implementation and the whole spec is again available in the Appendix.

Now there is AuthService, PlayersService and IntroController in place so let’s check the outer cycle again. Luckily because we made no mistakes, the Protractor specs also pass.

The details to this example can be all found in the Appendix.

[131] If you also think that testing private properties and methods is not essential

Karma and Protractor are really two efficient tools. There are a lot of tutorials and information available and there are many plug-ins for both tools. Karma is really fast and can be used all the time while developing. So developers get instant feedback immediately after hitting Strg+S.

Protractor in contrast executes the specs in a realistic scenario and is used to test the whole stack. As it is implemented for Angular there are a lot of handy things. Especially the handling of asynchronous code is eased. As seen in Listing 26 there is no need to set timeouts or wait for data. Protractor knows when Angular is ready and promises, route changes, http calls etc are resolved.

Both tools can run several browsers simultaneously and integrate with almost every continuous integration solution. They are easy to setup and fun to develop with. This makes adoption of test driven development also for web frontend projects a bliss. Because the default configuration comes with Jasmine using behavior driven development practices is also no problem.

The bundle of modern development tools like Karma and Protractor, the great features of Angular and the huge community around it make the Angular ecosystem a good choice for developing web apps.

6.3 Summary on Angular.js

That the Angular team is passionate about testing and test driven development is not just a phrase. It is true. From the architecture to the tools and even the tutorial everything is focused about testing. With Karma and Protractor there are two great projects to tackle testing. They make it easy and fun to start testing. The amount of time needed for setting everything up is small and compared to the value a development team gets back minimal.

6.3.1 Outlook

The community around Angular is vivid and almost every day new projects are announced and promoted[132]. Highly interesting are projects like the Ionic Framework[133] which make use of Angular to build HTML5 mobile apps. For Ionic there will be a visual editor to create the user interface for Ionic apps[134]. In conjunction with modules like ngCordova, which provides an API to the components of mobile devices, it is possible to create rich mobile phone apps. With Codrova it is even possible to compile Angular apps to native mobile apps. This is described in detail in the book “ng-book” [Ler13, pp. 458 - 482].

Also the team around Angular is currently working on version 2.0. There will be a lot of great changes but the Angular team suggests not to wait for them. This is due to the fact that they will build version 2.0 around the new ES6 standard. Since this standard is not stabilized and supported by browsers it is not ready for production yet. With the new standard the team wants to refine the Angular syntax and make it more expressive.

Angular 2.0 will be modular which means that it is possible to use only some parts of the framework. The first versions of Angular were “all or nothing”; newer versions are already split into modules like ngRoute and ngAnimate. These modules should also run standalone like the dependency injection module. Tero Parviainen shows how to combine it with backbone.js in his blog [Par14].

---

[132] Therefore check their Google+ page: https://plus.google.com/+AngularJS/ (retrieved 16.07.2014 12:57)

[133] More details about Ionic: http://ionicframework.com/ (retrieved 16.07.2014 12:59)

[134] The visual editor for Ionic apps will be available under: http://ionicframework.com/creator/ (retrieved 16.07.2014 13:32, at this date the editor is not ready to be used by the public)

There will be a lot of performance improvements. This is due to the fact that version 2.0 will only support modern browsers and therefore needs no special treatments and workarounds for browsers like Internet Explorer 8. Also new language features of ES6 like Object.observe will improve performance.

A big redesign will be applied to the directive API. Many users find this API quite tricky and complicated. Also the directive and templating system will be designed to integrate with component frameworks and web components. This is the place where Polymer and Angular could work well together.

There are many more ideas and it is not completely stabilized what features will make it into version 2.0 and how they will look. The process is ongoing and can be followed on the web. There are weekly meetings[135] which can be retrieved from the team's Google Drive; also all design documents[136] are available there.

[135] Weekly meetins to Angular 2.0: https://docs.google.com/document/d/150lerb1LmNLuau_a_EznPV1I1UHMTbEl61t4hZ7ZpS0/edit#heading=h.fd2iriw2247p (retrieved 16.07.2014 13:27)

[136] Design documents to Angular 2.0: https://drive.google.com/?pli=1#folders/0B7Ovm8bUYiUDR29iSkEyMk5pVUk (retrieved 16.07.2014 13:28)

7 Conclusion

During the practical part a web app was developed. This was done together with my colleague Paul Kapellari. The project is still ongoing and started in May 2013. During this time we gained a broad knowledge about web technologies and software methodologies. We used MEAN stack as described in chapter MEAN. Because the JavaScript community is totally into behavior driven development we gave it a try and combined it with extreme programming as outlined in chapter 3.7. The following chapter discusses the “lessons learned” during the last 15 month.

It is not an easy decision to use JavaScript on the whole development stack. This is due to the problems the language still has (see 5.1.2). Hopefully new versions of JavaScript (ECMAScript 6) will solve some of these issues. But it will take a long time until ECMAScript 6 arrives at all browsers. Therefore it is important to choose the right tool to tackle a certain problem. This is not easy because there are many choices, sometimes even too many. For frontend MVC JavaScript frameworks there are five widely spread frameworks and even more small ones. Before a project is started it is often impossible to predict which one is the best fit.

The choice of Angular was a great decision. There are many ready to use components and the community is great. For almost every problem there is a solution available somewhere on the internet. Angular has a steep learning curve but the longer the project was going the more enjoyable it was to work with Angular. Angular reduces the boilerplate code (Figure 7, Figure 8) a lot. Therefore developers can concentrate on the business logic. This makes you feel very productive and makes programming more fun. The great integration of testing tools further eases software development with Angular. Configuration and execution of the test suite are simple (see chapter 6.2.4 and Appendix) and well documented. There are a lot of best practices and tutorials to follow which cover testing in depth. For web development Protractor is a handy tool, because it makes writing end-to-end tests trivial. Karma and Protractor combined can be used very well to perform behavior driven development as intended.

One downside of Angular is that it does not promote a project setup[137]. Therefore we had to restructure the frontend code regularly. Tools like Yeoman try to solve this issue but if a project is slightly different they cannot be use out of the box. The setup which fits best for our project was to group code by its meaning and not by its type (as described in chapter Recommendation). This seems to be a good structure since it is also promoted by the Google Design Guidelines for Angular and many other blogs on the web.

Another big issue with Angular is lazy loading of code. This means to load only parts of the code which are really needed. Angular does not provide this out of the box (maybe it will in version 2.0). Right now the whole code has to be included via HTML script tag at once. This approach is not scalable for huge projects, although it has to be mentioned that this only applies for huge projects. For smaller projects it is overkill to load scripts lazily. This is due to the fact that the round trip time from the server to the client is the limiting factor. Most of the times it makes more sense to load everything at initial page load to the client. When everything is at the client the web app is reactive and a great user experience can be provided. If there is always a delay because the client needs to fetch new scripts the whole idea of web apps is blurred. Our approach to lazy loading was to load all the core code at initial page load and lazy load the customer specific code lazily later if needed. There are two requests for JavaScript code which is a good balance between a scalable project structure and performance. The lazy loading is performed by the module ocLazyLoad[138].

To implement a web server the framework express.js is used. Express.js runs on node.js. Because node.js is asynchronous a completely new mindset is needed to implement software with it. Because we were new to asynchronous programming we needed to do refactoring enormously often. Many errors are not visible at the first sight because they come with race conditions and similar multi threading issues. Node.js is single

[137] this means which folder structure makes most sense and is scalable

[138] Lazy loading module for Angular: https://github.com/ocombe/ocLazyLoad (retrieved 21.07.2014 11:10)

threaded but the asynchronous nature also leads to the same problems as in multi threaded environments. At the beginning of learning node.js it is not always clear what is affected by possible race conditions. Therefore it makes definitely sense to take a deep look into asynchronous programming, node.js and express.js before developing a business critical application with it.

The great thing with node.js is that it provides access to many ready to use packages. This is not only restricted to the server side, almost everything can be implemented with node.js. Like Karma and Protractor are also node.js packages. But because there are so many packages it is again not easy to choose the right one. Also the hype around some package quickly changes to another one and as developer you have concerns if the decision you made some month ago was the right one. Even task runners are changing quickly. When we started our project in May 2013 Grunt.js was the hyped task runner. In July 2014 it seems that the community hypes Gulp.js. One advice is to keep cool and relax. If a tool works for you well then there is no reason to drop it and jump on the next hype. But as a full stack JavaScript developer you should always keep an eye on new things. Sometimes the hype is justified and this particular tool really helps improving something.

Retrospectively the choice of full stack JavaScript was good. The whole JavaScript community (not only node.js or Angular) is very active and helpful. Tools like plunkr[139] or jsfiddel[140] ease knowledge exchange between different programmers around the world. Also the heavy usage of GitHub improves collaboration and idea exchange.

We also tried TypeScript and found it handy. We did not use it for the whole project because another layer of abstraction was too much for us. As a two person team it is feasible to write plain JavaScript. For larger teams TypeScript can be a great fit. We learnt a lot about JavaScript with inspecting the code after TypeScript was compiled to JavaScript. We believe that it is important to first learn JavaScript well and to be familiar with its design patterns (especially inheritance and modules) and then add another tool like TypeScript. Since no matter what tool is used it always compiles down to JavaScript at the end a broad understanding of the language is essential. Also it makes it easier to use some of the existing JavaScript tools, libraries and framework if someone is used to JavaScript.

The other part which should be reflected is the usage of a software methodology. At first extreme programming was used in its pure form. Later during the project we were curious about behavior driven development because it is omnipresent in the JavaScript community. The great thing was that in the middle of the project a real world customer entered it and therefore we were able to apply all the theoretical parts in practical.

Before we utilized behavior driven development it was hard to define specifications together with the customer. The customer was not able to realize the consequences of desired features. Often we heard sentences like this: “Feature X cannot be that hard to implement” or “Why does feature Y takes so much time to implement”. After introducing a ubiquitous language and using the given/when/then templates this process was eased a lot. Defining the specifications in a behavior driven development way together with the customer gave him also the insight which features are complex and why they take long to implement. Despite the fact that there is Cucumber.js to automate Gherkin features we only used them like regular story cards and did not automate them.

Even if there is a ubiquitous language and Gherkin features, misconceptions are not impossible. Therefore a short feedback cycle is essential. Because the customer was not working full time for this project it was not always possible to get feedback instantly. Most of the time this was not a big problem but in some cases this lead to big refactoring which could be avoided if the feedback loop would have been tighter. Another problem which is not easily preventable is the exploration of edge cases later during the project. In our case

[139] Tool to online share web development ideas: http://plnkr.co/ (retrieved 21.07.2014 11:39)

[140] Another online tool to share code ideas: http://jsfiddle.net/ (retrieved 21.07.2014 11:40)

this was for example some problem with user rights. The customer was totally satisfied with the implemented features. Many iterations later he recognized that the user rights have to be changed. This also forced us to refactor large parts of the code. This kind of things are hard to predict and also not easily solvable with quick feedback loops.

Another tricky part for agile software projects are fixed price projects. Because as mentioned before it is hard to predict how much time is needed for a given feature it is even harder to predict if the fixed price is fair or not. A solution would be to just implement as long as there is budget. But many customers do not want to agree on that. They want a fully working solution and not software which only solves certain parts of their problem. So fixed price projects are always risky for the development team and they should be evaluated exactly before started.

The outside-in approach of behavior driven development was great to adopt. After grasping a basic understanding of Protractor it was easier to specify at first the behavior of the full stack and then hit down to the unit testing in form of Jasmine and Karma. Because a new technology was adopted for the project it was often no small thing to write the unit tests beforehand, especially for things which dealt with asynchronous code, promises and callbacks. For this kind of things a broad understanding of JavaScript and the used frameworks are essential. But with the outside-in approach we gained two layers of defense. With end-to-end testing it is possible to verify if the whole stack behaves as expected and with the unit tests it can quickly be checked if all units behave as expected.

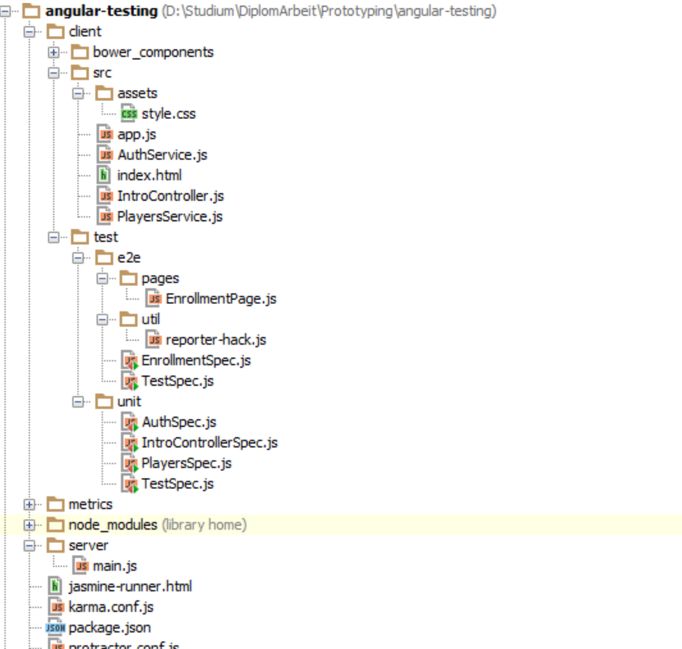
Karma and Protractor helped us also to find browser differences because both tools can run different real browsers at the same time. In our case these were different handling of the JavaScript date object and different cookie handling. Because we decided to support only modern browsers (which basically means no Internet Explorer older than version 9) there were not many browser incompatibilities. The cookie problem was only detected by end-to-end testing with Protractor.

For a small team, doing web development with full stack JavaScript can be a good fit. But learning a new technology stack also has a lot of challenges and to switch from a stack which a team knows well is a move that has to be profoundly evaluated beforehand. Because we got the chance to learn something new as part of this thesis we took this opportunity. But before developing something serious, for the commercial market, every team member has to have a deep understanding of the used software technology stack. It is the same with behavior driven development. It has a great methodology framework but it does not make sense to always use all of it. For example we did not adopt Cucumber.js because it did not bring us more value. In different situations this could be of course different.

A Appendix

The appendix provides more code of the example which is outlined in chapter 6.2.4.

A.1 Project structure



A.2 Index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Enrollments</title>
      <link type="text/css" rel="stylesheet" href="..">
  </head>
  <body ng-app="IntroApp">
    <div ng-controller="IntroController">
      <h2>Players of club {{ viewModel.clubId }} to enroll for tourn</h2>
      <p>There are <strong>{{ viewModel.players.length }}</strong> Player(s) available</p>
      <ul ng-repeat="player in viewModel.players" id="playerList">
        <li><span>{{ player }}</span>
          <button ng-click="viewModel.enroll(player)"
                  ng-hide="viewModel.isEnrolled(player)">Enroll</button></li>
      </ul>
      <h2>The best player of club {{ viewModel.clubId }} is:</h2>
      <p><strong>{{ viewModel.bestPlayer.firstname }}
                 {{ viewModel.bestPlayer.lastname }}</strong></p>
      <h2>Enrolled players for tournament</h2>
      <p><strong>{{ viewModel.enrollments.length }}</strong>
         Player(s) are enrolled</p>
      <ul ng-repeat="player in viewModel.enrollments">
        <li><span>{{ player }}</span>
          <button ng-click="viewModel.withdraw(player)">Remove</button></li>
      </ul>
    </div>
    <!-- consider using a CDN for Angular in a real world project -->
    <script src="../bower_components/angular/angular.js"></script>
    <!-- concatenate these files in a real world project -->
    <script src="app.js"></script>
    <script src="AuthService.js"></script>
    <script src="PlayersService.js"></script>
    <script src="IntroController.js"></script>
  </body>
</html>
```

A.3 App.js

```
var app = angular.module('IntroApp', []);
```

A.4 AuthService.js

```
app.service('Auth', ['$http', '$q', function ($http, $q) {

  var clubId = null;
  var authenticated = null;
  var deferredIsAuth = $q.defer();

  var doAuth = function () {
    $http.get('/auth').success(function (club) {
      clubId = club.id;
      authenticated = true;
      deferredIsAuth.resolve(authenticated);
    }).error(function (error) {
      // DO SOME ERROR HANDLING
      deferredIsAuth.reject(error.msg);
    });
  };

  doAuth();

  this.isAuthenticated = function () {
    return deferredIsAuth.promise;
  };

  this.getClubId = function () {
    return clubId;
  };
}]);
```

A.5 IntroController.js

```
app.controller('IntroController', ['$scope', 'Players', function ($scope, Players) {
  $scope.viewModel = {}; // viewModel is just picked as random name
  // NO EXTRA GETTER AND SETTER TO KEEP CODE SIMPLE
  $scope.viewModel.players = [];
  $scope.viewModel.bestPlayer = {};
  $scope.viewModel.clubId = 0;
  $scope.viewModel.enrollments = [];

  Players.dataAvailable().then(function () {
    $scope.viewModel.players = Players.getPlayersShort();
    $scope.viewModel.bestPlayer = Players.getBestPlayer();
    $scope.viewModel.clubId = Players.getClubId();
  });

  var isEnrolled = function (player) {
    var enrollments = $scope.viewModel.enrollments;
    var length = enrollments.length;
    for (var i = 0; i < length; i++) {
      if (player == enrollments[i]) {
        return true;
      }
    }
    return false;
  };

  var removePlayer = function (player) {
    var enrollments = $scope.viewModel.enrollments;
    var length = enrollments.length;
    for (var i = 0; i < length; i++) {
      if (player == enrollments[i]) {
        enrollments.splice(i, 1);
        return true;
      }
    }
    return false;
  };


  $scope.viewModel.enroll = function (player) {
    if (!$scope.viewModel.isEnrolled(player)) {
      $scope.viewModel.enrollments.push(player);
    }
  };

  $scope.viewModel.isEnrolled = function (player) {
    return isEnrolled(player);
  };

  $scope.viewModel.withdraw = function (player) {
    if ($scope.viewModel.isEnrolled(player)) {
      return removePlayer(player);
    }
    return false;
  }
}]);
```

A.6 PlayersService.js

```
app.service('Players', ['$http', '$q', 'Auth', function ($http, $q, Auth) {

  var players = [];
  var playersShort = [];
  var clubId = 0;
  var bestPlayer = null;
  var deferredData = $q.defer();
  var fetchPlayers = function () {
    $http.get('/players/' + Auth.getClubId()).success(function (playersIn) {
      players = playersIn;
      var length = players.length;
      for (var i = 0; i < length; i++) {
        var player = players[i];
        playersShort.push(player.firstname[0] + '. ' + player.lastname);
      }
      setBestPlayer();
      clubId = (length) ? players[0].clubId : clubId;
      deferredData.resolve();
    }).error(function (error) {
      // DO SOME ERROR HANDLING
      deferredData.reject(error.msg);
    });
  };

  Auth.isAuthenticated().then(function () {
    fetchPlayers();
  });

  var setBestPlayer = function () {
    var maxPoints = 0;
    var playerPos = 0;
    var length = players.length;
    for (var i = 0; i < length; i++) {
      var player = players[i];
      if (maxPoints < player.points) {
        maxPoints = player.points;
        playerPos = i;
      }
    }
    bestPlayer = players[playerPos];
  };

  this.getBestPlayer = function () {
    return bestPlayer;
  };

  this.getPlayersShort = function () {
    return playersShort;
  };

  this.getClubId = function () {
    return clubId;
  };

  this.dataAvailable = function () {
    return deferredData.promise;
  };

}]);
```

A.7 EnrollmentPage.js

```
module.exports = function () {
  this.nameInput = element(by.model('yourName'));
  this.greeting = element(by.binding('yourName'));

  this.get = function () {
    browser.get('http://localhost:3333/src/index.html');
  };

  this.getEnrollButtons = function () {
    return [
      element(by.repeater('player in viewModel.players').row(0)).element(by.css('button')),
      element(by.repeater('player in viewModel.players').row(1)).element(by.css('button')),
      element(by.repeater('player in viewModel.players').row(2)).element(by.css('button'))
    ];
  };

  this.getWithdrawButtons = function () {
    return [
      element(by.repeater('player in 
viewModel.enrollments').row(0)).element(by.css('button')),
      element(by.repeater('player in 
viewModel.enrollments').row(1)).element(by.css('button')),
      element(by.repeater('player in 
viewModel.enrollments').row(2)).element(by.css('button'))
    ];
  };

  this.getNumEnrollments = function () {
    return element(by.binding('viewModel.enrollments.length')).getText();
  };

  this.getNumPlayers = function () {
    return element(by.binding('viewModel.players.length')).getText();
  };
};
```

A.8 EnrollmentSpec.js

```
module.exports = function () {
  this.nameInput = element(by.model('yourName'));
  this.greeting = element(by.binding('yourName'));

  this.get = function () {
    browser.get('http://localhost:3333/src/index.html');
  };

  this.getEnrollButtons = function () {
    return [
      element(by.repeater('player in viewModel.players').row(0)).element(by.css('button')),
      element(by.repeater('player in viewModel.players').row(1)).element(by.css('button')),
      element(by.repeater('player in viewModel.players').row(2)).element(by.css('button'))
    ];
  };

  this.getWithdrawButtons = function () {
    return [
      element(by.repeater('player in
viewModel.enrollments').row(0)).element(by.css('button')),
      element(by.repeater('player in
viewModel.enrollments').row(1)).element(by.css('button')),
      element(by.repeater('player in
viewModel.enrollments').row(2)).element(by.css('button'))
    ];
  };

  this.getNumEnrollments = function () {
    return element(by.binding('viewModel.enrollments.length')).getText();
  };

  this.getNumPlayers = function () {
    return element(by.binding('viewModel.players.length')).getText();
  };
};
```

A.9 AuthSpec.js

```
describe('Service Auth', function () {

  var httpBackend, auth;

  beforeEach(function () {
    module('IntroApp')
    inject(function ($httpBackend, Auth) {
      httpBackend = $httpBackend;
      auth = Auth;
    });
  });

  afterEach(function () {
    httpBackend.verifyNoOutstandingExpectation();
    httpBackend.verifyNoOutstandingRequest();
  });

  it('should be available', function () {
    httpBackend.expectGET('/auth').respond(200, {id: 1});
    expect(auth).not.toBe(null);
    httpBackend.flush();
  });

  it('should perform authentication if instantiated', function () {
    httpBackend.expectGET('/auth').respond(200, {id: 1});
    auth.isAuthenticated().then(function (result) {
      expect(result).toBeTruthy();
    });
    httpBackend.flush();
  });

  it('should reject isAuthenticated if server response with 401', function () {
    var err = {msg: "Unauthorized"};
    httpBackend.expectGET('/auth').respond(401, err);
    auth.isAuthenticated().then(function (result) {
      expect(result).toBeFalsy(); // IS NEVER CALLED THROW AN ERROR IF
    }, function (error) {
      expect(error).toBe("Unauthorized");
    });
    httpBackend.flush();
  });

});
```

A.10 IntroControllerSpec.js

```
describe('IntroController', function () {
  var httpBackend, $scope, $rootScope, players, club, data;
  beforeEach(function () {
    module('IntroApp')
    inject(function ($injector) {
      club = {id: 3}; data = []; data.push({ firstname: 'Mary', lastname: 'Musterfrau', clubId: 3, points: 110 });
      data.push({ firstname: 'Sepp', lastname: 'Maier', clubId: 3, points: 120 });
      httpBackend = $injector.get('$httpBackend');
      httpBackend.when('GET', '/auth').respond(200, club); httpBackend.expectGET('/auth');
      httpBackend.expectGET('/players/' + club.id).respond(200, data);
      $rootScope = $injector.get('$rootScope'); $scope = $rootScope.$new();
      players = $injector.get('Players'); var $controller = $injector.get('$controller');
      createController = function () {
        return $controller('IntroController', { '$scope': $scope });
      };
    });
  });
  afterEach(function () {
    httpBackend.verifyNoOutstandingExpectation();
    httpBackend.verifyNoOutstandingRequest();
  });
  it('should be available', function () {
    expect(createController()).not.toBe(null);
    httpBackend.flush();
  });
  it('should set the viewModel correctly', function () {
    createController();
    players.dataAvailable().then(function () {
      expect($scope.viewModel.bestPlayer).toEqual(data[1]);
      expect($scope.viewModel.players).toEqual([ 'M. Musterfrau', 'S. Maier' ]);
      expect($scope.viewModel.clubId).toEqual(3);
    });
    httpBackend.flush();
  });
  it('should return false when isEnrolled called with Player G. Kothmeier', function () {
    createController();
    expect($scope.viewModel.isEnrolled('G. Kothmeier')).toBeFalsy();
    httpBackend.flush();
  });
  it('should enroll player G. Kothmeier', function () {
    createController();
    var player = 'G. Kothmeier'; $scope.viewModel.enrollments.push(player);
    expect($scope.viewModel.isEnrolled(player)).toBeTruthy(); httpBackend.flush();
  });
  it('should enroll player G. Kothmeier and withdraw him again', function () {
    createController();
    var player = 'G. Kothmeier';
    $scope.viewModel.enroll(player);
    expect($scope.viewModel.isEnrolled(player)).toBeTruthy(); expect($scope.viewModel.withdraw(player)).toBeTruthy();
    expect($scope.viewModel.isEnrolled(player)).toBeFalsy(); httpBackend.flush();
  });
  it('should enroll players P. Kapellari and G. Kothmeier and withdraw Kothmeier again', function () {
    createController();
    var player1 = 'P. Kapellari'; var player2 = 'G. Kothmeier';
    $scope.viewModel.enroll(player1); $scope.viewModel.enroll(player2);
    expect($scope.viewModel.isEnrolled(player1)).toBeTruthy(); expect($scope.viewModel.isEnrolled(player2)).toBeTruthy();
    expect($scope.viewModel.withdraw(player2)).toBeTruthy(); expect($scope.viewModel.isEnrolled(player2)).toBeFalsy();
    expect($scope.viewModel.isEnrolled(player1)).toBeTruthy(); httpBackend.flush();
  });
  it('should not double enroll a player', function () {
    createController();
    var player = 'G. Kothmeier'; $scope.viewModel.enroll(player); $scope.viewModel.enroll(player);
    expect($scope.viewModel.isEnrolled(player)).toBeTruthy(); expect($scope.viewModel.enrollments).toEqual([player]);
    httpBackend.flush();
  });
  it('should not go crazy if a player is removed which is not enrolled', function () {
    createController();
    var player = 'G. Kothmeier'; expect($scope.viewModel.withdraw(player)).toBeFalsy(); httpBackend.flush();
  });
  it('should return false if a player should be remove which is not enrolled', function () {
    createController();
    var player = 'G. Kothmeier';
    $scope.viewModel.isEnrolled = function (player) {
      return true;
    };
    expect($scope.viewModel.withdraw(player)).toBeFalsy();
    httpBackend.flush();
  });
});
```

A.11 PlayersSpec.js

```
describe('Service Players', function () {

  var httpBackend, players, club, data;

  beforeEach(function () {
    module('IntroApp')
    inject(function ($httpBackend, Players) {
      club = {id: 3}; data = [];
      data.push({ firstname: 'Mary', lastname: 'Musterfrau', clubId: 3, points: 110 });
      data.push({ firstname: 'Sepp', lastname: 'Maier', clubId: 3, points: 120 });
      data.push({ firstname: 'Paul', lastname: 'Kapellari', clubId: 3, points: 90 });
      httpBackend = $httpBackend;
      httpBackend.when('GET', '/auth').respond(200, club); httpBackend.expectGET('/auth');
      players = Players;
    });
  });
  afterEach(function () {
    httpBackend.verifyNoOutstandingExpectation();
    httpBackend.verifyNoOutstandingRequest();
  });

  it('should be available', function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, data);
    expect(players).not.toBe(null); httpBackend.flush();
  });

  it('should reject dataAvailable if server reponse with 403', function () {
    var err = { msg: 'Forbidden' };
    httpBackend.expectGET('/players/' + club.id).respond(403, err);
    players.dataAvailable().then(function () {
      expect(false).toBeTruthy(); // THIS SHOULD NEVER BE CALLED
    }, function (error) {
      expect(error).toBe('Forbidden');
    });
    httpBackend.flush();
  });

  it('should return short name forms, first letter of firstname and full lastname', function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, data);
    players.dataAvailable().then(function () {
      expect(players.getPlayersShort()).toEqual([ 'M. Musterfrau',
                                          'S. Maier', 'P. Kapellari' ]);
    }); httpBackend.flush();
  });

  it('should have no trouble with a club without players', function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, []);
    players.dataAvailable().then(function () {
      expect(players.getClubId()).toBe(0);
    });
    httpBackend.flush();
  });

  it('should return correct club id of players', function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, data);
    players.dataAvailable().then(function () {
      expect(players.getClubId()).toBe(club.id);
    });
    httpBackend.flush();
  });

  it('should return the best player, which is the player with most points', function () {
    httpBackend.expectGET('/players/' + club.id).respond(200, data);
    players.dataAvailable().then(function () {
      expect(players.getBestPlayer()).toEqual(data[1]);
    });
    httpBackend.flush();
  });
});
```

A.12 server\main.js

```
var express = require('express');
var app = express();
app.use(express.static(__dirname + '/../client'));

var players = [];
players.push({ firstname: 'Georg', lastname: 'Kothmeier', clubId: 1, points: 10 });
players.push({ firstname: 'Paul', lastname: 'Kapellari', clubId: 1, points: 100 });
players.push({ firstname: 'Kathrin', lastname: 'Hausberger', clubId: 1, points: 150 });
players.push({ firstname: 'Georg', lastname: 'Kettele', clubId: 2, points: 80 });
players.push({ firstname: 'Max', lastname: 'Mustermann', clubId: 2, points: 90 });
players.push({ firstname: 'Lydia', lastname: 'Kothmeier', clubId: 2, points: 140 });
players.push({ firstname: 'Mary', lastname: 'Musterfrau', clubId: 3, points: 110 });
players.push({ firstname: 'Sepp', lastname: 'Maier', clubId: 3, points: 120 });
players.push({ firstname: 'Florian', lastname: 'Krainer', clubId: 3, points: 130 });

app.get('/', function (req, res) {
  res.send('Hello World');
});

app.get('/players/:id', function (req, res) {
  var id = req.params.id;
  var length = players.length;
  var result = [];
  for (var i = 0; i < length; i++) {
    if (players[i].clubId == id) {
      result.push(players[i]);
    }
  }
  res.json(result);
});

app.get('/auth', function (req, res) {
  var id = Math.floor(Math.random() * 3) + 1;
  res.json({id: id});
});

app.listen(3333);
```

A.13 karma.conf.js

```
module.exports = function(config) {
  config.set({
    basePath: '', frameworks: ['jasmine'],
    files: [ 'client/bower_components/angular/angular.js',
      'client/bower_components/angular-mocks/angular-mocks.js',
      'client/test/unit/**/*Spec.js', 'client/src/app.js', 'client/src/**/*.js' ],
    exclude: [ ],
    preprocessors: { },
    reporters: ['coverage', 'spec'],  // dots or spec not both
    preprocessors: { 'client/src/**/*.js': ['coverage'] },
    coverageReporter: { type : 'html', dir : 'metrics/coverage/' },
    port: 9876, colors: true, logLevel: config.LOG_INFO, autoWatch: true,
    browsers: ['Chrome'],// 'Firefox', 'IE'],
    singleRun: false
  });
};
```

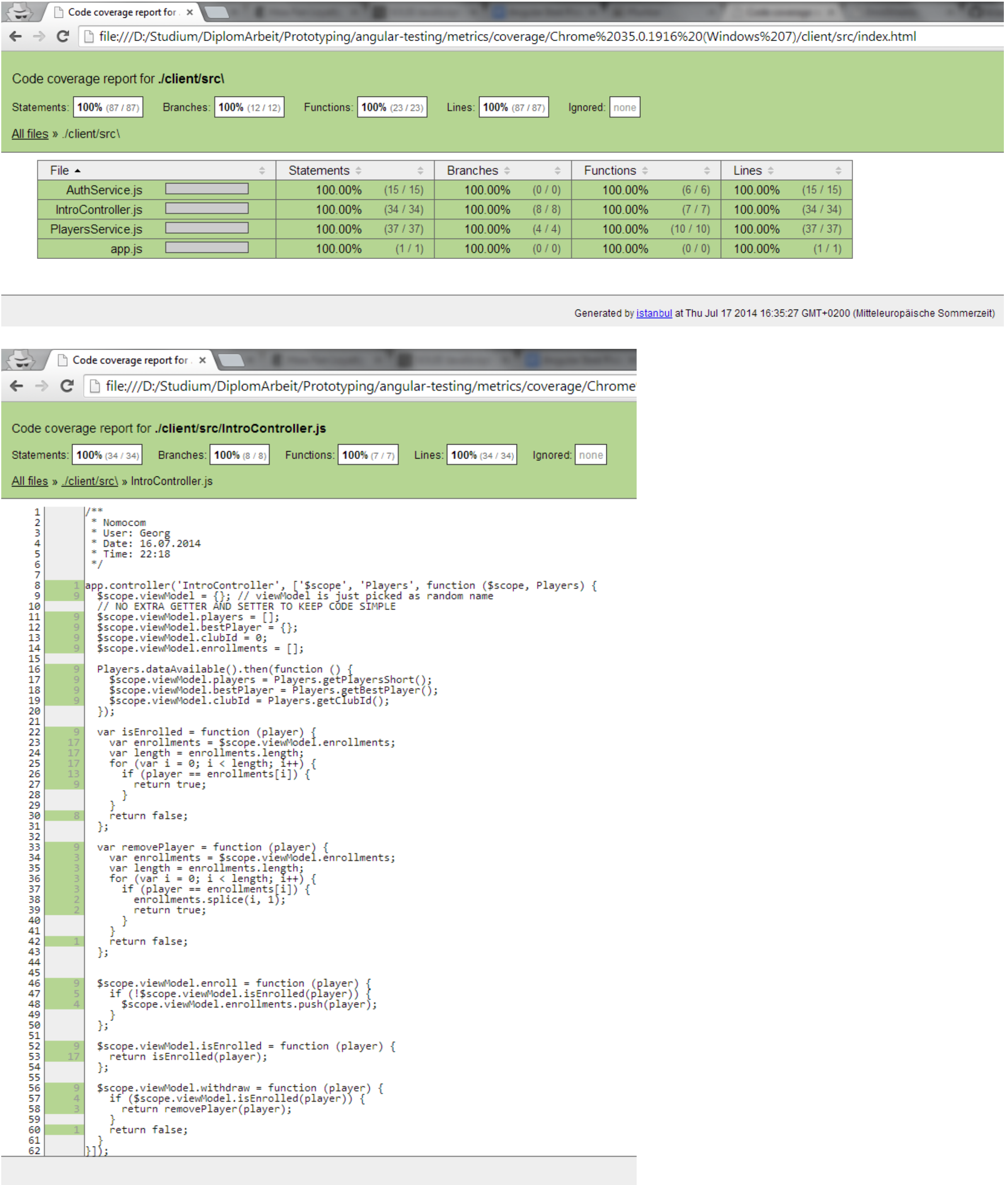
A.14 protractor.conf.js

```
exports.config = { seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: [ 'client/test/e2e/util/reporter-hack.js', 'client/test/e2e/**/*Spec.js' ],
  onPrepare: function () {
    require('jasmine-spec-reporter');
    jasmine.getEnv().addReporter(new jasmine.SpecReporter({displayStacktrace: true}));
  }
};
```

A.15 package.json

```
{ "name": "angular-testing",
  "version": "0.0.0",
  "description": "",
  "main": "server/main.js",
  "author": "",
  "license": "",
  "dependencies": {
    "express": "^4.6.1"
  },
  "devDependencies": {
    "jasmine-spec-reporter": "^0.4.0",
    "karma": "^0.12.17",
    "karma-chrome-launcher": "^0.1.4",
    "karma-coverage": "^0.2.4",
    "karma-firefox-launcher": "^0.1.3",
    "karma-ie-launcher": "^0.1.5",
    "karma-jasmine": "^0.1.5",
    "protractor": "^1.0.0-rc5"
  }
}
```

A.16 Istanbul.js coverage reports



A.17 Karma test runner output

```
Service Auth
  V should be available
  V should perform authentication if instantiated
  V should reject isAuthenticated if server response with 401

IntroController
  V should be available
  V should set the viewModel correctly
  V should return false when isEnrolled called with Player G. Kothmeier
  V should enroll player G. Kothmeier
  V should enroll player G. Kothmeier and withdraw him again
  V should enroll players P. Kapellari and G. Kothmeier and withdraw Kothmeier again
  V should not double enroll a player
  V should not go crazy if a player is removed which is not enrolled
  V should return false if a player should be remove which is not enrolled

Service Players
  V should be available
  V should reject dataAvailable if server reponse with 403
  V should return short name forms, first letter of firstname and full lastname
  V should have no trouble with a club without players
  V should return correct club id of players
  V should return the best player, which is the player with most points

Test
  V should pass if karma is configured correctly

rome 35.0.1916 (Windows 7): Executed 19 of 19 SUCCESS (0.141 secs / 0.132 secs)
```

A.18 Protractor test runner output

```
D:\Studium\DiplomArbeit\Prototyping\angular-testing>node_modules\.bin\protractor protractor.conf.js
Using the selenium server at http://localhost:4444/wd/hub
Spec started

  Enrollment page
    V should have the title "Enrollments"
    V should list all players
    V should enroll all 3 players
    V should enroll all 3 players and withdraw them again
    V should hide enroll button if a player is enrolled

  Test
    V should pass if protractor is configured correctly

Executed 6 of 6 specs SUCCESS in 10 secs.

D:\Studium\DiplomArbeit\Prototyping\angular-testing>
```

A.19 Commands to run Protractor and Karma

If Karma and Protractor are not installed globally the commands to start both test runners are slightly different than regularly found on the internet. At first it is necessary to open a terminal and change to the root path of the application. Then the following commands should be enough:

```
#RUN WEBDRIVER
node node_modules\.bin\webdriver-manager start

#RUN PROTRACTOR
node node_modules\.bin\protractor protractor.conf.js

#RUN KARMA
node node_modules\karma\bin\karma start
```

B Bibliography

[Nel13] S. Nelson-Smith, Test-Driven Infrastructure with Chef, 2nd edition ed., O'Reilly Media, 2013.

[Dor00] P. Dorsey, "Top 10 Reasons Why Systems Projects Fail," Dulcian, Inc, 2000.

[Cha05] R. N. Charette, "http://spectrum.ieee.org/," 2 September 2005. [Online]. Available: http://spectrum.ieee.org/computing/software/why-software-fails. [Accessed 29 May 2014].

[Ken99] B. Kent, Extreme Programming Explained, 1st edition ed., Addison Wesley, 1999.

[And03] D. J. Anderson, Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results, Prentice Hall Computer, 2003.

[Pet09] K. Petersen, C. Wohlin and D. Baca, "The Waterfall Model in Large-Scale Development," in *Product-Focused Software Process Improvement, 10th*, 2009.

[Stan11] The Standish Group, "Chaos Manifesto," Boston, 2011.

[Sho07] J. Shore and Chromatic, The Art of Agile Development, O'Reilly Media, 2007.

[Che10] D. Chelimsky, D. Astels, D. Zach, A. Hellesey, B. Helmkamp and D. North, The RSpec Book, O'Reilly, 2010.

[Bec00] K. Beck, Extreme Programming - The Manifesto, Addison-Wesley, 2000.

[Ken01] B. Kent, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, "Manifesto for Agile Software Development," 2001. [Online]. Available: http://agilemanifesto.org/iso/en/manifesto.html. [Accessed 2 June 2014].

[Sol11] C. Solís and X. Wang, "A Study of the Characteristics of Behaviour Driven Development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Limerick, Ireland, 2011.

[Nor06] D. North, "http://dannorth.net/," 1 March 2006. [Online]. Available: http://dannorth.net/introducing-bdd/. [Accessed 2014 June 3].

[Emr13] M. Emrich, Behaviour Driven Development with JavaScript - An introduction to BDD with Jasmine, Developer.Press, 2013.

[Ast06] D. Astels, "A new look on test driven development," Astels Consulting, 2006. [Online]. Available: http://www.elgustodecrecer.es/media/pdf/uploaded/menu/12375.pdf. [Accessed 3 June 2014].

[Rad11] B. Rady and R. Coffin, Continuous Testing: with Ruby, Rails, and JavaScript, Pragmatic Bookshelf, 2011.

[Car10] R. A. d. Carvalho, F. L. d. Carvalho e Silva and R. S. Manhaes, "Mapping Business Process Modeling constructs to Behavior Driven Development Ubiquitous Language," Nucleo de Pesquisa em Sistemas de Informação, Campos/RJ, Brazil, 2010.

[YeW13] W. Ye, Instant Cucumber BDD How-to, Birmingham, UK: Packt Publishing, 2013.

[Nor14] D. North, "What's in a Story? | Dan North & Associates," [Online]. Available: http://dannorth.net/whats-in-a-story/. [Accessed 6 June 2014].

[Goj11] A. Gojko, Specification by Example, Shelter Island, NY: Manning, 2011.

[Way12] M. Wayne and A. Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Dallas, Texas: Pragmatic Bookshelf, 2012.

[Amo13] E. Amodeo, "BDD with JS: Architecture, Tools & Patterns," 6 June 2013. [Online]. Available: http://eamodeorubio.github.io/bdd-with-js/#/15. [Accessed 2014 June 12].

[Nor09] D. North, "How to sell BDD to the business," 27 November 2009. [Online]. Available: https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business. [Accessed 12 June 2014].

[Mik13] M. Mikowski, J. Powell and G. Benson, Single Page Web Applications: JavaScript End-To-End, Manning Pubn, 2013.

[Cou01] G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems: Concepts and Design (3th Edition), Addison-Wesley, 2001.

[Bis13] T. Bissyandé, F. Thung, D. Lo, L. Jiang and L. Réveillère, "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects," Laboratoire Bordelais de Recherche en Informatique, Singapore Management University, France/Singapore, 2013.

[OGr14] S. O'Grady, "The RedMonk Programming Language Rankings: January 2014 – tecosystems," 20 January 2014. [Online]. Available: http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/. [Accessed 20 May 2014].

[Rod12] G. Roden, Node.js & Co, Heidelberg: dpunkt.verlag, 2012.

[Cro08] D. Crockford, JavaScript: The Good Parts, O'Reilly Media, 2008.

[Han13] S. Hanselman, "JavaScript is Web Assembly Language and that's OK.," 23 May 2013. [Online]. Available: http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx. [Accessed 20 May 2014].

[Bru13] G. Brunner, "Unreal Engine 3 ported to JavaScript and WebGL, works in any modern browser," 28 March 2013. [Online]. Available: http://www.extremetech.com/gaming/151900-unreal-engine-3-ported-to-javascript-and-webgl-works-in-any-modern-browser. [Accessed 23 May 2014].

[Res13] J. Resig, "John Resig - Asm.js: The JavaScript Compile Target," 3 April 2013. [Online]. Available: http://ejohn.org/blog/asmjs-javascript-compile-target/. [Accessed 23 May 2014].

[Vaa12] Vaadin, "The Future of GWT Report," Vaadin, Turku, Finland, 2012.

[App13] Appliness, "Miško Hevery about Angular.js - Interview," *Appliness,* pp. 77-89, 1 February 2013.

[Hev14] M. Hevery and B. Green, "Miško Hevery and Brad Green - Keynote - NG-Conf 2014," 16 January 2014. [Online]. Available: https://www.youtube.com/watch?v=r1A1VR0ibIQ#t=189. [Accessed 26 May 2014].

[Goo12] Google Doubleclick Team, "Rebuilding DoubleClick with AngularJS (AngularJS MTV Meetup 2012-08-14)," 7 September 2012. [Online]. Available: https://www.youtube.com/watch?v=ee3Ecw8rl1Y#t=676. [Accessed 26 May 2014].

[Git14] GitHub, "GitHub Octoverse 2013," 5 February 2014. [Online]. Available: https://octoverse.github.com/. [Accessed 26 May 2014].

[Eis14] R. Eisenberg, "Angular and Durandal Converge | AngularJS," 14 April 2014. [Online]. Available: http://blog.angularjs.org/2014/04/angular-and-durandal-converge.html. [Accessed 26 May 2014].

[Gre13] B. Green and S. Seshardi, Angular.js, Sebastopol, CA 95472: O’Reilly Media, Inc, 2013.

[Koz13] P. Kozlowsk and P. B. Darwin, Mastering Web Application Development with AngularJS, Birmingham : Packt Publishing Birmingham - Mumbai, 2013.

[Ler13] A. Lerner, ng-book - The complete book on AngularJS, Fullstack.io, 2013.

[Mey13] C. Meyers, "Code Organization in Large AngularJS and JavaScript Applications — Cliff Meyers," 21 April 2013. [Online]. Available: http://cliffmeyers.com/blog/2013/4/21/code-organization-angularjs-javascript. [Accessed 16 July 2014].

[Ang14] Angular Docs, "AngularJS: Developer Guide: Unit Testing," 23 May 2014. [Online]. Available: https://docs.angularjs.org/guide/unit-testing. [Accessed 27 May 2014].

[Kno13] A. Knol, Dependency Injection with AngularJS, Birmingham: Packt Publishing Birmingham - Mumbai, 2013.

[Fow04] M. Fowler, "Inversion of Control Containers and the Dependency Injection," 23 January 2004. [Online]. Available: http://www.martinfowler.com/articles/injection.html. [Accessed 28 May 2014].

[Ral13] J. Ralph, "Docs: relationship with Karma · Issue #9 · angular/protractor · GitHub," 24 June 2013. [Online]. Available: https://github.com/angular/protractor/issues/9#issuecomment-19931154. [Accessed 27 May 2014].

[Par14] T. Parviainen, "Angular 2.0 Dependency Injection - Applied To Backbone TodoMVC," 18 March 2014. [Online]. Available: http://teropa.info/blog/2014/03/18/using-angular-2-0-dependency-injection-in-a-backbone-app.html. [Accessed 16 July 2014].

[Kni13] C. Knight, "Five stages of grief: Evolving a multi-page web app to a single page application," 11 April 2013. [Online]. Available: http://de.slideshare.net/CharlesKnight1/five-stages-of-grief-evolving-a-multipage-web-app-to-a-single-page-application-charles-knight-kevin-burnett-final. [Accessed 21 May 2014].