

Christian Hofer

Behaviour Driven Web Development

Master's Thesis



Graz University of Technology

Institute for Software Technology

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, September 2014

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.

Graz, _____
Date Signature

Eidesstattliche Erklärung¹

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig 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 _____
Datum Unterschrift

¹Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

This thesis deals with the development of web applications employing agile software development methodologies. In particular the challenges for employing functional tests are investigated.

The growing popularity of mobile devices entails new requirements for web applications. Especially, the reduction of screen real estate and the support of different view modes are challenging. Therefore it is important to design flexible layouts, which adapt to the available screen space. Additionally, the limited transmission speed of the mobile web is a considerable constraint. In order to improve the response time, the contents should be separated from the representation data and transferred asynchronously in the background. These requirements complicate testing.

Subsequently agile software methodologies are discussed. Among them are development methodologies, such as Extreme Programming (XP), and Test Driven Development (TDD), as well as project management methodologies, such as Scrum, and Kanban. Moreover, test tools are examined, which are designed for web applications and are suitable for agile software development, in particular whether they fulfil the aforementioned requirements.

Finally, Behaviour Driven Development (BDD) is introduced and compared to the previously discussed agile software development methodologies. The advantages are pointed out including improved communication between developers and customers, always up-to-date documentation, and the supporting development process.

In the last chapter an implementation is illustrated, which employs the widely used BDD tool *Cucumber* to execute functional web tests. The used functional web testing framework is Selenium and the features are written in Gherkin, a domain specific language introduced by Cucumber. Afterwards the advantages and disadvantages of applying BDD and TDD are discussed.

Kurzfassung

Diese Masterarbeit beschäftigt sich mit der Entwicklung von Webanwendungen unter der Verwendung von agilen Softwareentwicklungsmethoden. Insbesondere werden die Herausforderungen für Funktionstests untersucht.

Durch die steigende Popularität von mobilen Geräten gibt es neue Anforderungen für Webanwendungen. Besonders die Reduktion der Bildschirmgröße und die Unterstützung von verschiedenen Ansichtsmodi sind problematisch. Deshalb ist es wichtig flexible Layouts zu entwerfen, die sich an den verfügbaren Bildschirmplatz anpassen. Zusätzlich ist die eingeschränkte Übertragungsgeschwindigkeit des mobilen Internets eine beträchtliche Einschränkung. Um die Antwortzeiten zu verbessern, sollte der Inhalt von den Darstellungsdaten getrennt und asynchron im Hintergrund übertragen werden. Diese Anforderungen erschweren das Testen.

Anschließend werden agile Softwaremethoden besprochen. Unter ihnen sind Entwicklungsmethodologien, wie Extreme Programming (XP) und Test Driven Development (TDD), sowie Projektmanagementmethodologien, wie Scrum und Kanban. Außerdem werden Testwerkzeuge untersucht, die für Webanwendungen bestimmt sind und für agile Softwareentwicklung eingesetzt werden können, insbesondere ob sie die zuvor erwähnten Anforderungen erfüllen.

Schließlich wird Behaviour Driven Development (BDD) eingeführt und mit den vorher besprochenen agilen Softwareentwicklungsmethodologien verglichen. Auf Vorteile wird hingewiesen, darunter verbesserte Kommunikation zwischen Entwicklern und Kunden, immer aktuelle Dokumentation und der unterstützende Entwicklungsprozess.

Im letzten Kapitel wird eine Implementierung veranschaulicht, die das weitverbreitete BDD Tool *Cucumber* einsetzt, um Funktionstests durchzuführen. Das verwendete Funktionstest Framework ist Selenium und die Features werden in Gherkin geschrieben, eine domänenspezifische Sprache, die mit Cucumber eingeführt wurde. Danach werden Vorteile und Nachteile bei der Verwendung von BDD und TDD besprochen.

Contents

1	Introduction	1
1.1	Agile Software Development	1
1.2	Catrobat Project	1
1.3	Community Website	3
1.4	Motivation	3
2	Web Applications	5
2.1	Divide and Conquer	6
2.2	Native Applications for Websites	7
2.3	Mobile First with Responsive Webdesign	9
3	Agile Software Development	11
3.1	Extreme Programming (XP)	12
3.2	Scrum	13
3.3	Kanban	14
3.4	Test Driven Development (TDD)	15
3.5	Continuous Integration (CI)	16
4	Test Tools	17
4.1	Unit Tests	18
4.1.1	Architecture	19
4.1.2	The Basics	22
4.1.3	Advanced Techniques	24
4.2	Functional and Integration Tests	26
4.2.1	Architecture	27
4.2.2	The Basics	31
4.2.3	Advanced Techniques	33
5	Behaviour Driven Development	36
5.1	The Basics	37
5.1.1	Express Requirements	37

Contents

5.1.2	Gherkin Syntax	38
5.1.3	Workflow Cycle	39
5.2	Benefits	40
5.2.1	Ubiquitous Language	40
5.2.2	Automated Acceptance Tests	41
5.2.3	Living Documentation	42
5.2.4	Software Quality	43
5.3	Related Tools	44
6	Guided Procedure	47
6.1	Create Features	47
6.2	Create Step Definitions	50
6.2.1	Transformations	50
6.2.2	Implementation	50
6.2.3	Global State	52
6.3	Create Production Code	54
7	Behaviour Driven Functional Testing of Web Applications	58
7.1	Introduce Behaviour Driven Development	58
7.1.1	Preconditions	59
7.1.2	Cucumber-JVM Meets Selenium Grid	59
7.2	Findings	68
7.2.1	Comparison	68
7.2.2	Disadvantages	71
7.3	Related Tools	71
7.3.1	Mink	71
7.3.2	Capybara	72
8	Conclusion	73
	Bibliography	76

List of Figures

1.1	Pocket Code programs are able to control robots and quadcopters	2
1.2	The community website shown in different form factors	3
2.1	Trend of global mobile internet traffic	6
3.1	The first two columns of the community website's Kanban board	15
4.1	The structure of a Selenium Grid setup	31
5.1	BDD workflow illustrated in a cycle	40
5.2	Living documentation of Tic-Tac-Toe on Relish	43
7.1	Selenium Grid console with connected nodes	60
7.2	Selenium screenshot taken after the last step of the search functionality scenario	67

List of Listings

1	Example output of PHPUnit test runner	19
2	PHPUnit usage information	20
3	A doc comment with group and data provider annotations . . .	21
4	Selenium wait for an expected condition evaluated by JavaScript	34
5	Selenium open page with defined viewport size	35
6	Example of a story template	38
7	Example of the add products to a shopping cart feature	48
8	Cucumber output with missing step definitions	49
9	Cucumber output with pending step definitions	51
10	Example of transformations used for step definitions	52
11	Example of implemented step definitions	53
12	Example of a world extension	53
13	Cucumber output with failing step definitions	55
14	Example of the shopping cart implementation	56
15	Support code to make the implementation accessible	56
16	Cucumber successfully executes all scenarios	57
17	The output of Cucumber when executed in an empty folder . . .	57
18	Selenium test that checks the website's search functionality . . .	62
19	Selenium tests that verify a certain text on the start page and address the website's language switch functionality	63
20	Selenium test that investigates the download count functionality	64
21	The aforementioned Selenium tests converted to Cucumber sce- narios	65

List of Tables

5.1	Test tools capable of performing StoryBDD	45
5.2	Implementations of Cucumber's wire protocol	46
5.3	Test tools capable of performing SpecBDD	46
7.1	Advantages and disadvantages of BDD performing functional web tests	70
7.2	Advantages and disadvantages of TDD performing functional web tests	70

1 Introduction

1.1 Agile Software Development

The process of software creation is composed of the determination of customer requirements and the development of a design that fits those requirements. In traditional software development this challenge is solved by creating a comprehensive design upfront with as complete specifications as possible. This approach was inherited from engineering fields building concrete objects, where late changes are always very expensive [Che+10].

However, it turned out that this practice is not best suited for software architecture. Furthermore, it contributes to factors which cause the delivery of software that is late, is over budget, has wrong functionality, is unstable in production, or is costly to maintain. Additionally, the fear of expensive change causes big design efforts, which are the main reason for expensive changes late in the development process. Therefore this practice can be described as self-fulfilling prophecy, since it provokes undesirable effects [Che+10].

When it was increasingly recognised as a major problem, new methods were explored to escape the described cycle. The obtained solutions were summarised under the terms of agile software development and agile project management. All involved methods focus on an effective creation of software, which prefers an iterative development in small steps and provides value to the customer at every iteration [Bee+01b].

1.2 Catrobat Project

The Catrobat project employs many aspects of first generation agile software development and agile project management. The aim of the project is to support children to gain programming experience in an enjoyable way. In order to

1 Introduction

keep the target audience of children motivated, an attractive presentation of the programming environment is important. Therefore a visual programming language is employed that is called *Catrobat* and is strongly inspired by the Scratch programming language¹ [Sla12]. In contrast to Scratch, *Catrobat* focuses on mobile platforms operated by Android, iOS, or Windows Phone.

The creation of games, animations, simulations, and other programs is achieved by combining blocks, which have different properties and functionalities, representing commands. For example, there are particular blocks capable of controlling a Lego Mindstorms robot², as shown in Figure 1.1, or a Parrot's AR.Drone quadcopter³ [Sla12].

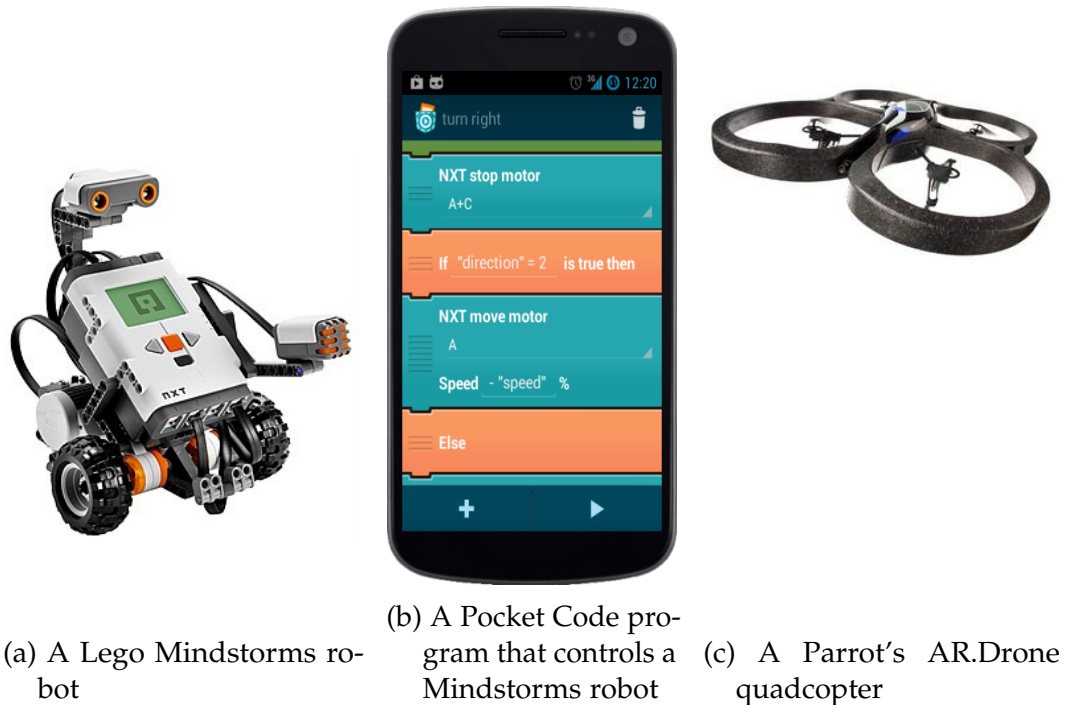


Figure 1.1: Pocket Code programs are able to control robots and quadcopters

¹<http://scratch.mit.edu/about/>

²<http://www.lego.com/en-us/mindstorms/>

³<http://www.parrot.com/>

1.3 Community Website

The community website⁴ is an integral component of the Catrobat project for sharing and distributing user created programs. Therefore, it is important that the website offers a pleasant user experience on the supported platforms including technical aspects like response times. The user created programs are licensed under a free software license, so that the community benefits from every program that is added. Additionally, this allows on the one hand to learn from existing programs and on the other hand to modify and re-share existing programs without limitations, which is called “remixing” [Sla12].

Moreover, it offers instructions to assist with first steps, and attempts to increase children’s motivation. The community website accomplishes two tasks to boost motivation for producing impressive and unique programs: it is a place to share and present the results of hard work, and it provides a collection of programs to compete with and to get inspiration from.



Figure 1.2: The community website shown in different form factors

1.4 Motivation

With regards to a growing software project, it is often necessary to increase the resources by adding additional employees in order to remain competitive.

⁴<https://pocketcode.org/>

1 Introduction

Although a higher number of employees does not automatically guarantee the generation of more value, as it increases the number of communication channels, and therefore slows down the overall development speed [Davog]. Accordingly, effective communication between team members is an important factor that affects productivity in large projects.

The Catrobat project is divided into several sub-teams, which concentrate on individual tasks. For instance, every supported mobile platform is managed by its own sub-team that makes use of the platform's native tools for development. Consequently, a variety of tools and programming languages are applied to create tests, which are fundamental to agile software development. It is apparent that project members, who are not involved in a particular sub-team, have to invest additional effort to identify the system's functionality.

Therefore, the project aims to create consistent tests written in plain text following Gherkin syntax (see Section 5.1.2), which are readable and comprehensible without special knowledge. The desired gain is on the one side a better overview of the overall system functionality and on the other side an improvement in the communication between sub-teams.

This thesis discusses properties and benefits of various agile software development and management methodologies. In addition it covers the introduction of a second-generation agile software development methodology to the community website, and discusses the encountered difficulties in regard to web development. The employed practice is called *Behaviour Driven Development* and its main objective is to encourage more effective communication between customers and developers, which is also applicable to the communication among sub-teams.

2 Web Applications

“The simple guideline is whatever you are doing—do mobile first,”
recommends Google Chairman Eric Schmidt [Sch11].

When intending to create valuable software that offers a positive user experience it is important to be aware of the target audience and their constraints. In the latest Internet Trends report [MW13] a rise in mobile web usage is noticeable, as shown in Figure 2.1. It indicates a general growing popularity in mobile devices, and according to a study from 2012 [Süd12] approximately 90% of 12 to 13 year olds have a personal mobile device. Therefore 15% of global Internet traffic was attributed to mobile devices in May 2013. In a few countries, this is even more apparent, such as in India in 2012 and in China in 2013, the number of users accessing the web using mobile devices surpassed the number of users using desktop PCs.

Additionally, the growth of mobile web usage still has potential for improvement and is expected to continue over the next few years. This becomes apparent from the distribution of mobile phones, which indicates that there are 1.5 billion smart phone users and 5 billion mobile phone users [MW12] [MW13]. In Austria this trend is also visible, since at least 78% of the population own a smart phone, and the vast majority of them use it to access the web [Mac].

Furthermore, the shift in usage implies that new constraints and requirements for web applications arise, which affects the way in which they are built. The most influential and noticeable change concerns the large reduction in screen real estate. Therefore the employment of common development strategies, like the creation of a fixed width layout, are no longer applicable [Wro11].

In addition, this new class of devices introduces further unique hardware characteristics, which need to be addressed individually. The displays of modern mobile devices are equipped with very high resolutions and pixel densities. Therefore, it is important to take varying pixel densities into account, as images appear to be blurry on screens with high pixel densities. A way to address this problem is to deliver images in higher resolutions, generally twice the size as

2 Web Applications

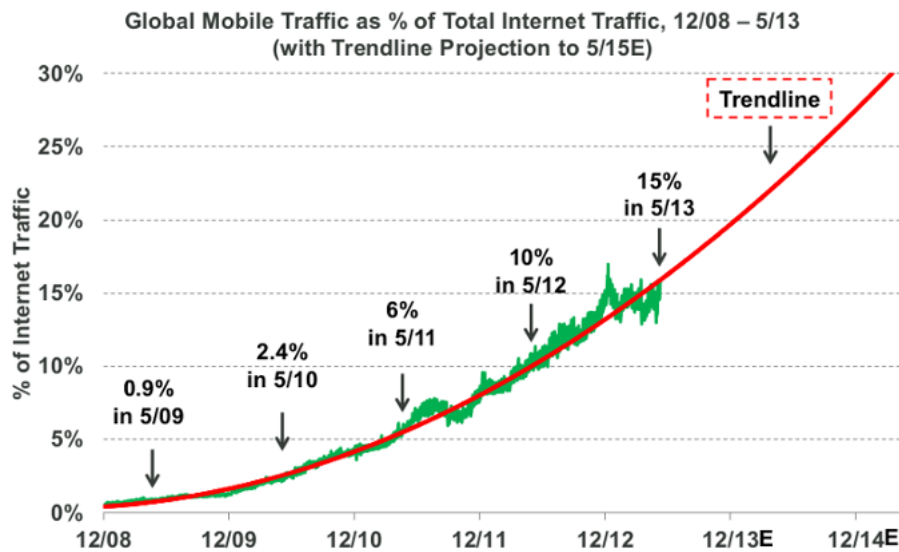


Figure 2.1: Trend of global mobile web traffic, taken from [MW13]

they are displayed.

Another unique property of mobile devices is the ability to alter the screen orientation between two possible view modes. The view modes are called landscape mode and portrait mode and have considerably varying screen proportions. To challenge this situation, an option is to create multiple versions adjusted to predefined screen dimensions, or to make use of fluid layouts also referred to as responsive web design.

2.1 Divide and Conquer

This is the most naïve approach, it suggests to create a separate web application, which is optimised for mobile usage. With the first access of the visitor the application determines the used device and loads the best fitting version. When applying this approach, it is good practice to prepare a subdomain for optimised versions, additionally when accessing the primary domain with a mobile device, the visitor should be automatically redirected [Mar10].

The main advantage of this approach is that no additional knowledge is necessary, since known techniques can be applied, and that a working solution is

2 Web Applications

obtained in a relatively short time, as available code can be reused and adapted.

The downsides with this solution are on the one hand higher maintenance costs resulting from an additional code base, and on the other hand, it is not future-proof. For considerable future changes in regard to the available screen size, the effort of building an optimised version has to be repeated. In addition, every new version adds additional time for executing the test suite, which slows down the development and increases the costs.

When examining current web usage statistics the weakness of this method becomes apparent immediately. Based on the numbers of the Internet Trends report [MW12], tablet shipments have surpassed desktop PCs and notebooks in the last quarter of 2012 suggesting that tablets become increasingly popular. This indicates that the range of screen sizes, which should be supported by web applications, has expanded, ranging from very small (smart phones) to very large (desktop PCs) and varying in-between sizes (tablets). It becomes difficult to decide how many optimised versions should be created and which sizes should be supported.

2.2 Native Applications for Websites

Another approach to improve the user experience for mobile users is to create a native application and let the platform framework take care of the platform's constraints. Compared with web applications, they have the following advantages. They can achieve better performance by omitting an additional abstraction layer, which is attributed to the browser engine. This is especially important, if performance is a concern, for example as it is the case for intense mathematical calculations or complex 3-D animations. Additionally, the user interface transitions and interactions have a smoother appearance.

They also can reduce the network latency by lowering the required amount of remote application data. This reduction of remote application data is achieved by making use of the native user interface. In addition network related performance issues can be overcome by caching previously received data. As a result, it leads to time and resource savings yielding an overall improved execution time, which is beneficial to the user experience [AGL10] [Wro11].

From the developer's point of view comprehensive development tools and well-described interface guidelines make a big difference. Conveniently, most mobile platforms offer great development tools and human interface guidelines

2 Web Applications

to reduce development efforts and to ensure a consistent user experience. The development tools usually consist of a software development kit (SDK), that grants access to low-level application programming interfaces (API), and an integrated development environment (IDE), which is tailored to the needs of the corresponding platform. With these tools, it becomes a lot easier to create small and medium sized mobile applications within a short time. As a downside these tools are limited to a single platform, and therefore developers have to be familiar with development tools of multiple platforms [Was10] [CL11].

By pursuing the native application approach, it becomes very expensive to reach users on multiple platforms, as it is necessary to build a separate application in each platform's native language. To reduce the development and maintenance costs, cross-platform development tools can be engaged to build applications that behave nearly the same as native applications.

Among the most applied frameworks are: RhoMobile's Rhodes¹, MoSync², and PhoneGap³. These frameworks take advantage of the fact, that all major mobile platforms offer a web browser and the capability to access the browser engine programmatically. Therefore, web technologies such as HTML5 and JavaScript are employed to render the application in a web view. However, relying solely on the web technology stack is not sufficient for accessing native device features, like the camera, compass, location information, local storage, and others that can be a requirement for certain applications. To overcome this restriction, these frameworks provide interfaces, which offer access to those features.

In conclusion, cross-platform development can be very powerful and is capable of utilising many advantages of native applications. However, there is still an additional effort required for creating a user interface with the native look and feel [Was10] [CL11].

Although native applications offer noticeable advantages, there are still certain use cases for which web applications are better suited. For instance the seamless delivery of updates and the ability to propagate them immediately to the user by the means of a page refresh. Therefore, this behaviour can be utilised to conduct A/B tests with little effort, which are used to evaluate different designs [Wro11]. Another advantage of conventional web applications is a reduction

¹<http://www.motorolasolutions.com/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite/Rhodes>

²<http://www.mosync.com>

³<http://phonegap.com>

in maintenance costs, in particular when the project depends on a website anyway.

2.3 Mobile First with Responsive Webdesign

This approach changes the development strategy by shifting the focus onto mobile devices, and reduces the fragmentation of previous approaches by employing modern web technologies. Focusing on mobile devices uncovers unique possibilities, since these devices offer new technical capabilities, such as location detection and touch input. That leads to an enhanced context awareness and a more intuitive way of interacting with the application, which can improve the user experience [Wro11] [Fra12].

For example when searching for local information, like the nearest restaurant, the nearest cinema or the nearest subway station, knowledge of the current location greatly increases the quality of search results. Additionally, the touch input enables direct access to control elements and supports a set of standard gestures mimicking real world interactions. Among them are pinch to zoom, swipe left to go back and others providing fast access to basic functionality [Wro11] [DD11].

However, the limited space enforces a greater concentration on the core values of the application, which leads to a much cleaner and better-structured layout. Additionally, a reduced version is well suited for progressively enhancing the design and content, as adding new elements is much easier than removing existing elements [Wro11].

Another important aspect of mobile devices is the data transmission speed of the mobile web. In order to reduce the bandwidth consumption, a technique called Asynchronous JavaScript and XML (Ajax) can be employed. The main idea is to separate the presentation and the content of the web application, so that succeeding requests update only elements that contain new information. Therefore the page load time can be improved, when a partial page update is sufficient. Additionally, it allows to continuously update elements without a page refresh by the means of asynchronous polling requests, which is useful for implementing a news ticker [UD07].

Nevertheless, previously unforeseen difficulties can arise due to the modified request/response cycle. The browser history can appear to be broken, when relying heavily on asynchronous communication, as asynchronous requests are

2 Web Applications

not handled by the browser and have to be addressed separately. In regard to search engine optimisation there are concerns that useful information is hidden from web crawlers, which affects the ranking of the website. Finally a developer-facing concern relates to code complexity that is caused by the increased use of callback functions and the corresponding error handling [UD07].

In conclusion this technique is highly beneficial for the user by providing a more responsive application and by saving bandwidth. However, it is more challenging for developers, as it leads to code that is more unpredictable and harder to test.

In order to best utilise the available screen space, a technique called responsive web design is employed. It applies the capabilities of the new HTML5⁴ and CSS3⁵ technologies to create adaptive designs, including fluid grids, flexible images, and media queries. As a consequence, layouts and images can be adjusted to the screen size, and control elements, such as links or buttons can be enlarged to increase the target area on small touch screens [Wro11] [Fra12].

The fundamental advantage of this approach is that the website adapts to all possible screen sizes of current and future devices, whether it is a smart phone, desktop PC, or tablet. However, this flexibility does not come without a cost, as it results in a higher test and development effort [Fra12].

⁴<http://www.w3.org/TR/html5/>

⁵<http://www.w3.org/TR/css-text-3/>

3 Agile Software Development

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software,” states the first principle of the Agile Manifesto [Bee+01b].

Agile software development is not a software development methodology on its own, but it is a philosophy defining principles, which should be included in an agile software development process. It emerged from the desire for an alternative to the waterfall model that is considered rigid and inflexible.

The inflexibility of the waterfall model results from the linear and sequential development flow, which requires substantial design efforts in advance. Therefore, changes to previous phases are very expensive and should be avoided. However, some requirements don't appear until the implementation phase. That leads to an enormous overhead for subsequent alterations and makes it a heavyweight development method. To address this, agile software development is applied using more lightweight methods [Mik13].

The principles of agile software development state that the customer's needs should be satisfied by creating valuable software, software that works and is as simple as possible, and delivering it consistently. Additionally the development process should be sustainable and changes in requirements should always be welcome. Another important aspect is that business people and developers should work together and communicate in a direct manner. The development team should be self-organised and built around motivated individuals, furthermore regular reviews to reflect on progress made and on how to improve the development process are strongly recommended [Bee+01b].

A recent survey [Amb11] shows that agile software development increases the success rate of software projects and the satisfaction of product owners. The product owner is the person who makes decisions and has the final say about the product. Agile software development is among the most successful development paradigms and is, in addition, very effective. In comparison to

other paradigms it generates the best value for product owners and it provides the best return on investment (ROI).

3.1 Extreme Programming (XP)

Extreme Programming is a widely used agile software development methodology that combines best practices of software development, such as unit testing, pair programming, and refactoring. The development process is divided into multiple cycles and is therefore more flexible than non-agile methods. This means, that it is not necessary to know all requirements up front and allows the frequent release of new versions. Additionally, changing requirements are not considered as an increased risk and are always welcome [Bec99].

In the beginning of every development cycle, requirements for the upcoming release are defined in a planning session. They are discussed and planned by the developers in collaboration with an on-site customer. They are then written onto story cards with an estimate of the required implementation time and the priority. If everyone is satisfied with the created tasks, the development phase begins [Bec99] [Hus+08].

This type of development adheres very strongly to the main statement of the Agile Manifesto, which states that software development should be done by doing it. Additionally, it follows a test first programming methodology. Therefore, it is an important aspect to express requirements as unit tests and to automate the test procedure. On the one hand the unit tests serve as acceptance tests, which indicate if a requirement is satisfied, and on the other hand they act as documentation. That is particularly important, because otherwise no specification documents are provided [Bec99] [Bee+01a].

A technique called pair programming is used in order to spread knowledge in the team. Two programmers work in a pair and share one workstation. They have different responsibilities, one writes code, referred to as the driver, and the other reviews the code as it is typed, referred to as the observer. They should regularly change positions. This leads to better code quality, because errors are promptly fixed and solutions for difficult problems can be developed as a team with their combined knowledge [Bec99] [Hus+08].

The primary objective of Extreme Programming is to write only as much code as is necessary to pass the unit tests. Therefore, refactoring is essential for

improving the overall design. It can be applied very efficiently, because due to collective code ownership every team member is responsible for the code quality and is encouraged to make changes to any code file within the project. Additionally changes can be done with confidence, as regressions are revealed immediately by the unit tests [Bec99].

3.2 Scrum

Scrum is a project management methodology suitable for agile software development and can be combined with software development methodologies, such as Extreme Programming and Test Driven Development. The development process becomes more visible and more transparent to the product owner and offers more room for interactions [SS13].

A Scrum team consists of a product owner, who decides which features should be included in the product, a development team, which develops the product, and a Scrum master, who is responsible for detecting and eliminating problems in the development process. All items that are required to produce the product features are collected in a product backlog by the product owner. The items are represented as user stories, which should be clearly expressed, so that everyone on the team understands the purpose and is able to prioritise its importance. These items are then processed in an iterative, incremental approach in order to collect and integrate feedback in an early development stage. An iteration is called sprint and takes between one week and one month, at the end of each iteration a working product can be demonstrated to the product owner [SS13].

In the beginning of every sprint, a planning meeting is held, in which the goals of the forthcoming iteration are discussed and defined. Typically, items are moved from the product backlog to the sprint backlog and are estimated by the development team. This is useful for revealing valuable information, because in order to estimate the expected time expenditure, additional information from the product owner is often required. To get this information the developers have to ask precise questions and have to listen carefully [Bjö09a].

During a sprint, no new requirements can be added, so that the developers can concentrate on the current tasks. Additionally there are daily Scrum meetings in which everyone briefly reports on yesterday's progress, which problems were encountered and what is planned for today. This helps the team to stay

3 Agile Software Development

up to date on the overall progress and enables team members to share valuable advice [Suto4] [SS13].

When the sprint is done, a review is held in order to report back on achievements. This is also a great opportunity to give feedback. After that, a sprint retrospective is held to evaluate the team performance in regard to people, relationships, process, and tools. In addition, potential improvements and things that went well are discussed in order to improve team performance for the next sprint. At this point either another sprint can be planned to further improve the product, or the product owner decides that the product is completed [SS13].

3.3 Kanban

Kanban is the Japanese word for index card and refers to a second-generation agile project management methodology originating from Toyota Production System¹ (TPS). At TPS the cards are used to regulate the flow of the assembly line production [Sco10].

The number of cards is fixed for each part in the upstream process limiting the amount of pre-produced parts. Every time a part is consumed by the downstream process, the index card is returned to the upstream process and the production of a new part is permitted. This method effectively prevents the overproduction of parts, which cannot be consumed by the downstream process. If the upstream process is not able to produce sufficient parts, the number of cards for the concerned part can be increased to improve productivity [Hiro8].

In software development, this method is used to limit the work in progress (WIP) by limiting the number of active tasks. This reduces the pressure on the team and leads to increased code quality, reduced waste, and higher productivity. One of the biggest differences to Scrum is that a continuous workflow is sustained in contrast to the iterative approach. Furthermore, it is useful for making all active work visible [Hiro8] [Sha11].

A common practice is to write a story card for each task and put them on a Kanban board. Usually story cards contain information about the task id, the task name, a time estimate, and who is working on the task. The Kanban board is divided into multiple columns representing the task state. If a column is filled with story cards, it is not permitted to start a task from the previous column.

¹http://www.strategosinc.com/toyota_production.htm

3 Agile Software Development

This restriction helps to detect bottlenecks in the workflow. The number and meaning of the columns can be chosen freely and can be different for each project [Hiro8] [Sha11].

When a product owner knows exactly which functionality should be included in the product, the Kanban method can potentially be faster than the Scrum method for finishing the product. This is due to the continuous workflow, and time savings made by not having sprint planning meetings. Kanban meetings are only held if they add value to the customer or they are required for making crucial decisions [Bjö09b].

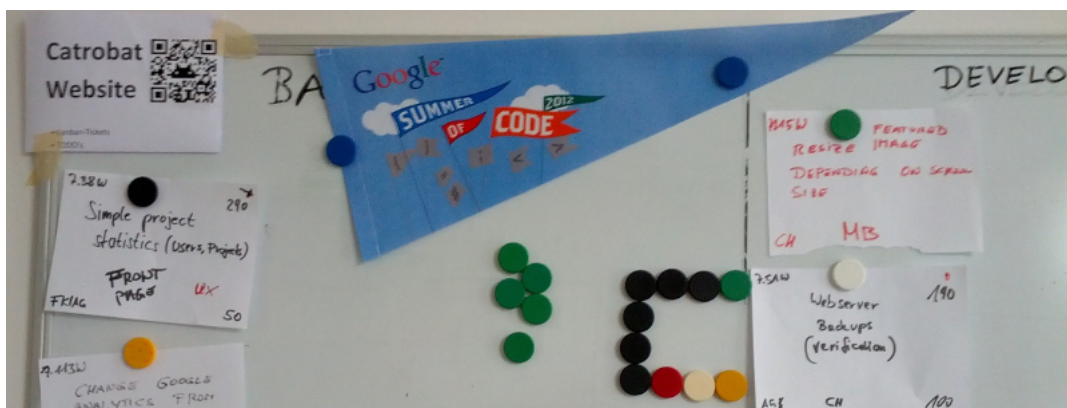


Figure 3.1: The first two columns of the community website's Kanban board

3.4 Test Driven Development (TDD)

The employment of Test Driven Development (TDD) generates a test suite with a large amount of automated test cases. It follows a test-first development style that imposes that a unit test is written before new code can be added. It is highly recommended that a test-first technique is applied to all kinds of agile software development, as the test suite strongly reduces regressions arising from frequent changes or refactoring [Beco3]. For that reason TDD heavily depends on unit tests. However, it is important that the unit tests provide rapid test feedback to remain productive, which can be achieved by optimising the execution time of the employed test utilities. In order to guarantee the value of newly created test cases, it is important that they fail in the beginning. Hence, it

3 Agile Software Development

is then possible to verify that the newly added code is responsible for passing the test and that the test is useful [Bec03].

Additionally, it is a good practice to write only as much code as is necessary to fulfil the test case. To accomplish this it is suggested to take small steps and frequently check if the written code is sufficient to pass the test. In addition this procedure simplifies the process of detecting errors, as there are fewer lines of code to search through for new errors [Bec03] [Amb].

After each completed task, the code should be refactored to eliminate code of poor quality that is hard to maintain, such as duplicate code. With the assistance of the test suite, code improvements can be achieved very effectively and without fear of introducing regressions. A further strength of the test suite is that it can act as a detailed executable specification and contains working examples for using the code [Bec03] [Amb].

3.5 Continuous Integration (CI)

Continuous Integration (CI) is a great technique for reducing the risk of a software project and is an enriching supplementation for agile development practices. In traditional software development the integration process is an expensive and unpredictable process with many risks. By integrating and testing code after a few hours of work the integration process becomes a routine and its execution becomes natural to the involved developers [Bec99] [Fow06].

The short integration/build/test cycle helps to find bugs and collisions with other developers more quickly, because rapid feedback is provided. In addition, communication between team members is encouraged when different opinions about solving a certain problem emerges. Whenever a bug or collision arises the most important task is to fix it in order to return to a stable version that serves as a basis for further integrations. Besides, it enables developers to complete their tasks at full speed, since they have not to consider changes from fellow developers [Bec99] [Fow06].

The best results can be obtained by employing automated tests and by providing a dedicated machine to regularly perform integrations. In addition the dedicated machine serves as a reference platform for the development team and helps to reveal problems arising from differences in local development environments [Bec99].

4 Test Tools

“Software features that can’t be demonstrated by automated tests simply don’t exist,” concludes Kent Beck [Bec99].

The previous chapter suggests using automated tests for improving productivity when applying agile software development methodologies. Accordingly, this chapter introduces tools, which are suitable for performing automated tests on web applications. These tests can be divided into two groups: unit tests and functional tests.

Unit tests are intended for use by developers following a white-box testing approach that requires knowledge about the internal functionality of the system. The internal functionality is tested in small pieces, in the size of a module, a class, or a function, and is suitable as living documentation. This kind of testing behaves similarly for different programming environments and therefore the process for writing a unit test for Android applications, or web applications have a lot in common.

Functional tests on the other hand are intended for customers applying a black-box testing approach to verify that the system is capable of performing the product requirements. Thus, these tests have to use the user interface for interactions and evaluate the output to determine whether the system behaves as expected. In Extreme Programming (XP) and other agile software development methodologies these kinds of tests are also referred to as acceptance tests. It is apparent that testing tools for different types of applications do not have much in common, as they are dependent on the user interface of the application under test. For that reason this chapter concentrates on tools that are suited for web applications.

4.1 Unit Tests

This section focuses on unit testing tools intended for the PHP¹ scripting language, because it is very popular and widely used for building web applications including the Catrobat project community website. Moreover, all unit testing frameworks belonging to the xUnit family behave similarly and include almost identical functionality. The foundation of the xUnit family originated from Kent Beck's SUnit that was implemented for the Smalltalk² programming language and was later ported to others. Today almost every programming language has a testing framework belonging to the xUnit family [Fow].

A good unit test should satisfy the following properties, it should be automated and repeatable, it should be easy to implement, it should remain for future use, it should be able to be run by anyone, it should run at the push of a button, and it should perform quickly. Additionally, it is recommended that each test case is independent in order to identify occurring failures faster. If one or more of the mentioned properties are not met, then the tests are probably integration tests and not unit tests [Osho9].

A list with the most promising and most mature testing frameworks for the PHP scripting language belonging to the xUnit family contains four tools.

1. PHPUnit³
2. Atoum⁴
3. Enhance PHP⁵
4. SimpleTest⁶

The most extensively used and best-supported testing framework is PHPUnit, therefore the following examples and descriptions refer to it. Nevertheless, as already mentioned, a lot can be applied to the other testing frameworks as well [MDG11].

¹<http://php.net/>

²<http://smalltalk.org/>

³<http://phpunit.de>

⁴<https://github.com/atoum/atoum>

⁵<https://github.com/Enhance-PHP/Enhance-PHP>

⁶<http://simpletest.org/>

4.1.1 Architecture

Test Runner

The core of every unit testing framework is the test runner that is responsible for running unit tests and reporting results. For that purpose, PHPUnit comes with a command line tool, which offers many options for coping with all kinds of use cases. An overview of all available options can be obtained by running the command line tool with the *help* argument, which produces the output illustrated in Listing 2.

In order to execute tests, the path to a file or a directory that contains unit tests must be passed to the command line tool. A test case can have five possible outcomes, and for each test case a symbol representing the result is displayed. The five possible outcomes with the corresponding symbol in brackets are, it succeeded (.), an assertion failed (F), an error occurred (E), it has been skipped (S), or it is marked as incomplete (I). In Listing 1 an example output of a test run that performs 21 tests and 40 assertions is shown [Ber].

```

catroweb@webbox:~$ phpunit admin
PHPUnit 3.6.12 by Sebastian Bergman.

.....

Time: 1 second, Memory: 11.50Mb

OK (21 tests, 40 assertions)

```

Listing 1: Example output of PHPUnit test runner

Organisation

For organising the tests either the file system or a XML⁷ configuration file can be used. It is suggested that a naming convention is followed, which states that the suffix of the filename should be `Test.php`, for example `AdminTest.php`. Thus, the test runner is capable of detecting and executing tests without additional information about the filename.

The file system approach can be utilised with little effort, as no further configuration files are necessary. It is recommended that a folder is created

⁷<http://www.w3.org/XML/>

4 Test Tools

```
catroweb@webbox:~$ phpunit --help
PHPUnit 3.6.12 by Sebastian Bergman.

Usage: phpunit [switches] UnitTest [UnitTest.php]
       phpunit [switches] <directory>

--log-junit <file>           Log test execution in JUnit XML format to file.
--log-tap <file>            Log test execution in TAP format to file.
--log-json <file>           Log test execution in JSON format.

--coverage-clover <file>    Generate code coverage report in Clover XML format.
--coverage-html <dir>       Generate code coverage report in HTML format.
--coverage-php <file>       Serialize PHP_CodeCoverage object to file.
--coverage-text=<file>      Generate code coverage report in text format.
                             Default to writing to the standard output.

--testdox-html <file>       Write agile documentation in HTML format to file.
--testdox-text <file>      Write agile documentation in Text format to file.

--filter <pattern>          Filter which tests to run.
--testsuite <pattern>       Filter which testsuite to run.
--group ...                 Only runs tests from the specified group(s).
--exclude-group ...         Exclude tests from the specified group(s).
--list-groups                List available test groups.
--test-suffix ...           Only search for test in files with specified
                             suffix(es). Default: Test.php,.phpt

--loader <loader>           TestSuiteLoader implementation to use.
--printer <printer>         TestSuiteListener implementation to use.
--repeat <times>           Runs the test(s) repeatedly.

--tap                        Report test execution progress in TAP format.
--testdox                   Report test execution progress in TestDox format.

--colors                     Use colors in output.
--stderr                     Write to STDERR instead of STDOUT.
--stop-on-error              Stop execution upon first error.
--stop-on-failure            Stop execution upon first error or failure.
--stop-on-skipped            Stop execution upon first skipped test.
--stop-on-incomplete        Stop execution upon first incomplete test.
--strict                     Run tests in strict mode.
-v|--verbose                 Output more verbose information.
--debug                      Display debugging information during test execution.

--process-isolation          Run each test in a separate PHP process.
--no-globals-backup          Do not backup and restore $GLOBALS for each test.
--static-backup              Backup and restore static attributes for each test.

--bootstrap <file>          A "bootstrap" PHP file that is run before the tests.
-c|--configuration <file>  Read configuration from XML file.
--no-configuration           Ignore default configuration file (phpunit.xml).
--include-path <path(s)>     Prepend PHP's include_path with given path(s).
-d key[=value]               Sets a php.ini value.

-h|--help                    Prints this usage information.
--version                    Prints the version and exits.
```

Listing 2: PHPUnit usage information

4 Test Tools

specifically for the test suite and to retain the structure of the system under test (SUT). For reasons of convenience, it is important to adhere to the previously mentioned filename convention. This guarantees that the test runner can execute all tests within a directory at once by recursively traversing it.

When more control over the test suite is desired, the XML configuration file approach is favourable. It allows the creation of a test suite with selected tests and the ability to determine the execution order. Additionally, regularly used command line options from Listing 2 can be predefined, which is especially useful for filters and groups [Ber].

Annotations

In order to make use of groups and filters it is necessary to provide additional information, which can be accomplished by the means of annotations. This additional information, also called metadata, can be accessed at runtime and has no direct effect on the execution of the code. Annotations can be applied to classes, methods, and member variables and are used to supply information about dependencies, group affiliation, expected exceptions, and data providers [Ber].

However, there is no native support for annotations in PHP. For that reason doc comments, which were introduced for PHPDoc, which are an adaptation of Javadoc, are exploited to mimic annotations. As can be seen in Listing 3 they are basically block comments, but with the convention that they have to begin with `/**` and that every line has to start with a `*`.

```
1  /**
2   * @group projectDetails
3   * @dataProvider randomIds
4   */
5  public function testIncrementViewCounter($projectId) {
6      ...
7  }
```

Listing 3: A doc comment with group and data provider annotations

This distinction of comments is important, because the PHP lexer⁸ discards comments, but stores doc comments as metadata that can be accessed at runtime⁹.

⁸https://github.com/php/php-src/blob/master/Zend/zend_language_parser.y

⁹<http://www.php.net/manual/en/reflectionclass.getdoccomment.php>

Logger

As can be seen in Listing 1, the test results are displayed as raw text in the console. Consequently, this output format of the results is not well suited for further processing, such as supplying a continuous integration server.

Therefore, PHPUnit has an logger module included that is able to output the test results in a machine-readable format. Among the supported formats are XML, JSON¹⁰ and TAP¹¹. The code coverage tool of PHPUnit that is described later, also makes use of the logger module to format and output the coverage report [Ber].

4.1.2 The Basics

Fixtures

The test fixture is defined as the process of preparing a known state before running test cases, and restoring the original state when they are complete. This is important, because meaningful results can only be obtained if the original state is known. To generate a known state, actions like creating objects with given parameters, loading the database with a specific state, or creating temporary files and directories can be involved. A test class that extends a PHPUnit test case has two predefined methods for setting up a known state and three predefined methods for restoring the original state [Ber]. These methods have different scopes and purposes:

setUpBeforeClass is run once before any test case of the test class is executed and the preparations apply to all test cases in the class.

setUp is run before each test case and the preparations apply only to the following test case.

tearDown is run after successfully completed test cases.

onNotSuccessfulTest is run after unsuccessfully completed test cases.

tearDownAfterClass is run after every test case in the test class is completed.

In addition the pre- and post-conditions can be examined by the means of `assertPreConditions` and `assertPostConditions` to ensure that an expected

¹⁰<http://www.json.com/>

¹¹<https://metacpan.org/pod/release/PETDANCE/Test-Harness-2.64/lib/Test/Harness/TAP.pod>

4 Test Tools

state has been reached.

It is obvious that it is possible to share fixtures across tests with the `setUpBeforeClass` method, but that should be used rarely, because shared fixtures reduce the value of test cases, since dependencies are added. However, this is useful for keeping database connections open that can be used by all test cases of the test class [Ber].

Assertions

Assertions are essential for every unit testing framework, they are used to verify expectations and abort the execution in the event of a failure. The PHPUnit testing framework offers 40 different assertions to verify a wide range of expectations. Among them are the usual assertions, such as `assertTrue`, `assertSame`, and `assertGreaterThan`, but there are also assertions for specific cases, such as `assertJsonStringEqualsJsonFile`, `assertContainsOnlyInstancesOf`, and `assertStringStartsWith` [Ber].

This large choice of assertions is relevant, because it allows the testing of complicated assumptions with a single assertion. Additionally, it is good practice to avoid multiple assertions in one test case, because unit tests should fail for exactly one reason. If a test case has multiple assertions, then assertions succeeding to a failing assertion are not executed, and therefore additional information about the cause of the error is lost. Nevertheless, multiple assertions are acceptable, when they are used as guard assertions, which guarantee a certain original state [Osho9].

Dependencies and Data Providers

Dependencies between tests can be achieved with the use of the `@depends` annotation. A test can have one or more dependencies and the return value of a dependant test can be used as an argument in the actual test. If one of the dependant tests fail, the actual test is not executed [Ber].

Data providers are useful for repeating the same test with multiple inputs, they are assigned with the `@dataProvider` annotation. Any public method that returns an array of arrays or an object, which is traversable by the means of an iterator returning an array, can be referenced as a data provider. The values

4 Test Tools

of the array represent one test vector that is passed to the test function as arguments [Ber].

Output and Exceptions

To test the standard output of a method, PHPUnit collects the outputs of all encountered calls to `echo` and `print` by the means of the PHP output buffer¹². When the test case reaches the end, the buffered output is asserted to the expectation stated by the string argument of the `expectOutputString` function [Ber].

There are various ways to test exceptions in PHPUnit. Firstly, annotations can be used to verify that an expected exception type, code, or message occurred. Secondly, there is a `setExpectedException` function that works in the same way as the previously explained method to test an expected output. It is also capable of testing the exception type, code, and message. Finally, an expected exception can be caught in order to successfully complete the test case and otherwise, if the exception was not triggered, a failure can be emitted [Ber].

4.1.3 Advanced Techniques

Stubs and Mock Objects

In the same way as it is favourable to avoid multiple assertions, it is considered to be more informative to reduce the sources of possible failures by isolating the functionality under test. In order to isolate functionality, the dependencies to other objects or external resources can be replaced by stubs or mock objects. They can be imagined as a simulation of an object or a resource that returns simulated values of complicated computations or resources that are difficult to access. This improves on the one hand the run time and on the other hand the robustness against changes of the unit test [MDG11].

The biggest difference between stubs and mock objects is that mock objects perform additional verifications in regard to the communication of the simulated object. Therefore, mock objects are able to fail tests but stubs cannot fail tests. This leads to the conclusion that a test case should only use one mock object,

¹²<http://www.php.net/manual/en/ref.outcontrol.php>

4 Test Tools

because as pointed out before, unit tests should fail for exactly one reason. To create a stub or a mock object PHPUnit has a `getMock` method to automatically create the object. Typically this generates a stub, but when expectations are added the object becomes a mock object [Osho9] [Ber].

Database Testing

Almost every web application relies on a database to store persistent data. Therefore, it is viable to provide ways to efficiently test these interactions. It is difficult to integrate database interactions in unit tests, because for each test case, a known state needs to be prepared and these are complex to set up and to maintain. In order to accomplish database interactions, the following issues need to be addressed.

The database schema and tables must be known and it must be possible for them to be automatically created. In the set up phase of the test case data rows that are required by the test must be inserted into the database to produce a known state. After the test run the produced database state needs to be verified. Finally, the performed actions must be cleaned up, so that the subsequent test case has a clean database state. Otherwise, it can lead to undesired side effects, which negatively affect the result of the subsequent tests [Ber].

One option for performing database tests is to use the previously mentioned stubs and mock objects by simulating the interactions. Another possibility is to use the DbUnit extension of PHPUnit, which provides an abstraction layer that simplifies the aforementioned issues. It is capable of keeping a database connection open for a complete test run to reduce the runtime of the unit tests. Then, methods are included to create expectation data sets and to validate expected database states against expectations, which can be expressed as XML, YAML¹³, CSV files¹⁴ or a PHP array. Additionally, the set up and tear down process have convenience functions such as automatically inserting specified data rows into the corresponding tables [Ber].

¹³<http://yaml.org/>

¹⁴<https://tools.ietf.org/html/rfc4180>

Code Coverage

Code coverage is a measure for indicating the lines of code that are executed when running the test suite. It is suited to identifying code that is not tested and can be used to visualise the progress of the project by the means of a continuous integration (CI) server. The code coverage tool that is included in PHPUnit employs the PHP Xdebug¹⁵ extension for the statement coverage functionality. A coverage report can have different output formats as discussed earlier. When the coverage report is outputted as HTML¹⁶, executed lines are highlighted green, unexecuted lines are highlighted red and lines that can never be executed are highlighted grey [Ber].

A 100% unit test code coverage is no guarantee that the finished product works as expected. This is because unit tests are designed to test independent units of code and not the interactions between them. In order to test that the separate units work together as desired, functional or integration tests must be applied. Nevertheless, a high code coverage is valuable for catching regressions early in development [MDG11].

4.2 Functional and Integration Tests

Functional and integration tests have a lot in common, but they focus on different aspects. Functional tests are used to test that the software meets the customer requirements, which often corresponds to a user story in agile software development. Integration tests are used to test that interactions between different components of the system, such as the file system or the database, work together as expected, which is often indicated by the presence of a data flow. Both test methods perform tests on the software as a whole, which means that no stubs and mock objects are allowed. Therefore it is possible to employ the same tools to achieve integration, or functional tests [Raj].

Modern web applications make heavy use of client side JavaScript to add functionality and to reduce the load on the web servers. Earlier, protocol driven test tools that mimic a browser by simulating the HTTP protocol¹⁷ were sufficient to test web applications, but they are not able to test JavaScript interactions

¹⁵<http://www.xdebug.org/>

¹⁶<http://www.w3.org/HTML/>

¹⁷<http://www.w3.org/Protocols/rfc2616/rfc2616.html>

4 Test Tools

[BGH07]. In addition, the content is often separated from the presentation and is asynchronously retrieved to reduce the data traffic. Moreover, it becomes increasingly important that the application is usable on different viewport sizes, which arises from diverse form factors relating to mobile phones up to desktop PCs. Beyond that, additional testing efforts are required due to different browser engines, which can produce varying results in certain situations.

Testing frameworks, which are capable of performing functional tests on web applications that support JavaScript and asynchronous requests to retrieve and dynamically display content, are limited. Among them are the following five testing frameworks, the first four are open source and the last one is commercial.

1. Selenium¹⁸
2. Watir¹⁹
3. Windmill²⁰
4. WebTest²¹
5. Sahi²²

All these frameworks are web automation tools that perform predefined steps in an actual web browser, except for WebTest, which uses HtmlUnit²³, a headless web browser implementation with basic JavaScript support. Overall, they work similarly, but some of them have constraints in regard to applicable programming languages and favourable working environments. The most general testing framework is Selenium, therefore the following examples and descriptions refer to it.

4.2.1 Architecture

Selenium Core

This is the original library also referred to as Selenium 1.0 created by Jason R. Huggins that serves as a basis for the later described Selenium Integrated

¹⁸<http://docs.seleniumhq.org/>

¹⁹<http://www.watir.com/>

²⁰<http://www.getwindmill.com/>

²¹<http://webtest.canoo.com/>

²²<http://sahipro.com/>

²³<http://htmlunit.sourceforge.net/>

4 Test Tools

Development Environment (IDE) and Selenium Remote Control (RC). It contains a JavaScript driven test runner that is capable of driving interactions with the web application and evaluating JavaScript in an automated manner. The command set used is called *Selenese* and has a very simple syntax, it consists of the command name and has a maximum of two parameters [BGH07].

In order to ensure compatibility with multiple browser platforms and therefore increase the gain achieved by using automated tests, the Selenium core relies on standard HTML to define and run tests. A test case is defined by a sequence of commands, which are saved in a HTML table, where a row corresponds to one command. Each row has three columns to store the corresponding arguments of the Selenese command. The test suite also consists of a HTML table, but with the difference that each row contains a hyperlink to a test case in order to add the referenced test case to the test suite [BGH07].

To execute a test suite the included test runner consisting of static HTML and JavaScript is employed. The use of a JavaScript powered test runner guarantees that the tests work in almost every browser. However, it has limitations due to the sandboxed JavaScript environment of modern browsers, such as the same-origin policy to prevent cross-site scripting. To avoid failures, which are attributed to this limitation it is necessary to place the tests in the same location as the application under test. Thus, this can lead to complicated test environments if the application is distributed over several locations. Another limitation arises, when programming logic is required to perform more complex testing tasks [Hun+10].

Selenium Integrated Development Environment (IDE)

The Selenium IDE offers the simplest way to create test cases, but it is unfortunately only available as an extension to the Firefox web browser. It is able to record the actions performed in the browser, such as clicking on a link, entering text in an input field, and further interactions with the web application. These actions are saved as a test case using the previously discussed Selenese commands and can be repeated in an automated manner as often as required [BGH07] [Bur12].

The recording of a test case does not include assertions. To add assertions, the Selenium IDE offers the possibility of bringing up a context menu with corresponding options when clicking on an element. Alternatively, assertions can be manually added to the test case [Hun+10].

4 Test Tools

Another useful feature is the capability to export recorded test cases to other programming languages including C#, Java, Python, and Ruby. This vastly reduces the time needed to create test cases that do not rely on Selenese commands, which applies to subsequently described test methods. However, a disadvantage of the IDE is that complex testing tasks cannot be handled, for example, when an iterator is required to test each element of a variable list. In order to cope with these situations one of the mentioned programming languages can be used to drive the tests with the Selenium Remote Control or Selenium WebDriver [RF11] [Bur12].

Selenium Remote Control (RC)

Selenium RC is applied for more sophisticated test suites that rely on automated execution and evaluation of tests, and is important for use with continuous integration practices. Considering that the tests are driven by programming languages and not Selenese commands, extensive test cases are possible, and well known testing frameworks like JUnit or TestNG can be used. If the test suite has one of the subsequent requirements it is recommended that Selenium RC is employed, as they cannot be achieved with Selenium IDE: conditional statements, iterations, logging and reporting of test results, error handling of particularly unexpected errors, database testing, test case grouping, re-execution of failed tests, test case dependencies, and capturing the current application state by means of a screenshot in the event of an error [Hun+10].

It is designed as a client-server architecture that allows the tests to run on two separate machines, but it is also possible to run the server and client on the same machine. The server is responsible for starting web browser instances in which the tests are run, receiving and executing Selenese commands, and returning the results of the execution. The client acts as an interface between the supported programming languages and the server, it is responsible for sending the commands to the server and returning the received results back to the program [Hun+10].

Another important task of the server is to make the test runner available and to avoid the previously mentioned same origin policy to make it work. This can be done by means of the proxy injection mode that uses a HTTP proxy to mask the web application with an embedded Selenium core within a fictional URL. A second possibility is to launch the browser in a special mode called

4 Test Tools

heightened privileges, that grants unusual rights to the web application, such as cross site scripting, or accessing file upload inputs [Hun+10].

Selenium WebDriver

Selenium WebDriver also referred to as Selenium 2.0 is a new approach to achieve browser automation that makes use of native interfaces to bypass the limitations of JavaScript driven tests, which are employed by Selenium core. The use of Selenium WebDriver is recommended, when support for page navigation, drag-and-drop, AJAX-based user interface elements, handling multiple frames, multiple browser windows, popup windows, or alert boxes are a main concern for the test suite [Hun+10].

The WebDriver API uses the accessibility API of the browser to run the tests and provides a more concise and object oriented interface for writing tests. It offers the ability to run the tests locally or remotely and provides an interface for Selenium RC that guarantees backwards compatibility to reduce porting efforts. Recently the World Wide Web Consortium (W3C) that is the most recognised organisation for defining web standards has started to work on the standardisation of the WebDriver API²⁴. As a result, the responsibility of the implementation of a working WebDriver API is transferred to the browser developers, which can lead to a more stable and better-supported interface [Bur12].

Selenium Grid

Selenium Grid should be used for two possible reasons, either to speed up a slow-running test suite or to test various browser platform combinations in a reasonable way. It is possible to run test cases that are written for Selenium RC or Selenium WebDriver in a grid setup, as illustrated in Figure 4.1. The core of the grid setup is the hub that is responsible for managing client requests and incoming nodes. Grid nodes are instances, which offer their service to execute tests on specific browsers and platforms. In order to run tests, the client sends a request with the required browser capabilities to the hub, and if there is an instance available that satisfies them, it is assigned to the client to perform the tests [Hun+10] [Bur12].

²⁴<http://www.w3.org/TR/webdriver/>

4 Test Tools

The biggest time savings stem from parallel test execution, since an arbitrary number of machines providing test instances can be connected to the hub. However, it is necessary to define the level at which the tests are able to run in parallel to make sure that there are no mutual interferences. The two possible options are to align the tests at class or method level [Bur12].

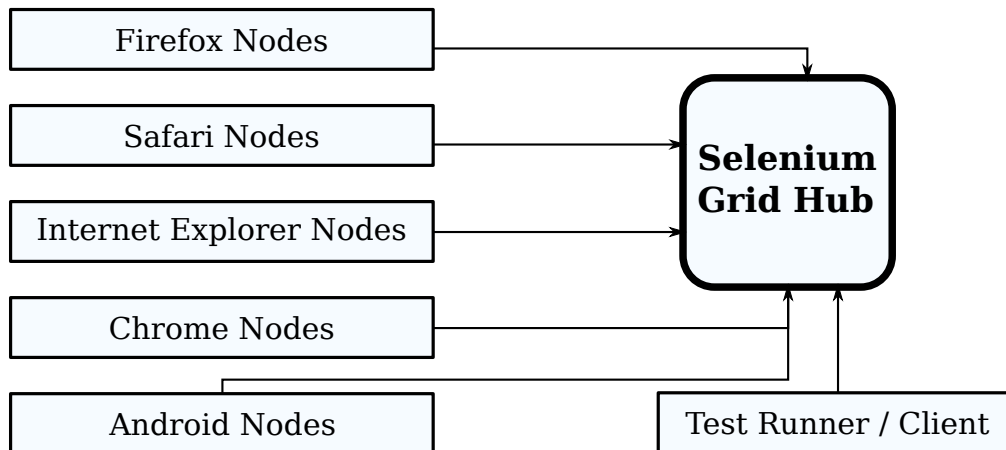


Figure 4.1: The structure of a Selenium Grid setup

4.2.2 The Basics

The HTML test suite is not suitable for satisfying more advanced requirements, as it has limitations. Therefore, it is recommended that Selenium RC or Selenium WebDriver are used in combination with a unit testing framework like JUnit or TestNG. Thus, earlier discussed capabilities such as fixtures, dependencies, data providers and error logging are available and they behave similarly to the PHPUnit equivalents.

In contrast to the Selenese commands there are no built-in assertions in Selenium RC and Selenium WebDriver. Therefore it is necessary to employ the assertions from the selected unit testing framework [Hun+10].

Accessing Elements

In order to perform actions, the most important task is to find and access elements of the web application. Generally, elements can be located by means

4 Test Tools

of an identifier, name, CSS class, or XPath expression. It is recommended that elements are accessed using a unique identifier although the name and CSS class attribute also work. However, these two attributes can have multiple occurrences, which results in the necessity of further selections [Hun+10] [Bur12].

If these methods are not sufficient for accessing the required element, another method that makes use of the XML Path (XPath²⁵) language can be employed to access elements. It is a query language that is able to locate elements from the Document Object Model (DOM²⁶) of the browser. Nevertheless, it should only be used when there is no other possibility, because it is an extremely costly operation compared to the previous methods [Bur12].

Verifications and Assertions

It is in the nature of web applications that they are served by web servers. Therefore, it is necessary to consider a delay for retrieving the contents. In order to wait for certain conditions Selenium offers several commands to perform explicit waits, which are able to await page loads, frame loads, and states like the presence or absence of alert messages, elements, and texts. These commands have a default timeout of 30 seconds, which can be adapted to specific needs. Whenever a condition cannot be completed and the event is within the timeout time, then the execution of the command is repeated and no log or error message is emitted [Bur12].

With regard to gathering information about unexpected behaviour, assumed conditions are either verified or asserted. The difference between verifications and assertions is that if an assertion fails, the execution is aborted and no further steps are taken. In contrast, if a verification fails, the failure is logged and the execution of the test continues. This can be useful for getting a better insight into what is causing a problem, because checks after a failure are performed since the test is not aborted. Therefore more information about the overall system is available [Hun+10].

Thus, a general Selenium test has to go through the following steps. Firstly, open a page and wait until it is loaded, secondly locate the required element, and finally perform an action on the element, or check expectations by the means of verification or assertion. The following capabilities are prepared to examine expectations, which are able to check if: an element is present, an element is not

²⁵<http://www.w3.org/TR/xpath20/>

²⁶<http://www.w3.org/DOM/>

present, a text is available, an attribute is correctly set, a checkbox is checked, an alert occurred, or the website title is correctly set [Hun+10] [Bur12].

4.2.3 Advanced Techniques

Asynchronous JavaScript and XML (AJAX)

In order to provide a reasonable user experience in modern web applications it is inevitable that Ajax is used to asynchronously load content and dynamically create elements. The key problem is that elements of the web application are modified without a page refresh and consequently most Selenium wait commands have no effect. Therefore, many invalid test failures are generated, which are caused by components that have not yet been created. However, there are commands, like `waitForElementPresent` or `waitForVisible`, to prevent these kinds of failures by waiting until a certain element is present or visible [Hun+10] [Bur12].

Unfortunately, these commands are no longer available in Selenium WebDriver, thus another solution for this problem is needed. The most obvious solution for solving timing issues is to pause the execution with a sleep command. However, this is not recommended, because it breaks an important rule that states that automated tests should run as quickly as possible. A more elegant solution is to apply an implicit wait, which ensures that whenever an unexpected failure occurs an assigned timeout is completed before the test execution continues. Thus, failures due to elements that have not yet been created can be prevented without using explicit waits, but if the test contains many consecutive failing commands, the test run is stretched [BGH07] [Bur12].

Finally, an expected condition can be defined and awaited. For example Listing 4 shows that the expected condition is evaluated by JavaScript, which is capable of directly interacting with the web application. In this case jQuery²⁷ is used to perform Ajax requests, therefore a member variable `active` can be accessed to determine the number of pending requests. If this variable equals zero, it means that all asynchronous activities are finished [BGH07].

²⁷<http://jquery.com/>

4 Test Tools

```
1 void ajaxWait() {
2     Wait<WebDriver> wait = new WebDriverWait(driver(), Config.TIMEOUT_WAIT);
3     wait.until(jQueryReady());
4 }
5
6 ExpectedCondition<Boolean> jQueryReady() {
7     return new ExpectedCondition<Boolean>() {
8         public Boolean apply(WebDriver driver) {
9             return ((Boolean) ((JavascriptExecutor) driver).executeScript(
10                 "return (window.jQuery.active == 0);");
11         }
12     };
13 }
```

Listing 4: Selenium wait for an expected condition evaluated by JavaScript

Mobile Devices

The WebDriver API is also able to perform tests on mobile devices running Android or iOS. However, it is sufficient to target Android devices to evaluate mobile user experience, because it has more browser engines available and they are responsible for the biggest differences in rendering. Among them is the WebKit browser engine, which is the only option for iOS devices, therefore the additional efforts to make the tests applicable on iOS is not particularly beneficial.

For testing purposes, either an emulator or a real device can be used. Previously, it was recommended to run the tests on a real device. The reason was that the Android emulator was very slow, since the available versions were only compatible with ARM-based architectures. However, that has changed with the release of new versions, which are compatible to x86-based architectures. In order to run tests, it is necessary to prepare the emulator or the physical device by installing the Selenium Server for Android²⁸ [Bur12].

Alternatively, a more straightforward approach with little set up effort can be applied, which controls the viewport size for imitating mobile devices. A single line of code can achieve this, as shown in Listing 5. This is sufficient for checking basic functionalities of web applications when applying responsive web design.

²⁸<http://selendroid.io/>

4 Test Tools

```
1 void openMobileLocation(String location) {  
2     driver().get(this.webSite + location);  
3     driver().manage().window().setSize(new Dimension(320, 480));  
4 }
```

Listing 5: Selenium open page with defined viewport size

Headless Testing and Screen Captures

Because the tests are running in an actual web browser, the machine becomes unusable while they are executed. That is caused by surfacing browser windows, which steal the focus of the currently selected window, since every test is performed in its individual browser instance. On machines operated by Linux, this behaviour can be suppressed by using a headless display server like the X11 Virtual Frame Buffer (XVFB²⁹), which allows applications to run in the background. This approach is also very useful for servers, which usually do not have a physical display hardware available [Bur12].

Occasionally, the cause of failing tests is not immediately obvious. Therefore, it is convenient to save the state of the failing web application by the means of a screenshot to examine the error later. Considering headless, parallel running, or distributed tests this can reveal substantial information, which is otherwise lost. An easy way of taking screenshots in the event of an error is to catch exceptions that were thrown by failing assertions. Selenium offers an interface that uses browser libraries for capturing screenshots, which returns either a file, or a base64 encoded string that can be transferred over the network [Bur12].

²⁹<http://www.xfree86.org/4.0.1/Xvfb.1.html>

5 Behaviour Driven Development

“Behaviour’ is a more useful word than ‘test’,” stresses
Dan North [Noro6].

Behaviour Driven Development (BDD) is an evolution of previously mentioned agile software development practises that aim to solve recurring misconceptions and uncertainties. It is influenced by Extreme Programming (XP) and especially by Test Driven Development (TDD), Continuous Integration (CI), and Acceptance Test–Driven Planning (ATDP). In traditional projects the main reason for failure is miscommunication. Therefore, a primary concern of BDD is to improve communication between all participants of the project including customers, analysts, designers, testers, and developers. An effective measure is to create a ubiquitous language to shape a shared vocabulary. As well as the ability to specify requirements in “plain English” to make them accessible to all team members [Noro6] [Che+10].

In regard to method names, it suggests the use of descriptive sentences for test method names in order to reveal their purpose. These names should exactly describe a single behaviour of the system and the corresponding test should only test as much functionality as expressed in the method name. This approach instructs developers to choose useful names, which generates an additional gain for future development in terms of readable documentation and fast error detection. Another essential point is to create valuable software, therefore the development of new features is prioritised after business value and features without business value are omitted [Noro6].

Furthermore, it adheres to an important principle, which all agile software development methodologies have in common. That is to write simple and clean code, which is limited to satisfy the test or fulfil the expected behaviour. In addition, as the plain text requirements can be executed automatically, these can be used as acceptance tests, and subsequently as regression tests [Noro6].

The most significant benefits of this second-generation agile development methodology are the development of a ubiquitous language, automated acceptance

tests, responsive documentation, also referred to as living documentation, and an improved software quality. Besides, it is easy to adopt for teams with experience of other agile software methodologies.

5.1 The Basics

5.1.1 Express Requirements

As mentioned, TDD had a great impact on BDD, however a substantial difference concerns the usage of the word “test”. The use of this word makes developers suppose that data types and internal structures should be covered by unit tests, which leads to tightly coupled tests and not required dependencies. Furthermore, these tests fix the implementation instead of the behaviour, which increasingly leads to less resilient tests. In regard to reporting errors, although the behaviour of tested objects has not changed, a considerable slow down in the development progress is apparent. Therefore, the word “test” was completely removed from the development process to reduce the described effect [Nor06] [Che+10].

There are two distinguishable ways of applying BDD: *SpecBDD* and *StoryBDD*. *SpecBDD* is comparable to TDD with the main difference being that when expressing specifications the word “test” is dismissed and replaced with “should”. In contrast, *StoryBDD* is suited for functional tests by executing scenarios, which are expressed as user stories.

An important criterion for producing valuable software, is to meet vital customer requirements and adhere to them throughout the entire project. As customer requirements usually contain behaviour, they can be captured as a user story. It is crucial that the user stories are consistent in order to be suitable for subsequent automatic processing. Therefore, it is suggested that a story template is used that provides the basic structure. An example of a simple story template is shown in Listing 6. It uses a semi-formal format, which is often embedded in a domain specific language (DSL), and allows the specification of requirements in plain text. This has the advantage, that requirements are easier to understand and comprehend for non-technical team members [Nor] [WH12]. Furthermore, due to the special structure of the user stories, they can be automatically executed by tools designed for that purpose (see Section 5.3) in order to obtain an executable specification.

5 Behaviour Driven Development

In BDD a user story is referred to as a *feature* and should meet the following properties: cover only a single behaviour, have a concise and precise title describing the story, contain background information, and define one or more acceptance criteria [Nor]. The background information serves to illustrate the business value of the feature by describing the expected benefit and identifying the recipient. For instance, the narrative section presented in Listing 6 offers a great template for providing the essential information. The scenarios are concrete examples, which represent the acceptance criteria of the associated feature. As can be seen in Listing 6, they have a title and follow a given-when-then structure to describe the scenario [Nor] [Che+10].

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
  And [some more context]...
When [event]
Then [outcome]
  And [another outcome]...

Scenario 2: ...
```

Listing 6: Example of a story template (taken from [Nor])

5.1.2 Gherkin Syntax

Gherkin is a widely used domain specific language (DSL) to express features for BDD that is adopted by a variety of related tools. The most important property is that it maintains human readability, as it mainly consists of plain text descriptions, which subsequently are utilised as documentation. Its syntax is composed of a set of keywords, which shape the basic structure that is required to automatically execute user stories. The essential keywords to create an automatically executable feature are *Feature*, *Scenario*, *Given*, *When*, and *Then*. Additional optional keywords are *Background*, *Scenario Outline*, *Examples*, ***, *And*, and *But* [WH12].

Feature holds the title of the feature and is followed by a narrative description, as shown in the example of a story template in Listing 6.

Scenario holds the title of the scenario and is followed by an acceptance criteria in the given-when-then structure.

Background holds steps, which are executed for all scenarios in a feature.

Scenario Outline is used in combination with **Examples** for repeating scenarios with different input or output values.

The remaining keywords are applied to specify the acceptance criteria of the feature. Each line starting with one of these keywords represents a step definition, which are processed consecutively to evaluate the acceptance criteria. Step definitions map the plain text description of the steps to programming code that is responsible for performing the described action. A significant characteristic of step definitions is that they are not scoped, which means that every specified step definition is globally available [Che+10] [WH12].

Considering that BDD is a tool to encourage team communication, a crucial aspect is the ability to write features in native language. Therefore, the Gherkin syntax is designed to support internationalisation and the mentioned keywords are available in multiple languages [WH12].

5.1.3 Workflow Cycle

The workflow of BDD reveals that the emphasis lies on the design practice and not on the test practice. It follows an outside-in process and strongly supports best practices of existing agile software development methods. In the following section the workflow cycle is described, which is illustrated in Figure 5.1.

The outer cycle corresponds to functional testing (StoryBDD) and the inner cycle corresponds to unit testing (SpecBDD). Every iteration starts in the outer cycle with the assessment of the acceptance criteria by creating a failing scenario (red) [Che+10].

Then the process moves to the inner cycle that follows traditional TDD. This consists of three steps, namely: write a failing test to demonstrate missing functionality (red), make the test pass by satisfying the expectation (green), and refactor the generated code to clean up duplications and to improve readability (dotted) [Oshog].

When the inner cycle is complete, the process continues in the outer cycle with the evaluation of the acceptance criteria, which verifies that the code is

5 Behaviour Driven Development

working as intended (green). On success, it is followed by refactoring (dotted) and the process can be repeated until the feature is complete [Che+10].

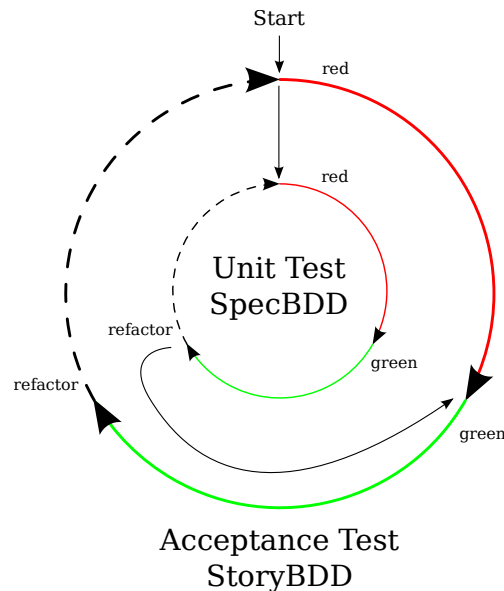


Figure 5.1: BDD workflow illustrated in a cycle (adapted from [Che+10])

5.2 Benefits

5.2.1 Ubiquitous Language

In *Agile Testing: A Practical Guide for Testers and Agile Teams* [CGo8] the collaboration with customers is described as a key success factor. Furthermore, it recommends encouraging direct communication between developers and customers as often as possible to reduce the risk of missing requirements and misunderstandings. To overcome the language barrier it suggests employing testers, since they have knowledge of both: the domain language, and technical language.

That aside, a more effective method is to adopt a consistent vocabulary in the project that is used by all participants. Usually, in larger projects a business

analyst is involved. The person with the business analyst role has the responsibility of promoting a shared vocabulary and to ensure that it is used in the user stories. A shared vocabulary (also called *ubiquitous language*) has to develop independently for each project, as it has to be adapted to the needs of different domains and the participating team members. Additionally, all team members have to agree on the use of certain vocabularies, in order to strengthen the commitment for employing the shared vocabulary [AM07] [Che+10].

Once a ubiquitous language has been established successfully, the benefits are fewer mistakes resulting from misunderstandings and a more motivated team. The motivation stems from more effective communication between business-facing and technology-facing parties when discussing new features or upcoming difficulties. Subsequently, it is important to evolve the language by keeping it consistent and up to date by incorporating frequent activities of the target audience [Adz11] [WH12].

5.2.2 Automated Acceptance Tests

In contrast to software development, it is relatively easy to determine when the construction of a building or the preparation of food is completed. In order to visualise the progress in a software project, automated acceptance tests, as they are applied in BDD, can be used. The acceptance tests are not only user-centric, but also customer-centric and can reveal whether the customer's intentions are satisfied. Whereas other agile development methods such as extreme programming (XP) do not promote automated acceptance tests, nonetheless they can be integrated in the development process. Certainly, the use of automated acceptance tests is highly recommended, as they are regarded as best practice [CG08] [Che+10].

In BDD the acceptance criteria is included in the concrete examples of the user stories and is therefore available from the beginning without additional effort. As a result, the automated acceptance tests emerge from the automated execution of the user stories, which usually consist of several concrete examples, each having an acceptance criterion for evaluation [WH12].

5.2.3 Living Documentation

In order to transfer knowledge or to determine system functionalities, the documentation of a project is an excellent source of information. In many projects the creation of documentation has a low priority, and is therefore often written all in one go at certain development stages. Subsequently it loses synchronisation to the actual code and becomes more and more ineffective as information source. A reason for this is costly maintenance resulting from frequent refactoring and short development cycles, but also pending tasks with higher priority cause update delays. Accordingly, relying on outdated documentation can have a counterproductive effect and the only remaining option for getting accurate information is to examine the system's source code. Evidently, this situation is not ideal for teams, which have members with little or no programming knowledge [Adz11].

In BDD the previously explained plain text features are used as documentation. This leads to the term *living documentation*, since, whenever a feature is updated, the documentation is updated too. The advantages are that it is an accurate representation of the real state of the application, it is as reliable as the source code of the system, and it is easy for all team participants to access and understand. In addition the documentation adapts with the business model over time, as the features are tightly coupled with the software and make use of ubiquitous language [Adz11] [WH12].

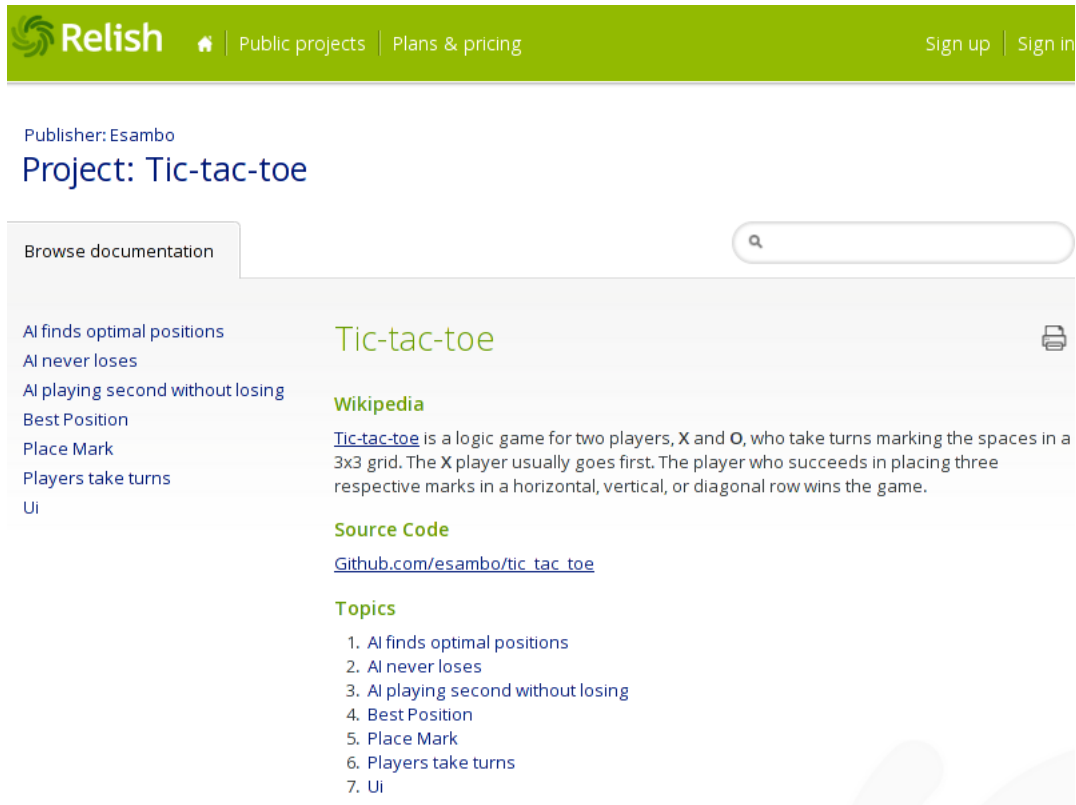
An example of living documentation covering the well-known logic game *Tic-Tac-Toe*, can be seen in Figure 5.2. The employed web service is called *Relish*¹ and offers a platform for publishing and sharing living documentation. It aims to present it in the same way as traditional specification documents [WH12].

However, *Relish* is going to be replaced by *Cucumber Pro*², a new service that offers the same features with the incorporation of many improvements. The most noticeable improvement concerns editing and reporting capabilities which are accessible from a web interface. Therefore it is a great addition to the development environment and especially well suited for business-facing team members [HBW].

¹<https://www.relishapp.com/>

²<https://cucumber.pro/>

5 Behaviour Driven Development



The screenshot shows the Relish documentation page for a project named 'Tic-tac-toe'. The page has a green header with the Relish logo and navigation links for 'Public projects' and 'Plans & pricing'. Below the header, it identifies the publisher as 'Esambo' and the project as 'Tic-tac-toe'. A search bar is visible in the top right. On the left, there is a sidebar with a 'Browse documentation' tab and a list of topics: 'AI finds optimal positions', 'AI never loses', 'AI playing second without losing', 'Best Position', 'Place Mark', 'Players take turns', and 'Ui'. The main content area displays the title 'Tic-tac-toe' with a print icon. It includes a 'Wikipedia' section with a brief description of the game, a 'Source Code' section with a link to 'Github.com/esambo/tic tac toe', and a 'Topics' section listing the same seven topics as the sidebar.

Figure 5.2: Living documentation of Tic-Tac-Toe on Relish [Sam]

5.2.4 Software Quality

An aspect of software quality is to build software that is useful to the customer. This is achieved by specifying multiple acceptance criteria for software features, which helps to focus on customer expectations [Che+10].

As shown in Figure 5.1, the workflow follows an outside-in process that is particularly suited to improve and evolve the software design in small steps. Furthermore, since testability is incorporated from the beginning, the produced code is not as tightly coupled as in traditional projects and dependencies are reduced. Additionally, the workflow encourages the creation of small and

simple methods, which simplifies the maintenance as a further consequence. The refactor-step, an integral part of the workflow, promotes the reduction of unnecessary code [Tat] [Che+10].

5.3 Related Tools

The original tool that implemented the idea of BDD is *JBehave*, it is based on JUnit and replaced all vocabulary referring to a test with words promoting behaviour. Another tool that had a strong influence on BDD is *Cucumber* that introduced the widely used Gherkin syntax [Noro6].

In Table 5.1 are tools listed, which are capable of performing StoryBDD. Additional information is given about the employed programming language and whether Gherkin syntax is supported. As can be seen, this table contains more than one tool, whose name contains Cucumber. A reason for this is the available Cucumber Technology Compatibility Kit³, which simplifies the creation of compatible implementations.

In addition Cucumber includes a wire protocol that allows the implementation of the core component, which is responsible for the execution of the steps of a scenario, in different programming languages. Table 5.2 lists available implementations of the wire protocol with the corresponding programming languages. Finally, Table 5.3 lists tools to practise SpecBDD and provides information about the associated programming language.

³<https://github.com/cucumber/cucumber-tck>

5 Behaviour Driven Development

Tool	Platform	Gherkin	Further Information
Behat	PHP	Yes	http://behat.org/
Behave	Python	Yes	https://pythonhosted.org/behave/
Cucumber	Ruby	Yes	http://cukes.info/
Cucumber.js	JavaScript	Yes	https://github.com/cucumber/cucumber-js/
Cucumber-JVM	Java	Yes	https://github.com/cucumber/cucumber-jvm/
Concordion	Java	No	http://www.concordion.org/
Coulda	Ruby	No	http://coulda.tiggerpalace.com/
Freshen	Python	Yes	https://github.com/rlisagor/freshen/
JBehave	Java	No	http://jbehave.org/
Lettuce	Python	Yes	http://lettuce.it/
NBehave	.NET	Yes	http://nbehave.org/
PHPUnit	PHP	No	http://phpunit.de/
RBehave	Ruby	No	http://dannorth.net/2007/06/17/introducing-rbehave/
SpecFlow	.NET	Yes	http://www.specflow.org/
StoryQ	.NET	No	http://storyq.codeplex.com/
Vows	JavaScript	No	http://vowsjs.org/

Table 5.1: Test tools capable of performing StoryBDD

5 Behaviour Driven Development

Tool	Platform	Further Information
Clucumber	Common Lisp	https://github.com/antifuchs/clucumber
Cucumber-CPP	C++	https://github.com/cucumber/cucumber-cpp
Cucumber-Lua	Lua	https://github.com/cucumber/cucumber-lua
Cuke4AS3	ActionScript	https://github.com/flashquartermaster/Cuke4AS3/
Cuke4Nuke	.NET	https://github.com/richardlawrence/Cuke4Nuke/
Cuke4PHP	PHP	https://github.com/olbrich/cuke4php/
Frank nStep	Objective-C .NET	http://www.testingwithfrank.com/ https://github.com/clearwavebuild/nStep

Table 5.2: Implementations of Cucumber's wire protocol

Tool	Platform	Further Information
Buster.JS JDave	JavaScript Java	http://docs.busterjs.org/en/latest/ http://jdave.org/
JsSpec MSpec	JavaScript .NET	http://code.google.com/p/js-spec/ https://github.com/machine/machine-specifications/
NSpec PHPSpec	.NET PHP	http://nspec.org/ http://www.phpspec.net/
RSpec Spec	Ruby Python	http://rspec.info/ https://github.com/bitprophet/spec/

Table 5.3: Test tools capable of performing SpecBDD

6 Guided Procedure

“The hardest single part of building a software system is deciding precisely what to build,” assures Frederick Brooks [Jr95].

This chapter demonstrates the guidance in the development process with behaviour driven development (BDD) by the means of a simple example. The example discusses the creation of a shopping cart with basic functionality, similar to the shopping carts used on several online shops. *Cucumber* is employed to drive the development in this example. It is chosen because of the great influence on BDD and the support of the widely used Gherkin syntax.

The first step in the development process is the creation of an empty folder that is intended for project related files. Running the Cucumber command line tool in that empty folder yields the suggestion to create a folder named *features*, which is the default location for storing feature files. With the folder in place, rerunning Cucumber results in the output that zero scenarios with zero steps were executed. The corresponding output is shown in Listing 17. That means everything is prepared to start following the BDD cycle shown in Figure 5.1.

6.1 Create Features

The red part of the outer circle gives the instruction to create a failing acceptance test. Therefore, the most valuable behaviour, which is missing from the system, is determined and written down as a new feature. In this example, the most important functionality is the ability to add products to the shopping cart. Therefore, a file named *add_products.feature* containing the description of the feature is put in the previously created features folder. The description follows the Gherkin syntax based on the story template from Listing 6.

This example covers four scenarios including the following activities: add a product to the shopping cart, get the name and price of a product in the shopping cart, calculate the total sum of all products in the shopping cart, and

6 Guided Procedure

remove accidentally added products from the shopping cart. When writing scenarios, it is important to take into account that no shared states exist. Therefore, scenarios must be executable independently. In Listing 7 an exemplary feature with the described scenarios is shown.

```
Feature: Add product to shopping cart

As a customer
I want to add selected products to my shopping cart
So that they are collected and listed for a purchase

Scenario: Add a product to my shopping cart
  Given the shopping cart is empty
  When I add a product to the shopping cart
  Then the shopping cart should contain 1 item

Scenario Outline: Display the name and price of an added product
  Given the shopping cart is empty
  When I add a product called "<name>" that costs <price> to the shopping cart
  Then the shopping cart should contain a product "<name>" that costs <price>

Examples:
  | name      | price |
  | Cucumber | 1.29€ |
  | Gherkin  | 0.89€ |
  | Lettuce  | 2.39€ |

Scenario: Calculate the total sum of all products in the shopping cart
  Given the shopping cart is empty
  When I add a product called "Cucumber" that costs 1.29€ to the shopping cart
  And I add a product called "Gherkin" that costs 0.89€ to the shopping cart
  Then the shopping cart should contain 2 items
  And the shopping carts total sum should be 2.18€

Scenario: Remove accidentally added products from the shopping cart
  Given the shopping cart is empty
  When I add a product called "Cucumber" that costs 1.29€ to the shopping cart
  And I add a product called "Gherkin" that costs 0.89€ to the shopping cart
  And I remove the product called "Cucumber" from the shopping cart
  Then the shopping cart should contain 1 item
  And the shopping cart should contain a product "Gherkin" that costs 0.89€
```

Listing 7: Example of the add products to a shopping cart feature

With the newly created feature in place, running Cucumber returns a considerable amount of output that essentially advises the creation of missing step definitions. Additionally, snippets covering the undefined steps are provided for implementation, illustrated in Listing 8.

6 Guided Procedure

```
catroweb@webbox:~$ cucumber
Feature: Add product to shopping cart

  As a customer
  I want to add selected products to my shopping cart
  So that they are collected and listed for a purchase

  Scenario: Add a product to my shopping cart
    Given ...

  Scenario Outline: Display the name and price of an added product
    Given ...

  Scenario: Calculate the total sum of all products in the shopping cart
    Given ...

  Scenario: Remove accidentally added products from the shopping cart
    Given ...

6 scenarios (6 undefined)
23 steps (23 undefined)
0m0.018s

You can implement step definitions for undefined steps with these snippets:

Given(/^the shopping cart is empty$/) do
  pending # express the regexp above with the code you wish you had
end

When(/^I add a product to the shopping cart$/) do
  pending # express the regexp above with the code you wish you had
end

Then(/^the shopping cart should contain (\d+) item$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end

When(/^I add a product called "(.*?)" that costs (\d+)\.(\d+)\u20ac to the shopping cart$/
/) do |arg1, arg2, arg3|
  pending # express the regexp above with the code you wish you had
end

Then(/^the shopping cart should contain a product "(.*?)" that costs (\d+)\.(\d+)\u20ac$/
/) do |arg1, arg2, arg3|
  pending # express the regexp above with the code you wish you had
end

Then(/^the shopping cart should contain (\d+) items$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then(/^the shopping carts total sum should be (\d+)\.(\d+)\u20ac$/) do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end

When(/^I remove the product called "(.*?)" from the shopping cart$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Listing 8: Cucumber output with missing step definitions

6.2 Create Step Definitions

When copying the proposed snippets to a file named *steps.rb* and running Cucumber, the output informs the user about pending step definitions, which are required to perform the tests. Step definitions are also referred to as glue code, as they are the link between plain text steps and the executable implementation. The produced output can be seen in Listing 9.

With regard to the implementation of the step definitions by the means of unit tests, the next step in the workflow cycle is reached, in particular the red part of the inner circle. However, Cucumber has no unit testing framework included and therefore a standalone tool that provides support for assertions is needed. Any tool that is able to perform unit tests is suitable, although it is recommended to use tools that are capable of performing SpecBDD. A small incomplete collection is prepared in Table 5.3. This example employs RSpec as a unit testing framework, because it was the first tool that made BDD available for Ruby, and Cucumber originated from it.

6.2.1 Transformations

As can be seen in Listing 7, some values are highlighted in the feature file. Furthermore, in the previously generated snippets of undefined steps, these values are replaced by regular expressions. All matches of these regular expressions are captured and are used as parameters for the step definition. Considering that these parameters are extracted from a text input, it occurs that a transformation or type cast is necessary for specific data types. It is suggested to extract and collect recurring transformations in a particular file. This example uses two transformations shown in Listing 10: the first captures simple numbers, and the second is used to capture Euro values.

6.2.2 Implementation

A complete example of the implemented step definitions using RSpec assertions can be seen in Listing 11. This resulting implementation has developed in very small steps, as is typical in an agile development process.

The presented example reveals interesting aspects. Firstly, it is sufficient to implement 7 step definitions to cover 17 steps, which were used in the acceptance

6 Guided Procedure

```
catroweb@webbox:~$ cucumber
Feature: Add product to shopping cart

  As a customer
  I want to add selected products to my shopping cart
  So that they are collected and listed for a purchase

Scenario: Add a product to my shopping cart
  Given the shopping cart is empty
    TODO (Cucumber::Pending)
    ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
    features/add_products.feature:8:in `Given the shopping cart is empty'
  When I add a product to the shopping cart
  Then the shopping cart should contain 1 item

Scenario Outline: Display the name and price of an added product
  Given the shopping cart is empty
  When I add a product called "<name>" that costs <price> to the shopping cart
  Then the shopping cart should contain a product "<name>" that costs <price>

Examples:
  | name      | price |
  | Cucumber | 1.29€ |
  TODO (Cucumber::Pending)
  ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
  features/add_products.feature:13:in `Given the shopping cart is empty'
  | Gherkin  | 0.89€ |
  TODO (Cucumber::Pending)
  ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
  features/add_products.feature:13:in `Given the shopping cart is empty'
  | Lettuce  | 2.39€ |
  TODO (Cucumber::Pending)
  ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
  features/add_products.feature:13:in `Given the shopping cart is empty'

Scenario: Calculate the total sum of all products in the shopping cart
  Given the shopping cart is empty
    TODO (Cucumber::Pending)
    ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
    features/add_products.feature:24:in `Given the shopping cart is empty'
  When I add a product called "Cucumber" that costs 1.29€ to the shopping cart
  And I add a product called "Gherkin" that costs 0.89€ to the shopping cart
  Then the shopping cart should contain 2 items
  And the shopping carts total sum should be 2.18€

Scenario: Remove accidentally added products from the shopping cart
  Given the shopping cart is empty
    TODO (Cucumber::Pending)
    ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
    features/add_products.feature:31:in `Given the shopping cart is empty'
  When I add a product called "Cucumber" that costs 1.29€ to the shopping cart
  And I add a product called "Gherkin" that costs 0.89€ to the shopping cart
  And I remove the product called "Cucumber" from the shopping cart
  Then the shopping cart should contain 1 item
  And the shopping cart should contain a product "Gherkin" that costs 0.89€

6 scenarios (6 pending)
23 steps (17 skipped, 6 pending)
0m0.010s
```

Listing 9: Cucumber output with pending step definitions

6 Guided Procedure

```
1 # shopping-cart/features/support/transforms.rb
2
3 CAPTURE_NUMBER = Transform /\d+$/ do |number|
4   number.to_i
5 end
6
7 CAPTURE_CASH_AMOUNT = Transform /(?:\d+\.\d+)?€$/ do |digits|
8   digits.to_f
9 end
```

Listing 10: Example of transformations used for step definitions

criteria of the feature. Secondly, the reuse and combination of implemented step definitions allows the creation of new scenarios with little or no additional effort. Finally, slightly different formulations or varying grammatical numbers of certain words with equivalent meaning can be aggregated in a single step definition. This is accomplished by the use of groups or optional letters in the regular expression matching the plain text steps. For example the Then step in Listing 11, which is responsible for checking the amount of items in the shopping cart, accepts the singular and plural form of the word “item”.

Considering the employment of assertions in step definitions, the Given and When steps are not obligated to include assertions, although it is recommended in order to find emerging regressions more easily. In contrast the Then steps are used to evaluate the produced outcome of the previously executed steps, and therefore an assertion is highly recommended.

6.2.3 Global State

The step definitions as they are implemented in Listing 11 do not work as expected. Considering that the `shopping_cart` variable is a local variable, its state is lost every time the scope of a step definition is left. In order to establish the expected behaviour, it is necessary to add the variable to a global state, which is called *world*. All resources in the world are persistent during the runtime of the scenario and are accessible from all step definitions. In Listing 12 it is shown how to make the `shopping_cart` variable available for all step definitions.

6 Guided Procedure

```
1 # shopping-cart/features/step_definitions/steps.rb
2
3 Given(/^the shopping cart is empty$/) do
4   shopping_cart.items.should eq(0), "Expected zero items " \
5     "but it had #{shopping_cart.items}"
6 end
7
8 When(/^I add a product to the shopping cart$/) do
9   product = Product.new("", 0.0)
10  shopping_cart.add(product)
11 end
12
13 When(/^I add a product called "(.*)" that costs #{
14   CAPTURE_CASH_AMOUNT}) to the shopping cart$/) do |name, price|
15   product = Product.new(name, price)
16   shopping_cart.add(product)
17 end
18
19 When(/^I remove the product called "(.*)" from the shopping cart$/) do |name|
20   shopping_cart.delete(name)
21 end
22
23 Then(/^the shopping cart should contain #{CAPTURE_NUMBER} items?$/) do |items|
24   shopping_cart.items.should eq(items), "Expected #{items} item " \
25     "but it had #{shopping_cart.items}"
26 end
27
28 Then(/^the shopping cart should contain a product "(.*)" that costs #{
29   CAPTURE_CASH_AMOUNT})$/) do |name, price|
30   products = shopping_cart.get_products
31   products[0].name.should eq(name), "Expected #{name} " \
32     "but it was #{products[0].name}"
33   products[0].price.should eq(price), "Expected #{price} " \
34     "but it was #{products[0].price}"
35 end
36
37 Then(/^the shopping carts total sum should be #{
38   CAPTURE_CASH_AMOUNT})$/) do |amount|
39   total = shopping_cart.total_sum
40   total.should eq(amount), "Expected #{amount} but it was #{total}"
41 end
```

Listing 11: Example of implemented step definitions

```
1 # shopping-cart/features/support/world_extensions.rb
2
3 module KnowsMyShoppingCart
4   def shopping_cart
5     @shopping_cart ||= ShoppingCart.new
6   end
7 end
8 World(KnowsMyShoppingCart)
```

Listing 12: Example of a world extension

6.3 Create Production Code

Since the implementation of the step definitions is complete, running Cucumber produces new output that is illustrated in Listing 13. The output informs the user about failing step definitions, which are caused by an uninitialized constant “KnowsMyShoppingCart::ShoppingCart”. That is a reasonable error message, as the required class is not implemented yet.

Therefore, the development of the production code can start following the inner circle of the BDD cycle. Iterating over the inner circle until all errors are resolved produces a working solution, which contains the expected behaviour. An example of the produced code is shown in Listing 14. However, the example is not complete yet, as the step definitions are not aware of the implementations location and thus are not able to access the implementation. A solution to fix this issue is to add support code pointing to the production code that is listed in Listing 15.

Now that there is everything set up, running Cucumber reports that all scenarios and steps were executed successfully, as shown in Listing 16. Therefore the inner cycle of the workflow cycle is complete and the green part of the outer circle is also complete. Considering that the acceptance criteria are met, the next step is to refactor the added feature. When examining the scenarios of the example feature, it becomes apparent that the Given step is used by all scenarios. Therefore the Given steps can be extracted to a Background step to improve the readability.

A subsequent verification of the refactored feature concludes the work on the first feature for the shopping cart. After that, either a new feature can be added or if the available behaviour satisfies all requirements, then the project can be finished.

6 Guided Procedure

```
catroweb@webbox:~$ cucumber
Feature: Add product to shopping cart

  As a customer
  I want to add selected products to my shopping cart
  So that they are collected and listed for a purchase

Scenario: Add a product to my shopping cart
  Given the shopping cart is empty
    uninitialized constant KnowsMyShoppingCart::ShoppingCart (NameError)
    ./features/support/world_extensions.rb:3:in `shopping_cart'
    ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
    features/add_products.feature:8:in `Given the shopping cart is empty'
  When ...

Scenario Outline: Display the name and price of an added product
  Given the shopping cart is empty
  When I add a product called "<name>" that costs <price> to the shopping cart
  Then the shopping cart should contain a product "<name>" that costs <price>

Examples:
  | name      | price |
  | Cucumber | 1.29€ |
  uninitialized constant KnowsMyShoppingCart::ShoppingCart (NameError)
  ./features/support/world_extensions.rb:3:in `shopping_cart'
  ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
  features/add_products.feature:13:in `Given the shopping cart is empty'
  | Gherkin  | 0.89€ |
  uninitialized constant KnowsMyShoppingCart::ShoppingCart (NameError)
  ./features/support/world_extensions.rb:3:in `shopping_cart'
  ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
  features/add_products.feature:13:in `Given the shopping cart is empty'
  | Lettuce  | 2.39€ |
  uninitialized constant KnowsMyShoppingCart::ShoppingCart (NameError)
  ./features/support/world_extensions.rb:3:in `shopping_cart'
  ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
  features/add_products.feature:13:in `Given the shopping cart is empty'

Scenario: Calculate the total sum of all products in the shopping cart
  Given the shopping cart is empty
    uninitialized constant KnowsMyShoppingCart::ShoppingCart (NameError)
    ./features/support/world_extensions.rb:3:in `shopping_cart'
    ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
    features/add_products.feature:24:in `Given the shopping cart is empty'
  When ...

Scenario: Remove accidentally added products from the shopping cart
  Given the shopping cart is empty
    uninitialized constant KnowsMyShoppingCart::ShoppingCart (NameError)
    ./features/support/world_extensions.rb:3:in `shopping_cart'
    ./features/step_definitions/steps.rb:3:in `/^the shopping cart is empty$/'
    features/add_products.feature:31:in `Given the shopping cart is empty'
  When ...

Failing Scenarios:
cucumber features/add_products.feature:7 # Scenario: Add a product to my shopping...
cucumber features/add_products.feature:12 # Scenario: Display the name and price...
cucumber features/add_products.feature:12 # Scenario: Display the name and price...
cucumber features/add_products.feature:12 # Scenario: Display the name and price...
cucumber features/add_products.feature:23 # Scenario: Calculate the total sum of...
cucumber features/add_products.feature:30 # Scenario: Remove accidentally added...

6 scenarios (6 failed)
23 steps (6 failed, 17 skipped)
0m0.013s
```


6 Guided Procedure

```
1 # shopping-cart/lib/shopping_cart.rb
2
3 class ShoppingCart
4   def initialize
5     @products = Array.new
6   end
7
8   def add(product)
9     @products.push product
10  end
11
12  def delete(name)
13    @products.each_with_index {
14      |product, index| @products.delete_at(index) if product.name === name
15    }
16  end
17
18  def items
19    @products.size
20  end
21
22  def total_sum
23    sum = 0
24    @products.each {|x| sum += x.price}
25    sum
26  end
27
28  def get_products
29    @products
30  end
31 end
32
33 class Product
34   def initialize(name, price)
35     @name = name
36     @price = price
37   end
38
39   def name
40     @name
41   end
42
43   def price
44     @price
45   end
46 end
```

Listing 14: Example of the shopping cart implementation

```
1 # shopping-cart/features/support/env.rb
2
3 require File.join(File.dirname(__FILE__), '..', '..', 'lib', 'shopping_cart')
```

Listing 15: Support code to make the implementation accessible

6 Guided Procedure

```
catroweb@webbox:~$ cucumber
Feature: Add product to shopping cart

  As a customer
  I want to add selected products to my shopping cart
  So that they are collected and listed for a purchase

  Scenario: Add a product to my shopping cart
    Given the shopping cart is empty
    When I add a product to the shopping cart
    Then the shopping cart should contain 1 item

  Scenario Outline: Display the name and price of an added product
    Given the shopping cart is empty
    When I add a product called "<name>" that costs <price> to the shopping cart
    Then the shopping cart should contain a product "<name>" that costs <price>

  Examples:
    | name      | price |
    | Cucumber | 1.29€ |
    | Gherkin  | 0.89€ |
    | Lettuce  | 2.39€ |

  Scenario: Calculate the total sum of all products in the shopping cart
    Given the shopping cart is empty
    When I add a product called "Cucumber" that costs 1.29€ to the shopping cart
    And I add a product called "Gherkin" that costs 0.89€ to the shopping cart
    Then the shopping cart should contain 2 items
    And the shopping carts total sum should be 2.18€

  Scenario: Remove accidentally added products from the shopping cart
    Given the shopping cart is empty
    When I add a product called "Cucumber" that costs 1.29€ to the shopping cart
    And I add a product called "Gherkin" that costs 0.89€ to the shopping cart
    And I remove the product called "Cucumber" from the shopping cart
    Then the shopping cart should contain 1 item
    And the shopping cart should contain a product "Gherkin" that costs 0.89€

6 scenarios (6 passed)
23 steps (23 passed)
0m0.009s
```

Listing 16: Cucumber successfully executes all scenarios

```
catroweb@webbox:~$ cucumber
No such file or directory - features. Please create a features directory to get
started. (Errno::ENOENT)
catroweb@webbox:~$ mkdir features
catroweb@webbox:~$ cucumber
0 scenarios
0 steps
0m0.000s
```

Listing 17: The output of Cucumber when executed in an empty folder

7 Behaviour Driven Functional Testing of Web Applications

“Automation in general is important for larger teams because it ensures that we have an impartial, objective measurement of when we’re finished,”
indicates Gojko Adzic [Adz11].

In large teams, different opinions are often encountered, which also applies to the judgement of whether a software feature needs improvements or is complete. Therefore automated acceptance tests, as they are provided by BDD, are well suited for introducing an objective measurement to unify the team understanding [Adz11].

Nevertheless, the identification of the right tool is challenging since no single best solution is known, which is capable of satisfying all possible requirements. It mostly depends on existing preconditions and objectives as well as the current state of the project. For established projects, the preferable practice is to make use of existing code and tools to reduce adaptation efforts, which unfortunately decreases the number of potential options.

However, as shown in the previous chapter, tools supporting BDD use step definitions in order to map plain text instructions to an executable implementation. This design provides great flexibility and simplifies the employment of available functional test tools for web applications, which were discussed in Section 4.2.

7.1 Introduce Behaviour Driven Development

This section introduces BDD to an established project, namely the Catrobat project’s community website¹. As discussed earlier, the project heavily relies on

¹<https://github.com/Catrobat/Catroweb>

agile development methodologies. Therefore, a test infrastructure with existing functional tests is available, which serves as a basis and is adapted in the next steps.

7.1.1 Preconditions

The functional tests make use of the Selenium Grid framework as described in Section 4.2.1 and are implemented in the Java programming language. A base class contains the most frequently used functionalities such as recurring assertions or the creation of test sessions for a variety of browsers and screen dimensions. In addition, helper functions are included, such as capture the current screen or an explicit wait for Ajax requests. Besides, scripts are provided to automate essential tasks required to set up the infrastructure for running the web application under test, which consists of a web server and a database server.

Thus, a BDD tool is suggested that is well suited for a Java development environment and is able to execute features written in Gherkin syntax, as an additional requirement, which was mentioned in the introduction. With these constraints, consulting the previously composed table of BDD tools (see Table 5.1) yields only one option that satisfies all requirements. That is *Cucumber-JVM* a Java implementation of Cucumber, as it is the only Java based tool that can interpret Gherkin syntax.

7.1.2 Cucumber-JVM Meets Selenium Grid

Preparations

Maven² and Ant³ are two well-known project management tools that automate the build process and perform recurring tasks in Java projects. Both tools work well with Cucumber-JVM and Selenium Grid. Therefore, the choice of which one of these tools is mostly a matter of personal preference. This small project is managed by Ant to perform tasks, such as downloading required libraries, launching a Selenium Grid hub, launching several WebDriver nodes, compiling

²<https://maven.apache.org/>

³<https://ant.apache.org/>

7 Behaviour Driven Functional Testing of Web Applications

and executing functional tests, and cleaning the test environment by removing artefacts, compiled Java classes and generated test reports.

The source code of the implementation is available at GitHub⁴ including the necessary `build.xml` file, in order to accomplish these Ant tasks. It contains existing functionalities of the original project to control Selenium Grid components and extends them to comfortably prepare a test environment with a large range of nodes. Following the *practical example* described on GitHub, results in a test environment consisting of a hub, a Firefox node, a Chrome node, a Safari node, an Internet Explorer node, and an Android node. An illustration of the status page of the hub with connected nodes can be seen in Figure 7.1.

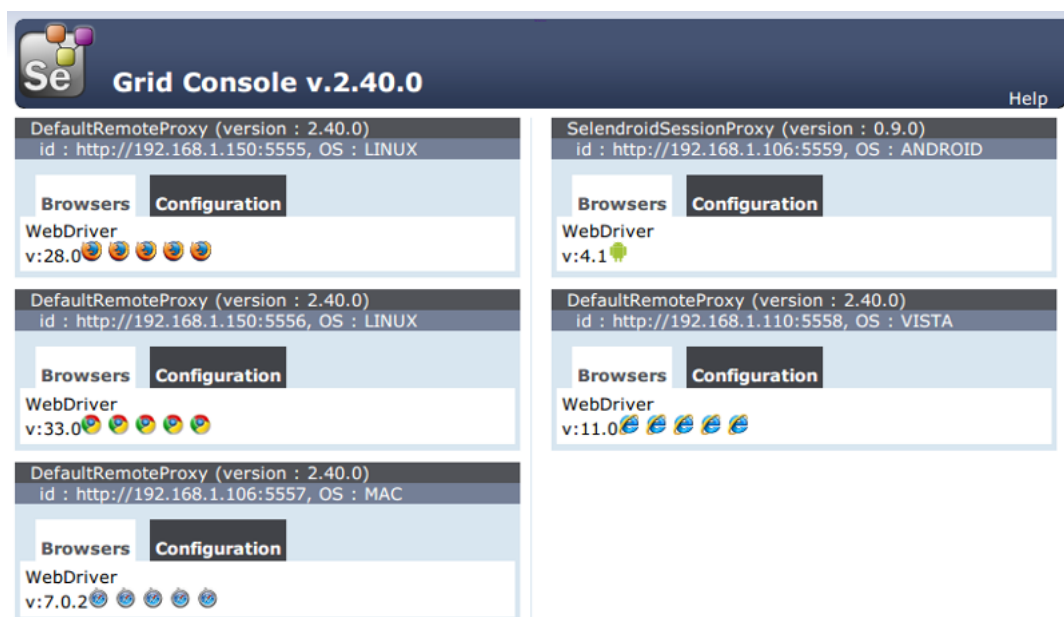


Figure 7.1: Selenium Grid console with connected nodes

Additionally the headless browser implementation PhantomJS⁵ can be connected to the hub and used to perform tests. It is based on the WebKit browser engine and it supports various web standards. The main advantage is that it significantly improves the test runtime, since system resources and the overhead caused by the browser startup are saved.

⁴<https://github.com/chrisss404/cucumber-selenium-grid>

⁵<http://phantomjs.org/>

Encountered Difficulties

These preparations are sufficient to perform Selenium tests in a distributed test setting. The introduction of Cucumber follows the same process as described in the previous chapter and incorporates minor adaptations to meet Java characteristics. One of the most significant differences is the absence of the global state, also referred to as “world”, that makes shared data accessible to all step definitions. In order to deal with this situation Aslak Helleøy offers two approaches [Hel], either to put all step definitions into a single class or to make use of a dependency injection module⁶ to connect step definitions from multiple classes. This implementation uses a single class to store step definitions, since it has a limited scope and is not intended for productive use.

The main reason for employing highly scalable Selenium Grid tests is the desire for a reduction of the test execution time, which becomes more apparent with a growing test suite. With regard to obtaining an improved runtime it is necessary to execute tests in parallel, however there are limitations. In contrast to the existing implementation, which was able to run tests alongside on a method level and to define a number of worker threads, this has to be addressed manually with this implementation. For instance, this behaviour can be achieved by creating groups using tags and by running each group in its own Cucumber instance. On the one hand, this approach allows more control over which methods run parallel to each other on the other hand it can be cumbersome to create balanced groups. Additionally, each instance generates its own test report, which can be confusing when searching for a particular test result. The described implementation makes use of two groups, which are divided into slow and fast tests.

Implementation Details

In order to identify the differences between traditional and Cucumber driven Selenium tests, four existing tests were selected and transformed to Gherkin features. The following functional web tests were selected from the community website’s test suite.

⁶http://cukes.info/install-cucumber-jvm.html#dependency_injection_modules_for_cucumberjava

7 Behaviour Driven Functional Testing of Web Applications

- The first scenario checks the behaviour of the website's search functionality by uploading a new project with unique title and searching for it, as can be seen in Listing 18.
- The second scenario navigates to the start page and verifies a certain text, in this case the title of the newest projects section, shown in Listing 19.
- The third scenario that is available in Listing 19 addresses the language switch functionality of the website.
- The fourth scenario displayed in Listing 20 concerns the behaviour of the project download counter.

```
1 // catroid/SearchTests.java
2 @Test(dataProvider = "randomProjects", groups = { "functionality", "upload" },
3       description = "search for random title and description")
4 public void titleAndDescription(HashMap<String, String> dataset)
5     throws Throwable {
6     try {
7         projectUploader.upload(dataset);
8
9         String projectTitle = dataset.get("projectTitle");
10        String projectDescription = dataset.get("projectDescription");
11
12        openLocation("search/?q=" + projectTitle + "&p=1");
13        ajaxWait();
14
15        assertTrue(isTextPresent (
16            CommonStrings.SEARCH_PROJECTS_PAGE_TITLE.toUpperCase()));
17        assertTrue(isElementPresent(By.xpath("//a[@title=\"" + projectTitle + "\"]")));
18
19        // test description
20        driver().findElement(By.xpath("//*[@id='largeMenu']/div[4]/input")).clear();
21        driver().findElement(By.xpath("//*[@id='largeMenu']/div[4]/input"))
22            .sendKeys(projectDescription);
23        driver().findElement(By.id("largeSearchButton")).click();
24        ajaxWait();
25        assertTrue(isTextPresent (
26            CommonStrings.SEARCH_PROJECTS_PAGE_TITLE.toUpperCase()));
27        assertTrue(isElementPresent(By.xpath("//a[@title=\"" + projectTitle + "\"]")));
28    } catch (AssertionError e) {
29        captureScreen("SearchTests.titleAndDescription." + dataset.get("projectTitle"));
30        log(dataset.get("projectTitle"));
31        log(dataset.get("projectDescription"));
32        throw e;
33    } catch (Exception e) {
34        captureScreen("SearchTests.titleAndDescription." + dataset.get("projectTitle"));
35        throw e;
36    }
37 }
```

Listing 18: Selenium test that checks the website's search functionality

7 Behaviour Driven Functional Testing of Web Applications

```
1 // catroid/IndexTests.java
2 @Test(groups = { "visibility", "popupwindows" },
3 description = "click download,header,details -links ")
4 public void index() throws Throwable {
5     try {
6         openLocation();
7         ajaxWait();
8         // test page title and header title
9         assertTrue(driver().getTitle().matches("^Pocket Code Website.*"));
10        assertTrue(isTextPresent (
11            CommonStrings.NEWEST_PROJECTS_PAGE_TITLE.toUpperCase()));
12
13        clickLastVisibleProject();
14        ajaxWait();
15        assertRegExp(".*\/details\/[0-9]+", driver().getCurrentUrl());
16        driver().navigate().back();
17        ajaxWait();
18        assertTrue(isTextPresent (
19            CommonStrings.NEWEST_PROJECTS_PAGE_TITLE.toUpperCase()));
20
21        // test home link
22        driver().findElement(By.xpath("//*[@id='largeMenu']/div[2]/a")).click();
23    } catch (AssertionError e) {
24        captureScreen("IndexTests.index");
25        throw e;
26    } catch (Exception e) {
27        captureScreen("IndexTests.index");
28        throw e;
29    }
30 }
31
32 @Test(groups = { "functionality", "upload" },
33 description = "language select tests")
34 public void languageSelect() throws Throwable {
35     try {
36         openLocation("termsOfUse");
37         assertTrue(isTextPresent("Terms of Use".toUpperCase()));
38         assertTrue(isElementPresent(By.xpath("//html[@lang=' " +
39             Config.SITE_DEFAULT_LANGUAGE + "' ]")));
40         openLocation("termsOfUse", false);
41         assertTrue(isElementPresent(By.id("switchLanguage")));
42         (new Select(driver().findElement(By.id("switchLanguage")))).selectByValue("de");
43         ajaxWait();
44         assertTrue(isTextPresent("Nutzungsbedingungen".toUpperCase()));
45         assertTrue(isElementPresent(By.id("switchLanguage")));
46         assertTrue(isElementPresent(By.xpath("//html[@lang=' de' ]")));
47         (new Select(driver().findElement(By.id("switchLanguage")))).selectByValue("en");
48         ajaxWait();
49         assertTrue(isTextPresent("Terms of Use".toUpperCase()));
50         assertTrue(isElementPresent(By.xpath("//html[@lang=' en' ]")));
51     } catch (AssertionError e) {
52         captureScreen("IndexTests.languageSelect");
53         throw e;
54     } catch (Exception e) {
55         captureScreen("IndexTests.languageSelect");
56         throw e;
57     }
58 }
```

Listing 19: Selenium tests that verify a certain text on the start page and address the website's language switch functionality

7 Behaviour Driven Functional Testing of Web Applications

These tests were adopted without modifications, except the *detailsPageCounter-Link* test from Listing 20 was reduced to the essentials. An example of these functionalities converted to Gherkin syntax can be seen in Listing 21. It shows that the newly written scenarios are much shorter and easier to comprehend compared to the former version. When running Cucumber with this feature file in place, unimplemented step definitions are reported, as discussed in the previous chapter. The implementation of these step definitions makes use of Selenium to control the web application for performing the indicated behaviour.

```
1 // catroid/DetailsTests.java
2 @Test(dataProvider = "detailsProject", groups = { "functionality", "upload" },
3 description = "view + download counter test")
4 public void detailsPageCounterLink(HashMap<String, String> dataset)
5 throws Throwable {
6     try {
7         String response = projectUploader.upload(dataset);
8         String id = CommonFunctions.getValueFromJSONObject(response, "projectId");
9         String title = dataset.get("projectTitle");
10        int numOfDownloads = -1;
11        int numOfDownloadsAfter = -1;
12
13        By downloadsElement = By.xpath(
14            "//*[@id='projectDetailsContainer']/div[6]/ul/li[4]/div[2]/span");
15        By downloadsButton = By.xpath(
16            "//*[@id='projectDetailsContainer']/div[3]/div/a[1]/div/span");
17
18        openLocation("details/" + id);
19        ajaxWait();
20        assertTrue(containsElementText(By.id("projectDetailsProjectTitle"),
21            title.toUpperCase()));
22
23        numOfDownloads = Integer.parseInt(
24            driver().findElement(downloadsElement).getText().split(" ")[0]);
25        driver().findElement(downloadsButton).click();
26
27        driver().navigate().refresh();
28        ajaxWait();
29        numOfDownloadsAfter = Integer.parseInt(
30            driver().findElement(downloadsElement).getText().split(" ")[0]);
31        assertEquals(numOfDownloads + 1, numOfDownloadsAfter);
32    } catch (AssertionError e) {
33        captureScreen("DetailsTests.detailsPageCounterLink." +
34            dataset.get("projectTitle"));
35        throw e;
36    } catch (Exception e) {
37        captureScreen("DetailsTests.detailsPageCounterLink." +
38            dataset.get("projectTitle"));
39        throw e;
40    }
41 }
```

Listing 20: Selenium test that investigates the download count functionality

7 Behaviour Driven Functional Testing of Web Applications

Moreover, the step definitions are the right place to add assertions for evaluating the application's behaviour. However, the Cucumber-JVM implementation does not include assertions. Therefore, JUnit assertions are employed for this task in this example.

```
Feature: Provide key functionalities to Catrobat's community website
@fast
Scenario: Open the startpage and check for a certain text
  Given I am on the startpage
  When I change the language to "English"
  Then the title of the featured section should be "FEATURED"

@fast
Scenario Outline: Use the language switch to change the website's language
  Given I am on the startpage
  When I change the language to "<language>"
  Then the title of the newest section should be "<title>"

  Examples:
  | language | title |
  | Deutsch | NEUESTE |
  | English | NEWEST |

@slow
Scenario: Use the search function to find a project
  Given I use Android browser
  And I am on the startpage
  And the website's language is "English"
  When I use the top search box to search for a project called "Tic-Tac-Toe"
  Then I should see "Search Results"
  And the number of search results should be 1

@fast
Scenario: Increase the download count when a project is downloaded
  Given I am on the details page of the project 1478
  And the website's language is "English"
  When I press the download button
  Then the download count should be increased by one
```

Listing 21: The aforementioned Selenium tests converted to Cucumber scenarios

The implemented step definitions corresponding to the aforementioned scenarios are available at GitHub⁷. In the same way as in the original JUnit framework, methods can be annotated with tags to assign particular tasks, for example to run a method before or after a test is executed. This example uses a method tagged with *@Before* to retrieve environment variables and to make Cucumber's scenario object available to all step definitions. The scenario object is respon-

⁷<https://github.com/chrisss404/cucumber-selenium-grid/blob/master/src/at/tugraz/ist/cucumber/SeleniumStepdefs.java>

7 Behaviour Driven Functional Testing of Web Applications

sible for the test report contents, and it is necessary to access it to add messages or images to the report.

Generally, the method tagged with *@Before* would be preferred to initialise the Selenium WebDriver driver object, which is required for performing browser interactions. Although a different approach is employed to provide more flexibility that allows to run each scenario with an individual driver object. It is inspired by the well-known singleton pattern that creates a new instance of an object on the first access. Therefore a method is prepared to access the driver object, which is responsible for the initialisation by either using the default capabilities or the capabilities given by the step definition. This enables step definitions to create driver objects with varying capabilities including the definition of a certain browser for approaching test.

In contrast to the *@Before* method the method tagged with *@After* satisfies the expectations, as it is responsible for closing and cleaning previously initialised driver objects. Additionally, it can determine if the test was successful or whether an error was encountered. In the event of an error it is useful to record the current state of the website, therefore a screenshot is taken and added to the test report.

Considering the distributed test environment it is important that the clients forward the captured screen to the test runner, thus it can be integrated in the test report. This can be done by creating a base64 encoded string or a byte array representation of the image that is suitable for network transmissions. In Figure 7.2 a screenshot is shown that was taken after the last step of the search functionality scenario of Listing 21.

To deal with timing issues caused by asynchronous JavaScript requests, the `JavascriptExecutor` of the WebDriver object is used to examine the current state in a similar way as it is shown in Listing 4. Certainly, this solution is only applicable when jQuery is employed to perform Ajax requests, however that is the case for this project.

Additionally, this implementation applies implicit waits in order to approach general timing issues caused by less powerful test systems, like Android devices. Thus spurious errors are strongly reduced, as failures are only reported after the defined timeout has expired.

Further Enhancements

For the development process, it is convenient to assign a unique tag to the scenario under development to reduce the test response time. Additionally, this leads to a more compact test report, which improves the identification of occurring errors. However a drawback is that newly introduced errors in the remaining scenarios are not recognised immediately.

As previously mentioned it is necessary to create test groups in order to execute the scenarios in parallel. The feature shown in Listing 21 divides the scenarios into a *slow* and a *fast* group with similar runtimes. Thus, if the groups are well balanced, the overall runtime is reduced by half. Furthermore, a maximum runtime can be specified and whenever this limit is exceeded, a new group, which is run by a new instance, can be introduced. Given that the used Grid hub has sufficient resources, a fixed runtime is achievable.

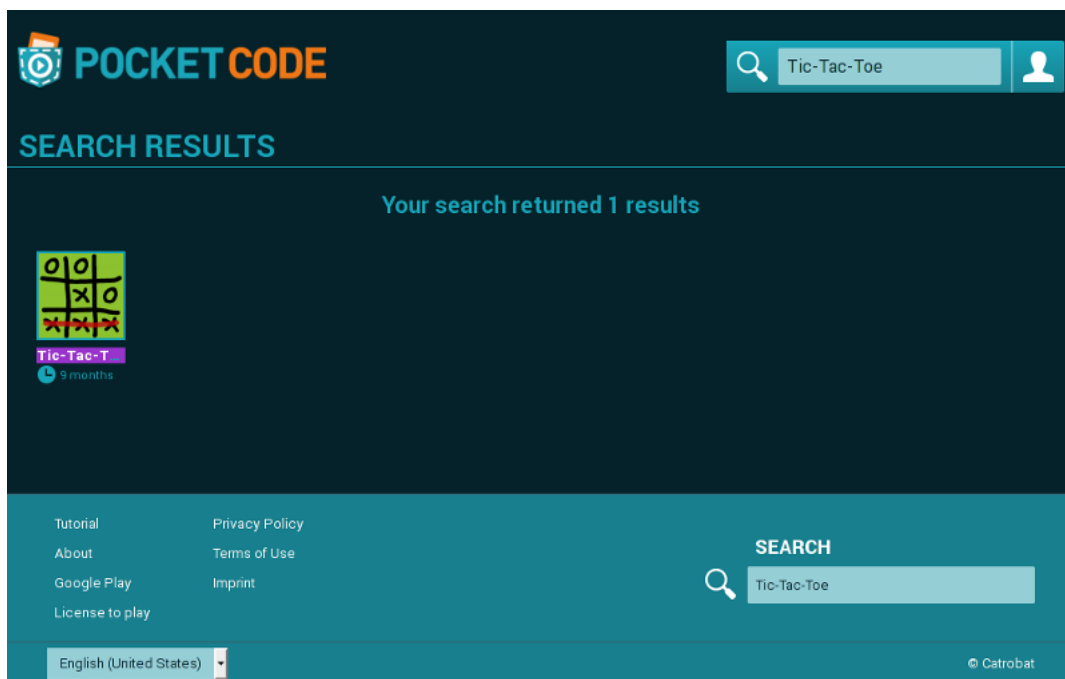


Figure 7.2: Selenium screenshot taken after the last step of the search functionality scenario

7.2 Findings

7.2.1 Comparison

This section compares the existing implementation, which follows a TDD approach, with the introduced implementation, which follows the discussed BDD approach. Both implementations make use of Selenium for interactions with the application under test to guarantee equal premises and to ensure a meaningful comparison between the development strategies. The subsequent aspects are examined for the identification of advantages and disadvantages performing functional web application tests, and are summarised in Table 7.1 concerning BDD and Table 7.2 regarding TDD.

Test Creation

The creation of new scenarios is easier in BDD, because of the ability to write scenarios in natural language following Gherkin syntax, and knowledge about the employed programming language is not required. Therefore team members working on design or usability tasks are able to create new scenarios independently, which is barely achievable when adhering to a TDD approach. However, the convenience in the test creation comes with a disadvantage that is the additional effort to make the Gherkin features executable. As previously discussed, it is indispensable to implement the corresponding step definitions, which is an additional step compared to TDD.

Complexity

The BDD test infrastructure introduces an additional layer due to Cucumber, which leads to an increase of the overall complexity. Consequently, it is important to master the supplementary tool in order to prevent potential impacts and issues. A typical issue due to the added complexity is that multiple scenarios start to misbehave after the implementation of a step definition has been modified. Another issue arises when scenarios are designed to depend on a state that is produced by a previous scenario, which is prone to failure and should be avoided. On the other hand this additional layer increases the readability and comprehensibility of scenarios compared to the test cases in TDD.

Execution Speed

In regard to test execution speed no significant difference emerged. This is not surprising, as the impact of Cucumber is restricted to the test runner and the performance bottleneck of Selenium tests is caused by WebDriver nodes. The reason for the high resource consumption of WebDriver nodes is based on the fact that each test is run in its own browser instance. Therefore, executing multiple instances of web browsers can be very demanding for average systems, in particular the memory usage can become an issue.

Comprehensibility

It is simple to identify available functionalities in BDD, as the scenarios are written in plain text and make use of ubiquitous language. Additionally it is not necessary to read and understand the source code of the test cases due to outdated documentation, because the Cucumber features are an accurate representation of the code. Therefore, in regard to comprehensibility Cucumber driven tests yield an improvement over the previous method employing TDD. Specifically large projects consisting of several teams profit highly from the reduced effort to understand the work of fellow teams.

Maintainability

The behaviour driven development process enforces to write modular code, which improves the maintainability and reusability, as each step definition is implemented as its own method. However, that is not a unique characteristic, since modular code can be created by other means as well. Nevertheless, additional value is added to the development process by providing a convenient way to combine methods in terms of writing Cucumber features. Considering, that reusability is a main concern, it is crucial that the method accurately follows its description, otherwise unexpected behaviour can be provoked.

For example the feature shown in Listing 21 involves 16 total step definitions versus 12 unique step definitions resulting in a reuse rate of 25% in this small demonstration. Another benefit is the flexibility to combine and rearrange step definitions, for example to run all scenarios of a feature in a specific browser, it is sufficient to add a single Given step to the Background section, which prepares the desired web browser capabilities.

7 Behaviour Driven Functional Testing of Web Applications

Advantages	Disadvantages
<i>Test creation</i> requires only knowledge about Gherkin syntax	<i>Scenario execution</i> is only achievable after implementing the corresponding step definitions
<i>Test comprehensibility</i> is improved, since scenarios are written in natural language	<i>Test complexity</i> is higher, because of the additional layer introduced by Cucumber
<i>Test maintainability</i> is enhanced, as a result of enforcing modular code	<i>Test failures</i> can be caused by an additional error source
<i>Project documentation</i> is always up to date	

Table 7.1: Advantages and disadvantages of BDD performing functional web tests

Advantages	Disadvantages
<i>Scenario execution</i> is achievable without additional effort	<i>Test creation</i> requires programming skills
<i>Test complexity</i> is not increased and depends on the used functional web testing framework	<i>Test comprehensibility</i> is worse, since scenarios are written in a programming language
<i>Test failures</i> are reduced, as a result of the decreased error source	<i>Test maintainability</i> depends on the carried out precautions
<i>Project documentation</i> is not automatically updated	

Table 7.2: Advantages and disadvantages of TDD performing functional web tests

7.2.2 Disadvantages

The freedom of defining scenarios can result in duplication, since it is easy to unintentionally use a different wording to describe the same distinct behaviour. Therefore, it is recommended to employ a modern IDE that offers an autocomplete feature, which in turn weakens the advantage of the simplified test creation. Besides, there is an increased effort to prepare the development environment, since there is an additional tool to maintain, integrate, and master. Furthermore, the necessity to apply regular expressions to determine corresponding step definitions reduces the execution speed in certain circumstances.

Finally, creating a test setting that is able to run multiple test instances is very cumbersome. On the one hand multiple web server instances are required to avoid interferences, which however is necessary for both development strategies. On the other hand it was not possible to create a single test report containing all results, which can be confusing when aiming for a specific test result.

7.3 Related Tools

There are numerous possibilities for obtaining a similar implementation, as it is demonstrated in Section 7.1.2. Basically, all functional web testing frameworks mentioned in Section 4.2 can be combined with any tool suitable for StoryBDD from Table 5.1. That is attributed to the flexible architecture, which is enforced by the behaviour driven development process, since there are no restrictions when implementing step definitions. However, both have to use the same programming language, except for Cucumber that offers a wire protocol, which supports several languages for this use case, as shown in Table 5.2. That aside, there are two popular frameworks providing the described functionality: Mink, and Capybara.

7.3.1 Mink

Mink⁸ is an extension for Behat, one of the tools listed in Table 5.1 that support StoryBDD, to run behaviour driven functional web application tests in a PHP environment. According to the online documentation [Lab] there are five

⁸<https://github.com/Behat/en-mink.behat.org/blob/master/index.rst>

drivers included which are capable of controlling web browsers in order to run functional tests. Among them are two headless drivers, which emulate a web browser to obtain fast test response times: GoutteDriver, and ZombieDriver. As well as three drivers, which are able to control real web browsers to perform tests: SeleniumDriver, WebDriver, and SahiDriver.

Since test execution time is a decisive factor of the development speed, the GoutteDriver is selected by default, as it is the fastest of the supported drivers. Unfortunately, the speed advantage is achieved by not supporting JavaScript. However this can be resolved by defining an alternative driver with JavaScript support and tagging scenarios requiring JavaScript with the `@javascript` tag. Therefore, all tagged scenarios are carried out by the alternative driver and the remaining scenarios make use of the default driver. Thus, two important aspects of automated tests are covered: fast test execution speed and lots of available functionality.

Additionally, the previously described Selenium Grid test environment can be employed by applying the WebDriver driver. Finally, a great advantage is that it includes a range of predefined step definitions of common actions ready for use. This allows to perform general test scenarios with little or no coding effort.

7.3.2 Capybara

As explained in the Cucumber book [WH12], Capybara⁹ is a framework used to perform behaviour driven functional web application tests in Ruby. It is designed for the employment in Cucumber tests as they are shown in Chapter 6. Since it is an improved version of Webrat and Webrat was strongly inspired by Watir, these tools have much in common and it suffices to discuss Capybara's properties.

Similarly to Mink, Capybara wraps several browser drivers including Selenium WebDriver behind a common interface. It includes a headless driver called RackTest that is comparable to Mink's GoutteDriver, which however only works for Ruby/Rack applications. Another similarity with Mink is that scenarios requiring JavaScript can be tagged with `@javascript` in order to run them with an appropriate driver. Otherwise the functionality is comparable to Mink's capabilities.

⁹<https://jnicklas.github.io/capybara/>

8 Conclusion

It has been shown that a behaviour driven development strategy is capable of performing functional web tests and that it adds additional value to projects, which already employ agile software development methodologies. However, the development process becomes more complex due to an additional layer. The migration process from TDD to BDD encountered no serious issues and the re-use of numerous aspects of the original implementation was feasible. Therefore it is suggested that potential migration efforts should not be a decisive aspect for judging about the employment of BDD.

One of the most noticeable advantages is the attainment of documentation that is responsive, grows with the system, and provides an accurate representation of the system's functionality. That is particularly an advantage, if a project consists of several sub-teams or involves non-technical team members. Additionally, the barrier to share knowledge about the system is greatly reduced. Moreover, requirements written in natural language promote the development of a ubiquitous language, which leads to shared vocabulary and improved team communication.

Another essential point of BDD is that developers are guided through the development process encouraging best practices, as described in Chapter 6. Therefore this approach appears to be more natural for employing test-first development, compared to former agile development processes, such as Extreme Programming (XP), or Test Driven Development (TDD). Furthermore, the promotion of a modular code structure improves the maintainability and reusability of the code, which yields a long-term benefit.

In regard to web applications, the most noticeable observation concerns the growing popularity of mobile devices. This increase of mobile web usage adds new constraints with a strong influence on the development process of web applications. The limited screen real estate of mobile devices is the most considerable aspect that needs to be addressed. Fortunately, techniques, like responsive web design and asynchronous requests, can improve the situation in favour of the user and are frequently used in modern web applications. The

8 Conclusion

demonstrated functional test environment employing Selenium WebDriver is capable of carrying out tests on web applications using these technologies.

In the beginning of functional web test frameworks, the automated control of different web browsers was inconsistent and unstable, as there was no common interface for that purpose available. However, the recently elaborated specification for a WebDriver API has greatly improved the situation, since the browser vendors are responsible for implementing an interface to control their web browser. Although it is still a working draft, the interface is implemented by the most widely used browsers and is already employed by functional web testing frameworks mentioned in Section 4.2. Therefore the ability to perform functional tests on sophisticated web applications has greatly improved.

The discussed solutions, which combine both approaches, are comparable in functionality, since they all offer access to the discussed WebDriver API for browser interactions. Therefore, the procedure to create a functional test is very similar with a considerable difference in regard to the employed programming language and development environment. Among them are the two frameworks Mink (employing PHP) and Capybara (engaging Ruby), as well as the example presented in Chapter 7 written in Java. The presented example can be used as a starting point, however, for productive use it needs improvements in regard to code organisation.

In conclusion, an agile development approach is well-suited for web applications, and the required functional test tools have strongly improved. However, the best fitting tool of the three aforementioned test frameworks is hard to determine, since it mostly depends on the project's constraints. It is suggested to experiment with the available frameworks in order to determine the best fitting tool. Furthermore, if the project already makes use of an agile development approach, it is highly recommended to examine BDD's capabilities and to identify its potential benefits.

Appendix

Bibliography

- [Adz11] Gojko Adzic. *Specification by Example*. Manning Publications, June 2011. ISBN: 9781617290084.
- [AGL10] Sarah Allen, Vidal Graupera and Lee Lundrigan. *Pro Smartphone Cross-Platform Development*. Apress, 2010. ISBN: 9781430228684.
- [AM07] Abel Avram and Floyd Marinescu. *Domain-Driven Design Quickly*. Lulu.com, Dec. 2007. ISBN: 9781411609259. URL: <http://www.infoq.com/minibooks/domain-driven-design-quickly>.
- [Amb] Scott Ambler. *Introduction to Test Driven Development (TDD)*. URL: <http://www.agiledata.org/essays/tdd.html> (visited on 8th Jan. 2014).
- [Amb11] Scott W. Ambler. *IT Project Success Rates Survey*. Nov. 2011. URL: <http://www.ambysoft.com/surveys/success2011.html#Results> (visited on 10th Dec. 2013).
- [Bec03] Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, Nov. 2003. ISBN: 9780321146533.
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, Oct. 1999. ISBN: 9780201616415.
- [Bee+01a] Mike Beedle et al. *The Agile Manifesto*. Feb. 2001. URL: <http://agilemanifesto.org/> (visited on 10th Dec. 2013).
- [Bee+01b] Mike Beedle et al. *The Agile Manifesto Principles*. Feb. 2001. URL: <http://agilemanifesto.org/principles.html> (visited on 10th Dec. 2013).
- [Ber] Sebastian Bergmann. *PHPUnit Manual*. URL: <http://phpunit.de/manual/current/en/phpunit-book.pdf> (visited on 16th Jan. 2014).
- [BGH07] C. Titus Brown, Grig Gheorghiu and Jason R. Huggins. *An Introduction to Testing Web Applications with twill and Selenium*. O'Reilly Media, June 2007. ISBN: 9780596527808.

Bibliography

- [Bjö09a] Tomas Björkholm. *What is Best, Scrum or Kanban?* June 2009. URL: <http://www.agileconnection.com/article/what-best-scrum-or-kanban?page=0,1> (visited on 19th Dec. 2013).
- [Bjö09b] Tomas Björkholm. *What is Best, Scrum or Kanban?* June 2009. URL: <http://www.agileconnection.com/article/what-best-scrum-or-kanban?page=0,2> (visited on 19th Dec. 2013).
- [Bur12] David Burns. *Selenium 2 Testing Tools Beginner's Guide*. Packt Publishing, Oct. 2012. ISBN: 9781849518307.
- [CG08] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, Dec. 2008. ISBN: 9780321534460.
- [Che+10] David Chelimsky et al. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends*. Pragmatic Bookshelf, Dec. 2010. ISBN: 9781934356371.
- [CL11] Andre Charland and Brian Leroux. 'Mobile Application Development: Web vs. Native'. In: *Communications of the ACM* 54.5 (2011), pp. 49–53.
- [Dav09] Barbee Davis. *97 Things Every Project Manager Should Know: Collective Wisdom from the Experts*. O'Reilly Media, Aug. 2009. ISBN: 9780596804169.
- [DD11] Josh Dehlinger and Jeremy Dixon. 'Mobile Application Software Engineering: Challenges and Research Directions'. In: *Workshop on Mobile Software Engineering* (2011).
- [Fow] Martin Fowler. *Xunit*. URL: <http://www.martinfowler.com/bliki/Xunit.html> (visited on 10th Jan. 2014).
- [Fow06] Martin Fowler. *Continuous Integration*. May 2006. URL: <http://www.martinfowler.com/articles/continuousIntegration.html> (visited on 9th Jan. 2014).
- [Fra12] Ben Frain. *Responsive Web Design with HTML5 and CSS3*. Community experience distilled. Packt Publishing, 2012. ISBN: 9781849693196.
- [HBW] Aslak Helleøy, Julien Biezemans and Matt Wynne. *Cucumber Pro Documentation*. URL: <https://cucumber.pro/documentation.html> (visited on 5th Mar. 2014).

Bibliography

- [Hel] Aslak Hellesøy. *Cucumber-JVM State of the "World"*. URL: <http://permalink.gmane.org/gmane.comp.programming.tools.cucumber/14010> (visited on 8th Apr. 2014).
- [Hiro8] Kenji Hiranabe. *Kanban Applied to Software Development: from Agile to Lean*. Jan. 2008. URL: <http://www.infoq.com/articles/hiranabe-lean-agile-kanban> (visited on 19th Dec. 2013).
- [Hun+10] Dave Hunt et al. *Selenium Documentation*. Feb. 2010. URL: http://oss.infoscience.co.jp/seleniumhq/docs/book/Selenium_Documentation.pdf (visited on 31st Jan. 2014).
- [Hus+08] Zahid Hussain et al. 'Optimizing Extreme Programming'. In: *Computer and Communication Engineering, ICCCE*. IEEE. May 2008, pp. 1052–1056.
- [Jr95] Frederick P. Brooks Jr. *The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering*. Pearson Education, 1995.
- [Lab] KNP Labs. *Web Acceptance Testing - Mink Documentation*. URL: <http://mink.behat.org/> (visited on 16th Apr. 2014).
- [Mac] Beate Macura. *Tablet, Smartphone, Datenbrille*. URL: <http://orf.at/stories/2206899/2206900/> (visited on 8th Jan. 2014).
- [Mar10] Ethan Marcotte. *Responsive Web Design*. May 2010. URL: <http://alistapart.com/article/responsive-web-design> (visited on 28th Oct. 2013).
- [MDG11] Peter MacIntyre, Brian Danchilla and Mladen Gogala. *Pro PHP Programming*. Apress, 2011. ISBN: 9781430235606.
- [Mik13] Kasia Mikoluk. *Agile Vs. Waterfall: Evaluating The Pros And Cons*. Sept. 2013. URL: <https://www.udemy.com/blog/agile-vs-waterfall/> (visited on 10th Dec. 2013).
- [MW12] Mary Meeker and Liang Wu. *Internet Trends*. Dec. 2012. URL: <http://www.kpcb.com/insights/2012-internet-trends-update> (visited on 18th Nov. 2013).
- [MW13] Mary Meeker and Liang Wu. *Internet Trends*. May 2013. URL: <http://www.kpcb.com/insights/2013-internet-trends> (visited on 18th Nov. 2013).
- [Nor] Dan North. *What's in a Story*. URL: <http://dannorth.net/whats-in-a-story/> (visited on 27th Feb. 2014).

Bibliography

- [Nor06] Dan North. 'Introducing BDD'. In: *Better Software* (Mar. 2006). URL: <http://dannorth.net/introducing-bdd/>.
- [Osh09] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications, July 2009. ISBN: 9781933988276.
- [Raj] Arun Raj. *Difference Between Functional Testing and Integration Testing*. URL: <http://arunrajvdm.blogspot.co.at/2013/10/difference-between-functional-testing.html> (visited on 28th Jan. 2014).
- [RF11] Rosnisa Abdull Razak and Fairul Rizal Fahrurazi. 'Agile Testing with Selenium'. In: *Software Engineering (MySEC), 2011 5th Malaysian Conference in*. IEEE. 2011, pp. 217–219.
- [Sam] Emmanuel Sambo. *Relish: Tic Tac Toe*. URL: <https://www.relishapp.com/esambo/tic-tac-toe/docs> (visited on 5th Mar. 2014).
- [Sch11] Eric Schmidt. *Digital-Life-Design (DLD) Conference*. Jan. 2011. URL: <http://www.youtube.com/watch?v=-ThfcvKBSug&t=8m20s> (visited on 26th Nov. 2013).
- [Sco10] Karl Scotland. 'Aspects of Kanban'. In: *Methods and Tools 19.1* (Summer 2010), pp. 3–14. URL: <http://www.methodsandtools.com/archive/archive.php?id=104>.
- [Sha11] Alan Shalloway. 'Demystifying Kanban'. In: *Cutter IT Journal 24.3* (Mar. 2011), pp. 12–17. URL: <http://www.netobjectives.com/files/resources/articles/Demystifying-Kanban.pdf>.
- [Sla12] Wolfgang Slany. 'Catroid: A Mobile Visual Programming System for Children'. In: *Proceedings of the 11th International Conference on Interaction Design and Children*. ACM. 2012, pp. 300–303.
- [SS13] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. July 2013. URL: <https://www.scrum.org/Scrum-Guide>.
- [Süd12] Medienpädagogischer Forschungsverbund Südwest. *KIM-Studie*. 2012. URL: http://www.mpfs.de/fileadmin/KIM-pdf12/KIM_2012.pdf (visited on 26th Mar. 2014).
- [Suto4] Jeff Sutherland. 'Agile Development: Lessons Learned from the First Scrum'. In: *Cutter Agile Project Management Advisory Service: Executive Update 5.20* (Oct. 2004), pp. 1–4.
- [Tat] Bruce Tate. *Behavior-driven testing with RSpec*. URL: <http://www.ibm.com/developerworks/web/library/wa-rspec/> (visited on 10th Mar. 2014).

Bibliography

- [UDo7] Chris Ullman and Lucinda Dykes. *Beginning Ajax*. Wiley.com, July 2007. ISBN: 9780470106754.
- [Was10] Tony Wasserman. 'Software Engineering Issues for Mobile Application Development'. In: *FoSER* (2010). URL: http://works.bepress.com/tony_wasserman/4.
- [WH12] Matt Wynne and Aslak Helleøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, Jan. 2012. ISBN: 9781934356807.
- [Wro11] Luke Wroblewski. *Mobile First*. Jeffrey Zeldman, 2011. ISBN: 9781937557027.