Lightweight requirements engineering in agile web development projects

Mike Arvela

University of Tampere
Department of Computer Sciences
M.Sc. Thesis
Supervisor: Zheying Zhang
28.6.2010

University of Tampere
Department of Computer Sciences
Mike Arvela: Lightweight requirements engineering in agile web development projects
M.Sc. Thesis, 74 pages
June 2010

Abstract

Web applications have become more commonplace, but it is still common for development projects to fail to meet their intended goals either in terms of budget, time-to-market or quality. Although increasingly popular in the recent years, the introduction of agile software development methodologies does not seem to make a significant difference as far as project success is concerned.

I present that the problems are often due to failures or shortcomings in the discipline of Requirements Engineering (RE), a set of practices present in all software projects regardless of the development model being used. This thesis attempts to gather data and present a set of concepts and ideas which, introduced in an agile web development project, help to discover and maintain requirements critical for the success of the project. Vixtory, a prototype of a web-based tool for lightweight requirements documentation, is also presented and evaluated.

Keywords: Web Application Development, Agile Software Development, Requirements Engineering, Requirements Documentation, Lightweightness, Agile Tool Support.

Acknowledgments

This thesis would not have been possible if it wasn't for the following people. First of all, I want to express my gratitude towards my supervisor Zheying Zhang for her resilience and all the good advice throughout the writing process, as well as Timo Poranen for always being a pleasure to collaborate with.

I also want to thank Henri Sora from *Ambientia* for making the subject possible in the first place, and Matias Muhonen for making me believe it could be done.

Finally, I want to thank my parents for always believing in me, and my wife Taika for being there for me. Love you guys.

Tampere, June 2010
Mike Arvela

Contents

1 Introduction

The web has become a significant application platform in the recent years. The special features and intricacies of applications targeted for the web have been studied [Lowe, 2001; Murugesan *et al.*, 2001; Murugesan & Ginige, 2005], yet signs of development methods tailored specifically for the web are nowhere to be found. Although some suggestions on how one should develop for the web have been made in the past [Murugesan *et al.*, 2001], the industry at large seems to follow the same development practices regardless of the platform the development is targeted for.

Having been adopted in increasing numbers during the last few years, agile software development methodologies are often portrayed as contributors to successful software development projects. A vast majority of companies doing software development claim to use iterative, if not agile, software development processes [Abrahamsson, 2008]. Yet not all software projects are successful or meet their intended goals. Despite the increase of knowledge among software development practitioners in the area of project management and software development methodologies, there has been no significant improvement in overall software project success rates. For example, The Standish Group CHAOS Report [Standish, 2009] reveals last year having the highest project failure rate in a decade. Projects still frequently struggle with problems related to requirements, with nearly a half of development problems stemming from them [Hall *et al.*, 2002].

The fundamental practices of Requirements Engineering (RE) are present in all software projects regardless of the development model being used. Requirements define the needs and expectations a user has for the product; although definitions of success vary, a software project may generally be considered successful if it both meets the requirements set for the system developed and also does that in the given budget and time frame. The problem domain is illustrated by the "project triangle" [Bethke, 2003]: of *quality*, *time* and *price* you may pick any two, and the rest is up to you to make it happen.

The success in defining and maintaining requirements in a software project is a significant contributor to the overall success of a project. Failing in coming up with relevant requirements may cause the system to be built badly, or even the wrong system being built. Many of the aspects in agile methodologies revolve around the issue of how requirements should be gathered and maintained. Still, as opposed to the more traditional and document-oriented ways of managing requirements, remarkably little formalism and concrete advice exists on how RE should be practiced in agile projects.

It has become obvious that agility is no "silver bullet", and no universal fit-for-all project management solution exists. An answer is unlikely to be found in either of the stereotypic extremes of doing extensive documentation and barely any documentation at all but rather in a balance of the two. Thus, it seems reasonable to think that the traditional ways of software development are not to be abandoned as a whole but should rather be rethought to enable easy adaptation and improve communication and collaboration between various stakeholders.

In the pursuit for a sweet spot between formality and pragmatism, we set out to explore the intricacies of web applications, attempting to find answers to the following questions:

- Is it possible to incorporate the phases of RE within a typical agile development process?
- What kind of tool would it require to provide support for handling requirements in agile web development projects?

To answer these questions, we identify the common grounds of agile, RE, and web and apply to them the idea of *lightweightness*. We also present a prototype of an agile requirements documentation tool for web-based projects embracing the principles presented and analyze the findings of a group having evaluated it.

This thesis consists of eight chapters. We begin by discussing the features and properties of the web as an development environment in Chapter 2. The next chapter looks into some common web development approaches and discusses the characteristics and specialities of web application development. The importance of requirements and the Requirements Engineering (RE) is introduced in Chapter 4, where the RE lifecycle is mapped to agile methodologies based on the concept of lightweightness. In the next chapter, a conceptual model and theoretical foundation for combining agile, RE and web is presented.

Chapter 6 introduces Vixtory, a web-based requirements documentation tool based on the principles laid out previously. In the following chapter, the evaluation done on Vixtory is presented and its findings analyzed, with Chapter 8 finally concluding the thesis, summarizing the research and discussing possibilities for future work.

2 Overview of web applications

2.1 Background

The World Wide Web (WWW) is a global success story. In a relatively short time span it has grown increasingly mainstream and has been widely adopted by people of various ages and origins. What has made it possible for WWW to grow so fast is that its content is not only created by few but as an collaborative effort by people from all over the world, companies and individuals alike. Figure 2.1 illustrates the growth of the web on a logarithmic scale, with the number of web sites having grown rapidly throughout the early 21st century.

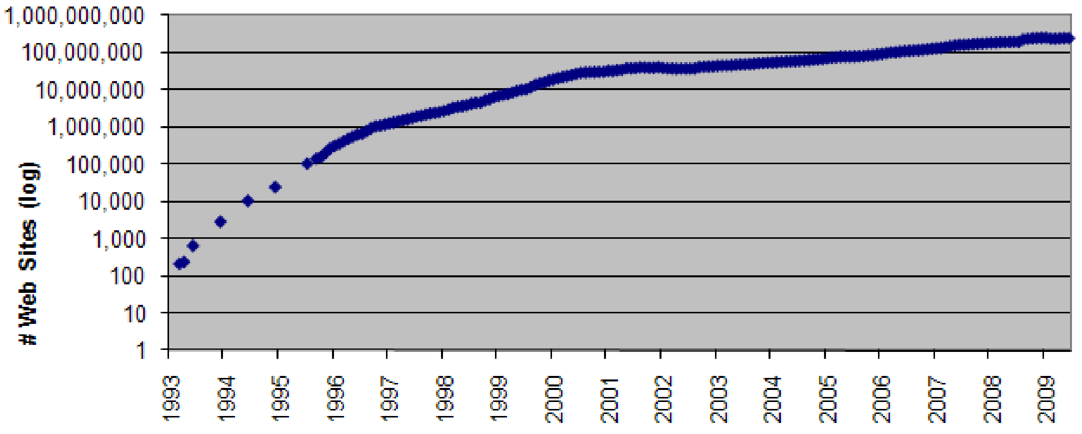


Figure 2.1 Logarithmic growth of the web [Zakon, 2010]

Not only does the web enable access to a huge database of information, but it also allows for new styles of connectivity. Information is readily available and mobile devices boost its usage by allowing people to stay in touch virtually no matter where or when they are located.

For many people, Internet in general has become so commonplace that we take it for granted. Yet, in the context of computers and software development, the web is a newcomer. Even though computers have been programmed for more than half a century, changes in programming paradigms are if not absent, then at least very uncommon. One could consider the concept of object-oriented programming as an example. Although having its roots as far back as the 1960s [Dahl & Nygaard, 2008], it was popularized at the latest by the release of the Java programming language in the 1990s [Oracle, 2010], and at the dawn of the

2010s still seems to be the dominant programming paradigm [TIOBE, 2010].

Many existing paradigms such as object-orientedness, Relational Database Management Systems (RDBMS) and the Model-View-Controller (MVC) pattern are applied to web development as such. This might lead one to think that developing software for the web does not substantially differ from conventional software development, but instead just provides a new view layer, as opposed to windows in graphical user interfaces or text in character-based ones.

However, we present that the web as a software development environment presents an outstanding cross-concern paradigm in itself and, needs to be treated as such. In the context of software development, observing the web as just an alternative presentation layer is not only a hasty, but also a potentially hazardous conclusion. As it is only human to apply previously acquired knowledge to new situations, there is always a risk of underestimating the importance of acknowledging the special characteristics of web applications. In order to mitigate risks and become successful in a web development project, one must be aware of these special features and adapt to new ways of working.

Murugesan and Ginige [2005] present a categorization of web applications based on their functionality. As shown in Table 2.1, they present six categories of web applications: *informational, interactive, transactional, workflow oriented, collaborative work environments*, and *online communities and marketplaces*. These categories effectively cover all existing web applications, ranging from online newspapers to games and distributed authoring systems. It is also presented that the scope and complexity of web applications is much varied. It ranges from short-lived or small-scale applications to large-scale enterprise applications distributed across the Internet, as well as corporate intranets and extranets only visible to a limited group of people.

2.2 Defining a web application

The Oxford English Dictionary defines an *application* “a program or piece of software designed and written to fulfill a particular purpose of the user”. A web application can be thought as the web equivalent of a conventional desktop application, where instead of having program files stored on a local hard drive, some or all parts of the software are downloaded from the web each time the application is run.

While a web application might technically be implemented in various technologies such as Java Applets or Adobe Flash, most modern web applications run

Functionality/Category	**Examples**
Informational	Online newspapers, product catalogues, newsletters, manuals, reports, online classifieds, online books
Interactive	Registration forms, customized information presentation, online games
Transactional	Online shopping (ordering goods and services), online banking, online airline reservation, online payment of bills
Workflow oriented	Online planning and scheduling, inventory management, status monitoring, supply chain management
Collaborative work environments	Distributed authoring systems, collaborative design tools
Online communities, marketplaces	Discussion groups, recommender systems, online marketplaces, e-malls (electronic shopping malls), online auctions, intermediaries

Table 2.1: Categories of web applications based on functionality [Murugesan & Ginige, 2005]

on standardized and universally available browser technologies such as HTML and JavaScript. Recently, the web developer community was buzzing following the rare announcement made by Apple about not adding Flash support in their popular mobile devices, the iPhone and the iPad [Jobs, 2010].

The terms *web application*, *web site*, and *web-based system* are seen frequently. Even if the term 'web application' puts more weight on the functionality whereas a 'web site' emphasizes the content and 'web-based system' might be used as an umbrella term to describe them both, regardless of their semantic nuances the three are often used interchangeably. From the viewpoint of a user, a web application is basically a web site with some intriguing functionality. Although the term 'web application' might in some contexts also be used to refer to client programs that communicate with web servers using standard web protocols, it is most commonly associated with browser-based applications. This is also the case in the scope of this thesis.

Characteristic to a browser-based web application is that it is dynamic, cross-platform and multi-tier.

Dynamic

The World Wide Web started as a small collection of static user-created pages written by scientists to exchange information with each other. Publishing content involved only HTML markup used for formatting text and no programming was required, or initially even supported. No enablers for user interaction existed at the time. Back then, the content on the web was purely *static*.

In 1993, a foundation for interaction on the web was laid in the form of the Common Gateway Interface (CGI)[1], making it possible for a user to post data to the web server. By the introduction of the CGI specification, it was now possible to actually implement applications adhering to this definition, now in a web environment. As opposed to non-changing static content, all web applications may be considered *dynamic* web pages, or vice versa.

Cross-platform

While there have been web applications for almost two decades now, it was not until the late 2000s that the web became acknowledged as a considerable rival for standard desktop applications. This was made possible by the evolution of the web browsers and the invention of AJAX (Asynchronous Javascript and XML), which now allowed to submit and receive data from the server without completely reloading the current page in view. As illustrated in Figure 2.2, an intermediate "Ajax engine" layer is introduced between the browser client and the web server. The engine uses an *XmlHttpRequest* object provided by the web browser to make additional HTTP calls asynchronously, without the page being refreshed. This leads to a better end-user experience, as additional content may be loaded for without the user even noticing it.

Consequently, a new term *Rich Internet Application* was coined. Typical to these "new kinds of" web applications was that they were now *native* for the web, working on any modern web browser without requiring any third party extensions.

As it is rare to come across a web site that does not contain any dynamic or generated content or means of submitting data, it is safe to say the web has

[1] `http://tools.ietf.org/html/rfc3875`

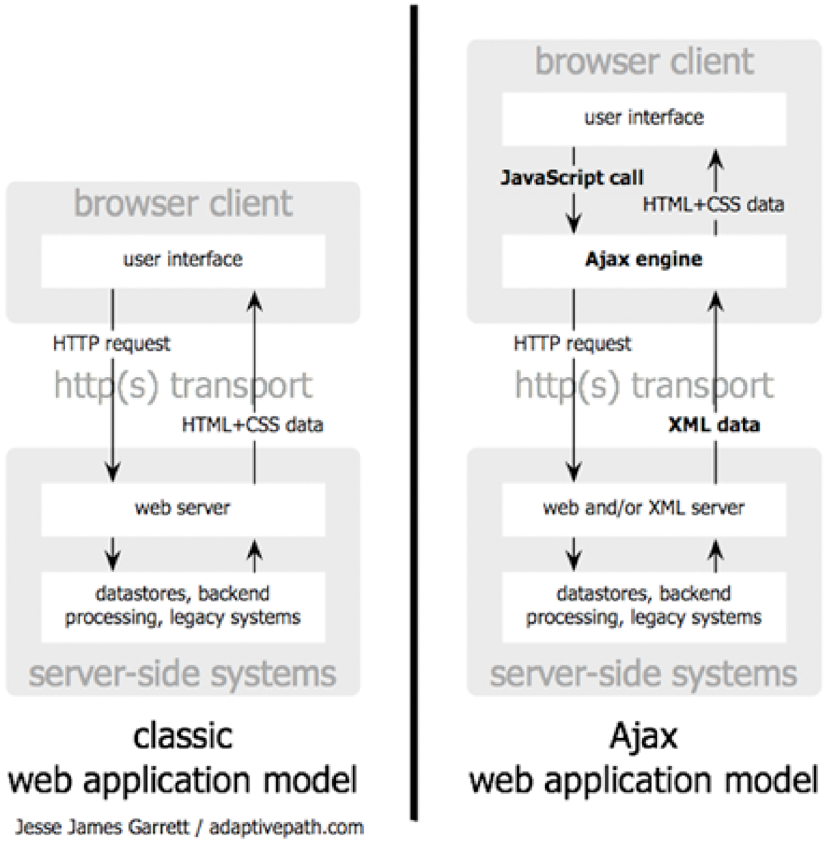


Figure 2.2 AJAX web application model [Garrett, 2005]

become dynamic. As such, most modern web pages may be referred to as "web applications". However, as broad as the term is, a "web application" is nowadays likely to be understood as something that might as well exist as a desktop application.

The brilliance of WWW as an application platform lies in its principle of "least common denominator": basically any device supporting a set of commonly accepted web technologies and protocols is capable of displaying content and providing basic user interaction via a web browser program. Whereas there has been the convention of calling desktop software either "native" or "cross-platform" depending on the environment it was designed to run in, web applications are by definition *cross-platform*.

Multi-tier

Due to the client-server nature of the web (Figure 2.3), business logic could be run either on the server, on the client, or on both of them. Typically a combination of both is used, but this has not always been the case. In the 1990s, much effort was placed in providing user interaction with external components on the client side. Java Applets and Microsoft ActiveX components are probably the best known of these. Adobe's Flash technology is still in use today, but there have already been discussion on whether it will be ultimately replaced by the upmarch of new web standards in the near future.

Although the technology stack used to render content on the client side (web browser) largely remains the same, the server side is different. The components, programming languages and server run-times used on the server to produce the content seen by the end-user may vary greatly.

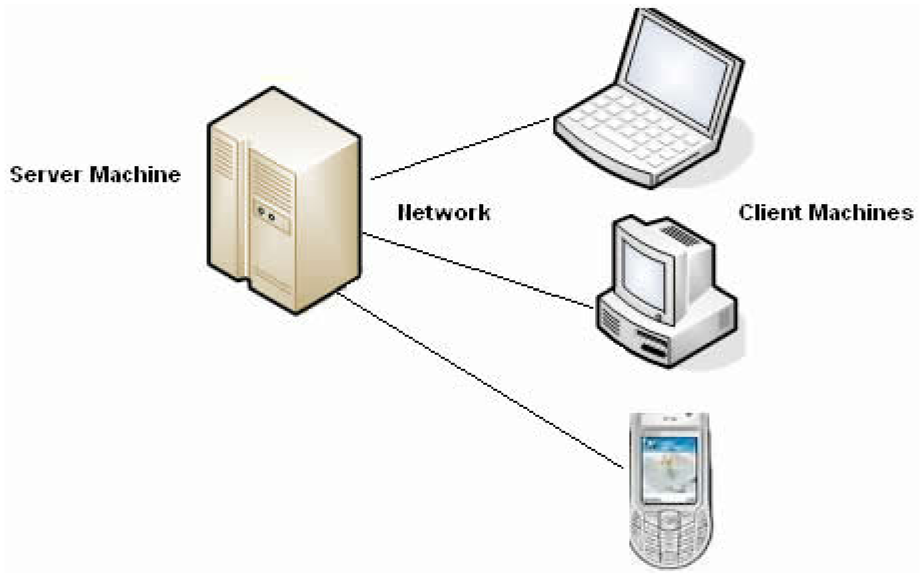


Figure 2.3 Client-server architecture[2]

[2] `http://www.acm.org/ubiquity/volume_9/images/client_server_architecture.jpg`

2.3 Features unique to web applications

Lowe [2001] discusses the differences between web and 'conventional' systems. In his research, the main difference between the two is that the former commonly have an impact in the business models of the adopting organization whereas the latter do not. In most cases, the concept of conventional systems is generalizable to desktop applications, which practically leads to comparing applications that are accessed using a web browser and those that are not.

There are a number of features unique to Web based systems:

Distributed nature of the Internet. The Internet would never have been born if it weren't possible for the great public to get connected to it. Since fundamentally all entities connecting to the Internet are assigned an IP address, it also means it is theoretically possible for everyone to run their own server and thus host content that can be accessed by anyone. The regulatory authorities excluded, everyone also has the same possibilities of publishing online.

Visibility to external stakeholders. Unless specifically protected, anyone will have access to the same web resources. The very client-server nature of the Internet dictates that everyone accessing the same server will be served the same content.

Rapidly changing nature of underlying technologies. The number of technologies used to create web pages has exploded since the 2000s. In the early years most of the dynamic content served online were created by invoking CGI scripts built in Perl or even the C programming language. Nowadays there are dozens of sophisticated web frameworks for a plethora of programming languages designed specifically for web development.

It is also not uncommon for popular web services to change their background technologies. For example in 2009, Twitter, a hugely popular microblogging service ported its backbone from Ruby to Scala to multiply its throughput [Venners, 2009].

Lightweight component-based structure (often open-source components). Modern web development does not emphasize any particular technology. It is possible to build a scalable, efficient and rich web site with tools that are freely available for anyone to use, many of which are also open source. Since there are numerous pre-built and well tested components available for different purposes and technologies, it is usually not feasible or necessary to develop one's own.

More sophisticated information architecture. The aspects of information architecture such as content viewpoints, interface metaphors and navigational structures are significantly more intricate than in conventional software systems.

Practices such as Search Engine Optimization (SEO)[3] contribute to the demands regarding information architecture. In the 1990s, it was common for people to gather links on their home pages. There were also services providing categorized links. Little by little search engines such as Altavista and Yahoo became more commonplace and ultimately Google was able to crawl sites so efficiently it became dominant in the market. Today SEO has become a business of its own. It concentrates on how the contents of a site should be presented, organized and optimized for it to reach as high search engine rankings as possible. This has to do with the nature of Internet; because content is there for everyone to see, efforts must be made regarding to information architecture to stand out from its competition.

Furthermore, Lowe [2001] presents that many web-based applications have a significant impact on how the organization interacts with its customers and external stakeholders. This typically leads to a change in not only the business processes but the business model that relies on them. In other words, the introduction of a web-based system changes the problem being addressed and hence affecting the requirements of the solution. This makes the problem and the solution interdependent, as they are mutually constituted.

This also suggests agile approaches embracing change would be a particularly powerful fit for web-based development, since according to Lowe [2001] requirements will be affected by the very introduction by the solution itself!

2.4 Technological considerations

The client-server nature of the web sets boundaries and limitations to functionality of web applications. Considering that HTTP itself is a stateless protocol by nature, web development has already come a long way in terms of what is possible to implement in a completely browser-based environment. A good example of this is JavaScript, which despite being a core web technology (and still the only technology enabling UI functionality in any graphical web browser), was for many years overlooked and considered malicious by both web developers and end

[3] `http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=35291`

users. In the recent years, though, JavaScript has become increasingly popular in the so-called *Rich Internet Applications*, and the ways it is being written have changed rapidly, not least by the increased interest among web enthusiasts but also by multi-purpose component libraries such as jQuery[4], Prototype[5] and the Yahoo YUI library[6].

The downsides to providing "rich content" are obvious. Most web applications of today that are considered *rich* might not work at all with Javascript or cookie support disabled. As the limits of original HTML specification have been pushed to the maximum, it has also set requirements on client software on a completely new level. This is especially problematic in case of legacy web applications which were only designed for a specific browser version. However, times have changed, and supporting the proprietary features of these old browsers is challenging, if not downright impossible.

In addition to being able to better exploit existing techologies, it currently looks like the family of core web technologies will grow significantly in the early 2010s. The most modern approaches making use of technologies such as HTML5 (with built-in `video` tags), accelerated web graphics (WebGL), advanced drawing routines (the `canvas` element), web sockets for keeping a connection alive between the web server and the browser, are likely to eventually, if not render third-party add-ons such as Adobe Flash obsolete, at least make them less significant.

Core web technologies such as HTML, CSS style sheets and JavaScript have become so well known and commonplace that their use has been extended beyond that of WWW environment. Special platforms have been created so that these technologies may be used to create applications for e.g. mobile devices. These applications are commonly called "widgets". Popular widget runtimes include JIL[7], Nokia Web Runtime (WRT)[8], Palm WebOS[9], Opera Widgets[10], PhoneGap[11], and the WholeSale Application Community[12].

[4] `http://jquery.com/`

[5] `http://www.prototypejs.org/`

[6] `http://developer.yahoo.com/yui/`

[7] `http://www.jil.org/`

[8] `http://www.forum.nokia.com/Technology_Topics/Web_Technologies/Web_Runtime/`

[9] `http://palmwebos.org/`

[10] `http://dev.opera.com/sdk/#widgets`

[11] `http://www.phonegap.com/`

[12] `http://www.wholesaleappcommunity.com/`

Although the techniques used to develop Widgets are the very same ones that are also used to develop for the web, the Widget development differs in terms of target environment and usage in general; whereas Web applications are designed to be viewed in a web browser, Widgets do their best to mimic native applications. It might be impossible for the end user to tell whether the application he is using is actually a widget containing HTML markup and JavaScript functionality. As an example, consider Figure 2.4 displaying a WRT weather widget running on a Nokia N97.

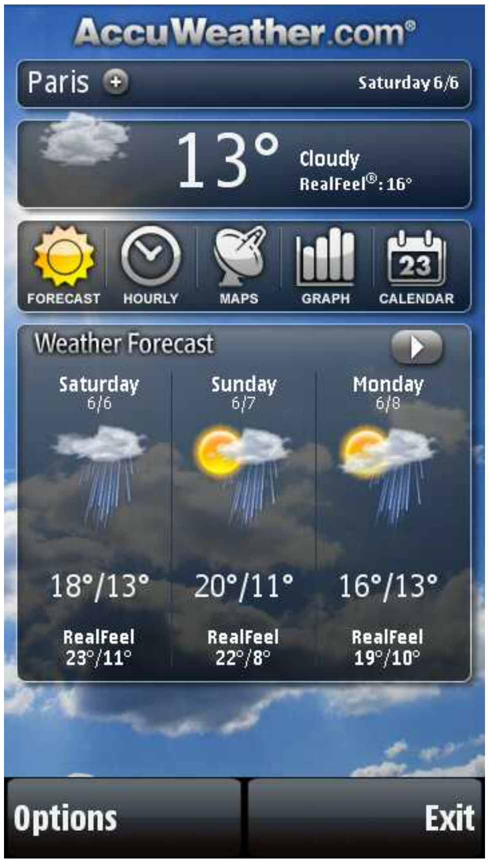


Figure 2.4 A WRT weather widget by AccuWeather.com

3 Web development approaches

Web development is a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology. – Thomas A. Powell

3.1 Current practice

The number of traits unique to web development suggests paradigms specific for web application development be required. However, quite interestingly, evidence of current practice points towards the contrary. Even though research has been made regarding methodologies for specifically targeting web-based development [Murugesan & Ginige, 2005; Rosson *et al.*, 2004; McDonald & Welland, 2001; Lowe & Eklund, 2002; Kautz & Nørbjerg, 2003], it still seems common for web application development projects of today to make use of the exactly same development paradigms as any other software development projects.

This view is shared by Murugesan et al. [2001; 2005], who suggest web application development approaches have been ad hoc. According to them development, management and quality control practices are commonly neglected. Furthermore, most practices are considered to heavily rely on the expertise of individual developers and their own practices. Proper testing and documentation approaches are absent.

They partly put the blame on the rapid growth and evolution of the web. Other perceived causes include underestimating the complexity of web-based systems and web's legacy as an information medium rather than an application medium. The process of generating content for the web may be seen as more of an authoring activity than something in which well-known software engineering and management principles could be applied.

Murugesan et al. [2001] have even expressed their concern on how the lack of disciplined web development approaches might eventually escalate to a "Web crisis". They present an approach called "Web Engineering" for building scalable and high-quality web applications and avoiding this "potential chaos".

What makes the situation interesting is that in many cases the absence of disciplined development approaches may be attributed to the adoption of agile methodologies. This is also the source of prevalent criticism towards agile: although a lot of good general level advice is given, there rarely are detailed instructions on how specific things should be done. Many organizations also claim to be doing agile, but the veracity of these claims has been questioned [Abrahamsson, 2008].

For instance, agile and lean methodologies such as Scrum[1], XP[2], and Kanban[3] seem to be present in building conventional desktop applications as well as projects targeting web or mobile devices. On the other hand, it might also be a case of compensating for the lack of processes with the use of common sense and *ad hoc* approaches, as encouraged by the agile development ideology.

In the early 2000s, Murugesan and Ginige [2005] regarded the existing web application development and maintenance practices as "chaotic and being far from satisfactory". It was proposed that a disciplined web development process was required in order to successfully build large and complex web-based systems and applications.

They present that because of the evolutionary nature of web applications, a sustainable model for developing them is to "follow an evolutionary development process where change is seen as a norm and is catered to". This is analogous to the agile principle of embracing change, but in contrast to valuing individuals over processes, the Web Engineering discipline seems to emphasize the importance of having a well-defined and disciplined process.

The overview of their proposed web development process is presented in Figure 3.1. The process begins with *context analysis*, in which the major objectives and requirements of the system are elicited. In *system architecture design*, the components of the system and how they link to each other, are decided. A process model and project plan are then agreed upon. Then, the various components of the system and web pages are designed, developed, and tested. After deployed online, the system is subject for continuous maintenance, for which maintenance policies and procedures are formulated, based on decisions in the architecture design stage. Also emphasized is the importance of close coordination as the facilitator of continuous project management and quality control practices throughout the project.

[1] `http://www.scrumalliance.org/`

[2] `http://www.extremeprogramming.org/`

[3] `http://www.limitedwipsociety.org/`

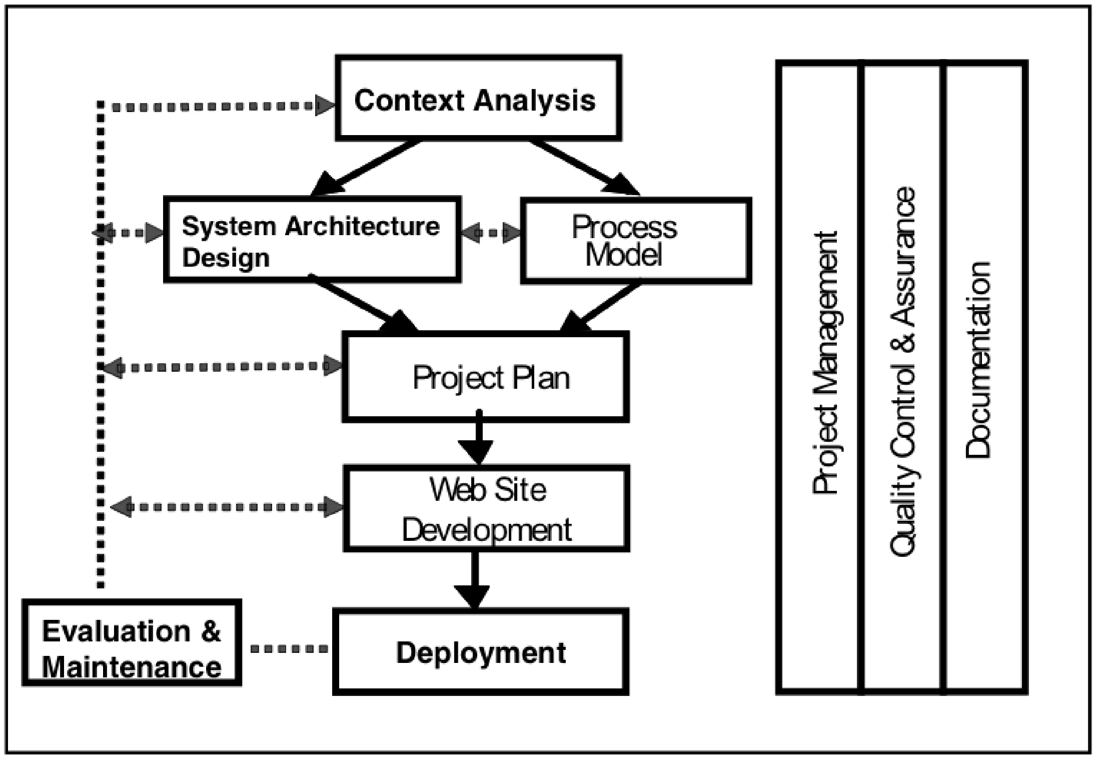


Figure 3.1 Web development process proposed in the Web Engineering discipline [Murugesan & Ginige, 2005]

3.2 Agile software development

The term *Agile software development*, often referred to as just *Agile*, was coined in the year 2001. It was born by the formulation of the Agile Manifesto [Beck *et al.*, 2001], a statement drafted by 17 representatives of various emerging software development methodologies. The Agile Manifesto reads as follows:

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

The Agile Manifesto itself is not a development methodology or a process. Instead, it is an umbrella term covering various actual development methodologies such as Scrum, eXtreme Programming (XP) and Rational Unified Process (RUP). In the recent years, most notably the late 2000s, agile methodologies have taken the software development world by storm. After decades of document and process heavy development disciplines, agile has been seen as a breath of fresh air. As a result, there has also been considerable interest in them by both developers and the academia.

The whirl around agility has been extensive and in many cases practice has not quite seemed to be able to live up to the hype. The fast and wide adoption of agile methodologies has resulted in not only developers but also management having unrealistic expectations of what is possible by "just forgetting about documentation and starting to write code", a common stereotypical misinterpretation of the essence of agile become so widespread that it has made a rarely seen appearance in mainstream media, as illustrated by the Dilbert comic strip in Figure 3.2. At its worst, this has culminated in situations where the bare introduction of agile methodologies has been expected to prevent failure in software projects [Abrahamsson, 2008; Schneider, 2002]. This combination of no prior experience and overly high expectations has resulted to many "agile drive-ins" failing and contributed to agile not having lived up to its promises.

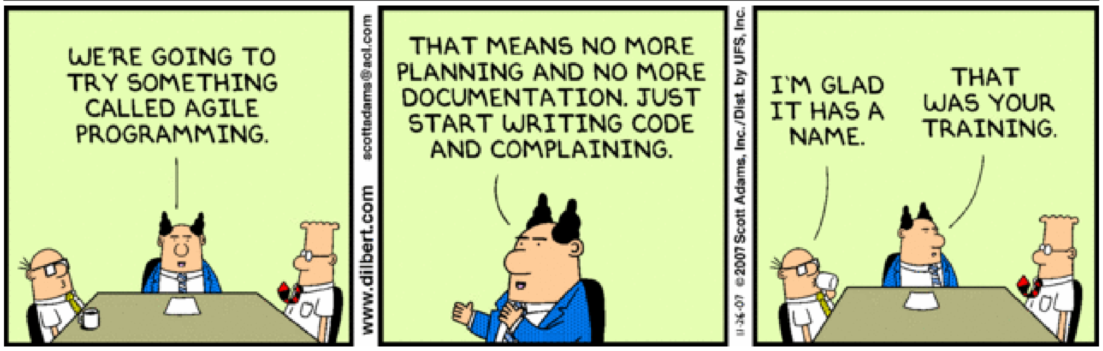

WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.
www.dilbert.com scottadams@aol.com
11-26-07 © 2007 Scott Adams, Inc./Dist. by UFS, Inc.

Figure 3.2 Dilbert on Agile Programming (`http://dilbert.com/strips/comic/2007-11-26`), used with permission.

While no final verdicts or omnipotent conclusions may be drawn, one of the most common criticisms towards agile development methodologies has been along the lines of that they mostly lay general-level guidelines but rarely give concrete advice of what to do. The term "silver bullet" [Brooks, 1987] is often used in referring to the disappointment of agile not being able to miraculously remedy any software project and make it succeed.

In spite of continual criticism and adverse judgement, agile continues to thrive. A concept of *Post-agilism* has been introduced to suggest agile is just the beginning of something better that will evolve from the current practices [Kohl, 2006; Gorman, 2006]. One notable example is the *Lean* movement, most notably the Kanban methodology [Patton, 2009; Sugimori *et al.*, 1977], which strives for minimizing all the extra work and phases ("waste") in a production chain. While Kanban is seen by some as the direction of evolution for agile, the opinion yet appears far from unanimous [Gat & Heintz, 2009].

As for web application development, Baskerville [2003] suggests that agile principles are better suited for building software for the Internet than traditional software development approaches. This is in line with the author's personal experiences, in part due to the frequent change caused by heightened emphasis of visual appearance.

3.2.1 Common development practices

This section briefly introduces various development disciplines or methodologies for use in web application development. Most notably they are not process models in themselves, but rather development ideologies that might be useful applied in certain parts or aspects of a project. As they are in general not mutually exclusive, more than one of them might be used at a time.

As often the case with any development methodology, no single discipline claims to make the problems and the risks go away, but rather tries to assist and mitigate some of the risks involved in a specific problem domain.

Test Driven Development

Test Driven Development (TDD) originated from an idea introduced by Kent Beck [2002], the creator of JUnit test framework and one of the Agile founding fathers. Originally associated with the Extreme Programming discipline, TDD has since become a paradigm in its own right. It still still often associated and used in conjunction with agile principles.

Test Driven Development in practice typically follows the following sequence [Beck, 2002]:

1. Write a test
2. Check if the test fails
3. Write some code
4. Run all tests
5. Refactor code

When the flow has been completed, it can be started again by writing or rewriting a test. It is sometimes emphasized that while encouraging test-first development, TDD is also strongly a design practice as it requires developers to think of the implementation and interfaces before writing actual code.

Behavior Driven Development

Behavior Driven Development (BDD) is a practice originally proposed in 2003 by Dan North [2003] as a response to Test Driven Development. Behaviour Driven Development extends Test Driven Development in that test cases are written in a near-natural language, sometimes called "business-readable domain-specific language" [Fowler, 2008]. This makes the test cases readable and understandable (although not necessarily writeable) by also people that are not programmers.

As the basis of BDD, North outlined three principles [Chelimsky *et al.*, 2010]:

1. Enough is enough: do as much planning, analysis, and design as you need, but no more
2. Deliver stakeholder value: everything you do should deliver value
3. It's a behavior: everyone involved should have the same way of talking about the system and what it does

The concept of BDD has been extended and improved upon during the last few years and today, numerous frameworks and tools exist for various programming languages and environments.

Continuous Integration

Continuous Integration (CI) is a quality control concept. It aims to *reduce risk* by both improving the quality of software and reducing the time taken to implement it. This is achieved by replacing the traditional model of testing only after programming has been done to testing frequently, preferably automatically, in small pieces. Continuous Integration was introduced in the Extreme Programming community in the early 2000s and discussed by the agilistas Kent Beck and Martin Fowler [2006].

In short, a developer submits his changes into a code repository, after which the software is built from the latest source by a CI system automatically. As a result, the software is *integrated* continuously, typically multiple times a day. The build process typically includes a set of tests that are being run, and the failing of which will also cause the actual build to be aborted. Optimally, a build run by the integration server would also generate artifacts such as the latest ready-to-run executable of the software, and possibly also deploy the build to a target environment.

Fowler [2006] lists a number of practices of Continuous Integration:

1. Maintain a single source repository.
2. Make your build self-testing
3. Everyone commits to the mainline every day
4. Every commit should build the mainline on an integration machine
5. Keep the build fast
6. Test in a clone of the production environment
7. Make it easy for anyone to get the latest executable
8. Everyone can see what's happening
9. Automate deployment

3.3 Web vs. conventional software development

It is crucial to realize the web is not only a different deployment environment but also sets demands of its own. Besides the unique characteristics of web applications, the Internet as a development environment is in general considered to intensify software development problems by emphasizing shorter cycle times and fast-changing requirements [Baskerville *et al.*, 2003].

Overmyer [2000] sees web development primarily as a communicational challenge, in which user experience and the how information is presented require more careful planning than conventional information systems. This view is in line with Lowe's emphasis on careful systems design and architecture.

Listing the differences between conventional and web software development (refer to Section 2.2 for an overview of the terms) helps to understand the important differences between the development processes.

Roughly, the web application characteristics mentioned above can be divided into two viewpoints: business and development.

Business characteristics

The *business* viewpoint is concerned with perks that are common for the Internet industry in general and features that are expected of web applications in the market. These factors include user experience (UX) considerations such as visual design, ease of use, accessibility, as well as general quality factors such as security policies, error recovery, fault tolerance, and the expected amount of downtime or maintenance work. All these relate to the customer's expectations on the attributes of the final product delivery. Business considerations include:

Increased importance of quality attributes. In a web environment, it is almost impossible to hide application flaws. In contrast, they are often visible directly to the end user. Be it usability, general performance or robustness, not being able to deal with quality issues carefully enough may lead to customer dissatisfaction. [Lowe, 2001]

Highly competitive. It is a common perception that given the amount of authoring tools widely available, "anyone can do a web site". There is a possibility of clients developing inappropriate expectations when they are dealing with a company that might, in the end, only be able to offer a little more than basic HTML and tentative graphic design. Lowe refers to this as "uninformed competitiveness". [Lowe, 2001]

Highly variable client understanding. Clients' understanding of their own needs and expectations evolves and improves during project. This may ultimately lead to a "requirements creep" where requirements keep pouring into the project, practically making it extremely hard to develop a product that satisfies user expectations. A contributing factor is also the client's poor understanding of the realities of a software project, which makes initial system definition difficult. This increases the importance of incremental and prototyping approaches. [Lowe, 2001]

Increased emphasis on user interface. As users can switch between web sites with minimal effort, it becomes critical to be able to engage users and meet their needs. Since there is heightened emphasis on visuality [Lowe, 2001; Murugesan *et al.*, 2001] and making a good first impression, it is crucial for a web application to have a competent user interface actively supporting the functionality the site provides.

Seemingly similar results may be achieved by different approaches. Companies may try to sell projects that are built on their own platform, regardless of whether the platform will ultimately scale and live up to the needs and expectations of the customer.

Diverse user base. Users of web applications are of diverse skills and capability. This multiplicity of user profiles complicates information presentation and user interface design. [Murugesan *et al.*, 2001, 7]

Content-drivenness. Most web applications are document-oriented containing static or dynamic web pages. Development also often includes development of the actual content to be presented. [Murugesan *et al.*, 2001, 7]

Short time frames for initial delivery. Web as a development platform combined with modern development tools allows for rapid development. This in turn leads to web development efforts often having delivery schedules shorter than those of conventional IT projects [Lowe, 2001]. This also makes it difficult to apply the same level of formal planning and testing as used in conventional software development [Murugesan *et al.*, 2001, 7].

Development characteristics

The *development* point of view mainly considers web application characteristics that are not visible to the end user. These relate to the architecture of the software being built, the technologies used, and the experience of the developers and team members involved. An important role is also played by maintenance, the importance of which should never be downplayed. A market research conducted in 2007 [Neemuchwala, 2007] reported 47% of software projects having higher-than-expected maintenance costs. Characteristics relating to the development perspective include:

Fine-grained evolution and maintenance. Since it is possible to make changes to a web application that are immediately accessible to all its users, web sites tend to evolve in notably smaller increments than conventional applications. Content-intensive sites may also be updated regularly with centralized administrative tools without needing any additional changes to the site itself.

Rapidly changing technologies. Compared to conventional desktop systems where it might not be possible at all to change from one implementation technology to another, the rapid evolution of web development tools increases the importance of creating flexible solutions and encouraging the use of reusable data formats such as XML (eXtended Markup Language), JSON (Javascript Object Notation) and YAML (Yet Another Markup Language).

Another area of impact lies in the expertise of the developers. As the number of available technologies grows, it becomes more and more important for developers to widen their knowledge to be able to adapt to change and not be limited to development on a specific stack of technologies.

Highly variable developer background. The people building and developing web applications vary greatly in their background, skills, knowledge and understanding of the underlying technologies as well as in their perception of the web. It is also likely that the nature of web requires more diverse developers, as development is done both in the backend (server side) and frontend (client side) with several aspects as security, scalability and visual appearance often emphasized.

Open modularised architectures. The building blocks of web applications are often (although not necessarily always) modularized open-source components. A site may be constructed from several COTS (*commercial-off-the-shelf*) components, possibly from multiple vendors. [Lowe, 2001]

4 Disparity between RE and agile

4.1 Overview of requirements

Requirements are a specification of what should be implemented. They describe how the system should behave, information on the application domain, constraints on the system's operation, or specifications of its properties and attributes. [Kotonya & Sommerville, 2004]

As Hunt and Thomas [1999] put it, "Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need."

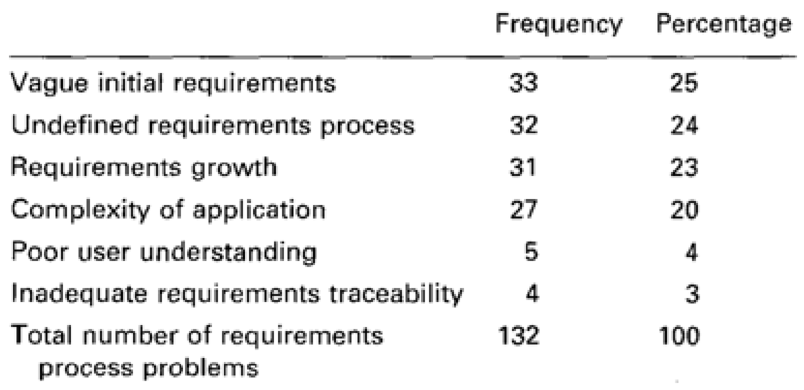
Requirements can be further divided into *functional* and *non-functional* ones. While functional requirements (FR) describe what a products should do, non-functional requirements (NFR) place restrictions specify external constraints that the product must meet [Kotonya & Sommerville, 2004]. These may include aspects such as product reliability, supply chain management or legal issues.

4.1.1 Requirements as project success factors

There have been extensive studies on what are the factors that make software projects succeed, as well as fail. While various coefficient elements continue to be identified and the so-called "lightweight" (agile and lean) software methodologies have become more commonplace, there have still been no signs of improvement in the overall success of software projects.

Even though many different factors contribute to the success of a software project, requirements continue to be a key element which in no project can be overlooked. The comprehensive detail of RE processes is today often seen as "anti-agile" and too document-oriented, whereas agile methodologies emphasize value and direct results. From a lean point of view, extensive requirements documentation might be perceived as waste that could well be optimized or even replaced by communicating them in a totally different way.

Studies and research back up the argument on requirements being one of the most important factors in the success of a software project. In a focus group study conducted in twelve software companies [Hall *et al.*, 2002], 48% of all development problems stemmed from requirements. Of those requirements problems, 63% could be attributed to organizational factors external to the requirements process. A classification of the problems identified in the requirements processes themselves are presented in Figure 4.1. In this classification, the biggest single



	Frequency	Percentage
Vague initial requirements	33	25
Undefined requirements process	32	24
Requirements growth	31	23
Complexity of application	27	20
Poor user understanding	5	4
Inadequate requirements traceability	4	3
Total number of requirements process problems	132	100

Figure 4.1 Classification of requirements process based requirements problems [Hall *et al.*, 2002]

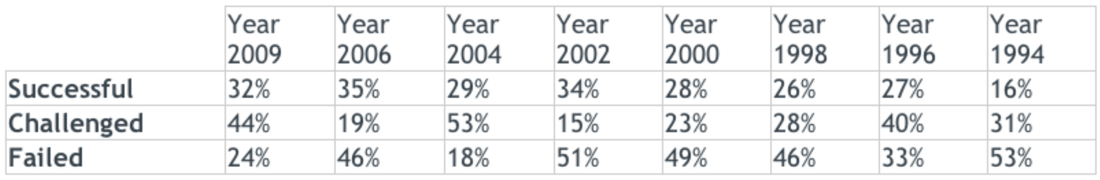
problem is the vagueness of initial requirements, followed by undefined requirements process, growth of requirements and the complexity of the application. All of these were almost similar in scale and accounted in total to some 92% of the requirements problems discovered. Overall, most of the experienced problems were organizational, and the results suggested a relationship between requirements problems and process maturity.

Keil [1998] identified failing to understand requirements as the third most important risk factor in software projects, with incomplete requirements being the biggest reason for software project failures.

Probably the most frequently referred reports and measurements are those of the Standish Group CHAOS reports. The most famous and also notorious of these reports is the one from 1994, the numbers reported in which conclude that only some 16% of all projects were considered a success by all measures. [Standish, 1994]

The Standish Group reports have been criticized for only measuring project success by time, budget and requirements while omitting quality, risk and customer satisfaction [Dominguez, 2009]. Even so, they still give a rough indication of the scale of projects that are not finished on time or within budget. While both the measurements and the means to achieve them may be questioned, there is yet little reason to doubt the lists gathered on the factors that contribute to overall project success.

The Standish Group reports typically classify projects into *Successful*, *Challenged* and *Failed*. Successful projects are completed on-time and on-budget, with all features and functions as specified initially. Challenged projects are completed



	Year 2009	Year 2006	Year 2004	Year 2002	Year 2000	Year 1998	Year 1996	Year 1994
Successful	32%	35%	29%	34%	28%	26%	27%	16%
Challenged	44%	19%	53%	15%	23%	28%	40%	31%
Failed	24%	46%	18%	51%	49%	46%	33%	53%

Figure 4.2 An overview of Standish Group CHAOS Report percentages 1994-2009 [Dominguez, 2009]

and operational but over budget, over the time estimate, and with fewer features than specified. Failed projects are ones that are cancelled at some point during the development.

Figure 4.2 presents the variation in success rates from 1994 to 2009. In the light of numbers, year 1994 is the darkest one as far as software project success is considered, with 53% of all measured projects failing. Even if the numbers for year 2009 indicate a much better result with less than half of the failure rate, a great deal of variation can still be observed between reports.

Top success factors
1. User involvement
2. Executive management support
3. Clear statement of requirements
4. Proper planning
5. Realistic expectations

Table 4.1: Top factors in "successful" software projects [Standish, 1994]

Table 4.1 lists five of the top ten project success factors. Even if these elements alone will not guarantee a successful project, according to the Standish Group, they will enable a much higher probability of success, if done well. The differentiators in projects that proved challenged are listed in Table 4.2. These projects include ones that were over budget, over time, or did not contain all functionality originally specified. Finally, a number of factors in projects that were considered a failure are listed in Table 4.1.1. Most notably, from our point of interest, the incompleteness of requirements is listed as the first and most notable failure factor.

Top "challenged" project factors	
1.	Lack of user input
2.	Incomplete requirements & specifications
3.	Clear statement of requirements
4.	Lack of executive support
5.	Technical incompetence

Table 4.2: Top factors in "challenged" software projects [Standish, 1994]

Top failure factors	
1.	Incomplete requirements
2.	Lack of user involvement
3.	Lack of resources
4.	Unrealistic expectations
5.	Lack of executive support
6.	Changing requirements & specifications
7.	Lack of planning
8.	Didn't need it any longer
9.	Lack of IT management
10.	Technical illiteracy

Table 4.3: Top factors in "failed" software projects [Standish, 1994]

In total, requirements were closely related to the top features of all the three listings. In successful projects, requirements were stated clearly and the users were involved, whereas in the challenged and failed ones the incompleteness and volatility of requirements and specifications, together with the lack of user involvement, contributed to the decline of the project.

While no simple path to success is to be found, there are signs that can be learned from. With requirements capturing the essence of the project, neglecting them (or the users they originate from) may result to undesirable outcomes. Improving customer collaboration and transparency on the whole can make for better and more successful projects.

4.2 On rework and quality of requirements

The significance of rework has also been studied. Preferably, not only would all the requirements be gathered the first time, but also they should be the correct ones. Boehm [1988] did comprehensive work on studying development costs related to rework and requirements as early as 1988, with remarkable findings. Not only can rework consume 30 to 50 percent of total development costs, but requirement defects alone account for some 70 to 85 percent of the rework costs. They also found that it is much more inexpensive to fix or rework software in the earlier parts of its life cycle than in the later phases. This is supported by the illustration in Figure 4.3.

A significant observation made by Boehm was also that 80 percent of the rework costs typically result from 20 percent of the problems.

Taking into account that the strategy of *failing early* is a commonly encouraged practice among seasoned software developers and testers [Shore, 2005; Hunter, 2005], evidence seems to suggest that defects are indeed best discovered and fixed early in the project.

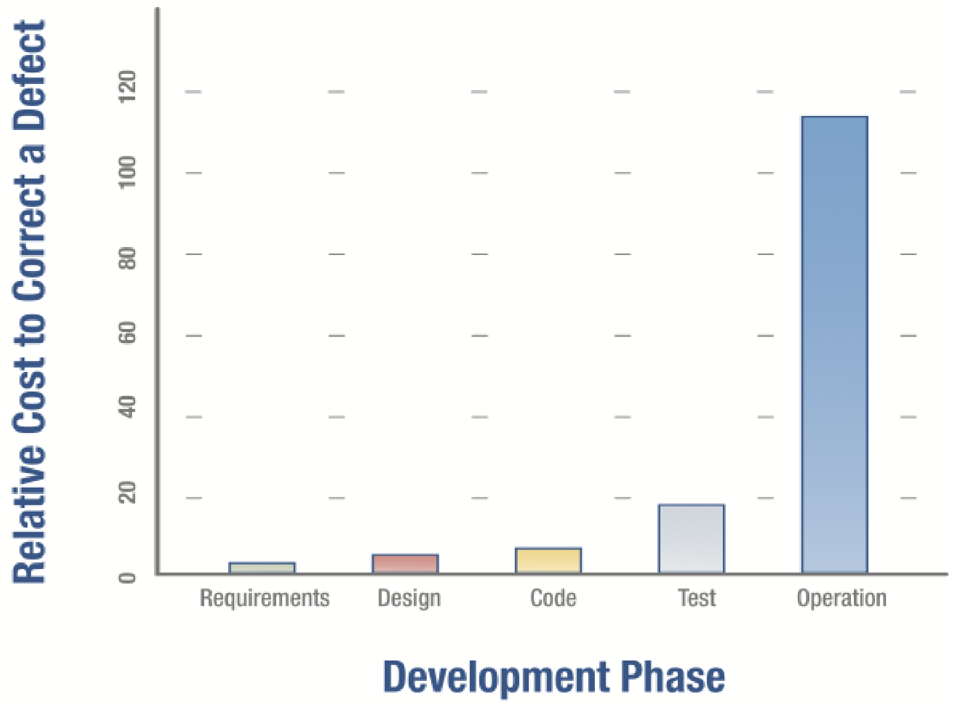


Figure 4.3 Relative cost to correct a requirement defect depending on when it is discovered. [Serena RTM]

4.3 The requirements engineering process

Requirements Engineering (RE) is a systematic and structured process comprising activities for discovering, documenting and maintaining a set of requirements in the form of a *requirements document* [Kotonya & Sommerville, 2004]. It aims to help in knowing what to build before system development starts in order to prevent costly rework [Paetsch *et al.*, 2003]. The Requirements Engineering discipline is divided into the consequent phases of *Elicitation*, *Analysis*, *Documentation*, *Validation* and *Management*. These phases are typically carried out one-by-one in a linear fashion, however, a certain phase may also be revisited if alterations are needed.

Few organisations have an explicitly-defined RE process. There is not a single process which is right for all organisations, either, nor is the process transferable between them. The process should take into account the type of systems developed, organisational culture, and the level of experience and capabilities of the people involved. Most of the existing standards regarding RE only relate to how the output of the process, the resulting requirements document, is structured. [Kotonya & Sommerville, 2004]

In the light of the aforementioned and the continuous debate on software development and requirements management processes, it is tempting to suggest that the industry practice in how requirements are elicited and managed likely varies greatly not only between companies but also between teams and projects.

4.3.1 Elicitation

Requirements elicitation is the process of obtaining the requirements of a system. This happens through gathering information from stakeholders, documentation and external sources. Its ultimate goal is to understand the users' needs and constraints [Pohl *et al.*, 2005]. While Young [2004] mentions that over 40 elicitation techniques exist, some of the commonly used include [Paetsch *et al.*, 2003; Kotonya & Sommerville, 2004]:

- *Brainstorming.* First as many ideas as possible are generated, after which they are discussed, revised and finally reduced to the ones that are considered most useful by the group. For prioritizing the ideas, various voting techniques may be used.
- *Document analysis.* Requirements are extracted from written documentation such as user manuals.

- *Interviews* are in-person meetings with business analyst asking questions to get information from the stakeholder. These can be closed or open. Interviews yield a rich collection of information, with the disadvantage that the amount of qualitative data may be hard to analyze. Different stakeholders can also provide conflicting information.
- *Focus groups.* Small group of people of various backgrounds discuss the features of a prototype of the system developed. This approach helps to identify user needs and perceptions, and spontaneous ideas are often brought out. Observations are a good complement to focus groups.
- *Observation.* Users are watched performing their daily tasks and they are asked questions about those tasks and their work. Observation may be either direct with the investigator present, or indirect, using e.g. recorded video. By using this approach, the issue of stakeholders describing idealized or oversimplified work processes may be overcome.
- *Prototyping.* Requirements are validated against dedicated prototypes or partially finished versions of the software.
- *Scenarios.* The system is discussed together with a customer in a natural language to discover typical usage scenarios.
- *Use cases.* Existing use cases are analyzed and discussed.
- *Workshops* are meetings where many stakeholders are gathered for an intensive and focused get-together. Brainstorming may be applied.

Hunt and Thomas [1999] describe the elicitation process according to their experience: "Requirements rarely lie on the surface. Normally, they're buried deep beneath layers of assumptions, misconceptions, and politics." They suggest elicitating requirements is hard work: "don't gather requirements – dig for them."

4.3.2 Analysis

Requirements analysis is a phase in which requirements engineers and stakeholders negotiate on the details of the requirements to be included in the requirements document. This phase is closely linked to to the elicitation process, and in practice the two are interleaved. The elicitation-analysis-negotiation may be illustrated as a spiral (Figure 4.4). [Kotonya & Sommerville, 2004]

Common methods of requirement analysis are [Paetsch *et al.*, 2003]:

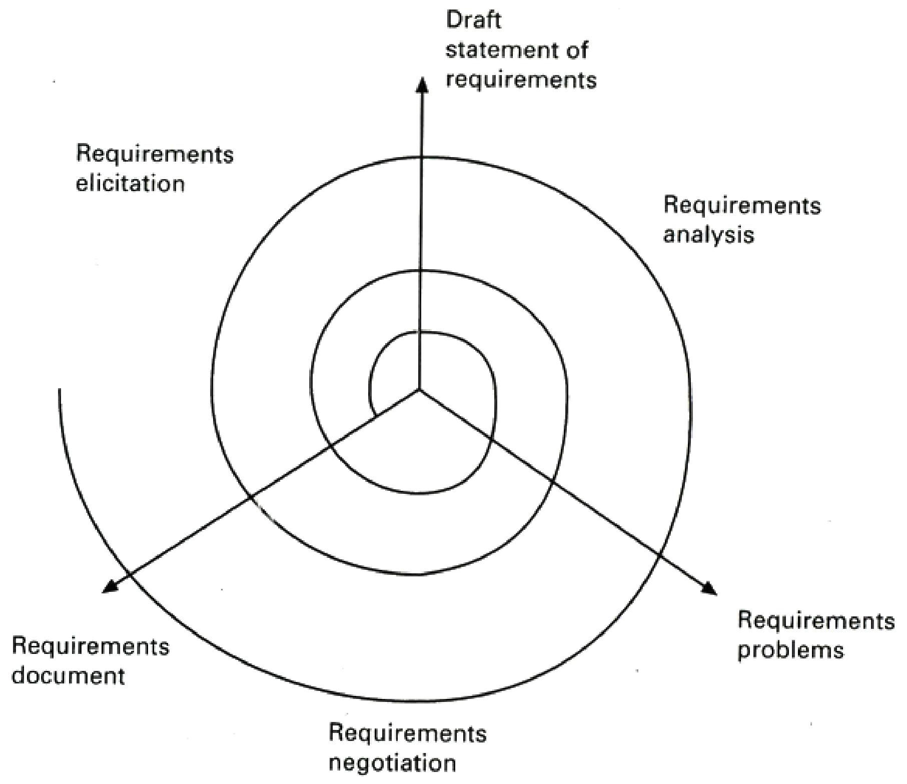


Figure 4.4 The elicitation, analysis and negotiation spiral [Kotonya & Sommerville, 2004, p. 58]

- *Joint Application Development (JAD)* is a facilitated workshop session in which developers and customers of different backgrounds discuss desired product features. The JAD is designed to define a project on various detail levels, design a solution, and monitor the project until its completion.
- *Requirements Prioritization* is used to set priorities early in a project to help to decide which features can be skipped when under tight schedule and limited resources. Optimally, input is given by both customer and developers. Customers prioritize tasks given by their benefits, and developers point out the technical risks, costs and difficulties.
- *Modeling* is used to bridge the gap between the analysis and the design processes. Different modeling techniques are used to describe system requirements.

Other requirements analysis methods used include interviews, questionnaires, observation, procedure analysis and document survey.

As with requirements elicitation, Hunt and Thomas [1999] comment on the complexity of requirements analysis:

> "How can you recognize a true requirement while you're digging through all the surrounding dirt? (...) very few requirements are as clear-cut [as a statement of something that needs to be accomplished], and that's what makes requirements analysis complex."

4.3.3 Documentation

Requirements documentation is the practice of writing down the gathered requirements in an accepted document format. Documentation might be done using a word processor or a using a specific requirements management tool.

Use cases are a commonly used technique to capture the functional requirements of a system by using a scenario-driven approach. They describe what happens when *actors* (people or other systems) interact with a system to achieve a desired goal [Cockburn, 2001]. Use cases are not system features by themselves, but are rather used to describe how a system will behave in reaction to a given input. A use case may relate to one or more features, and a feature may also exist in one or more use cases.

Use cases should:

- hold functional requirements in an easy to read format
- describe what the system shall do for the actor to achieve a particular goal
- be at an appropriate level of detail
- not specify user interface design
- not specify implementation detail, but rather the intent.

Use cases can be created using a number of different ways and notations. While some use case templates exist as bases for textual use case documents, there are no standardized ways of creating use cases. Use cases may also vary in scope, being either *business* or *system* use cases, as well as in their degree of detail. Cockburn [2001] grouped use cases in three based on their level of detail: *brief*, *casual* and *fully dressed*.

The idea of use cases dates back to the work by Ivar Jacobson in 1986. Since then, the idea has been widely adopted and contributed to by a number of people. Use cases are generally considered most usable in describing interaction-based requirements. On the other hand, features they have been criticized for include their questionable clarity (usefulness of use cases depends on the writer), no standard definition exists of how a use case should be like; and the fact that people seem often to be confused by the level of detail in which the user interface should be described in use cases. As Hunt and Thomas [1999] put it: "Unfortunately, Jacobson's book was a little vague on details, so there are now many different opinions on what a use case should be."

Nevertheless, the lack of details in the original specification has not made the approach any less popular. Although a number of models and new modeling approaches have been derived from the original solution, use cases are still present in many software development approaches in one form or another.

4.3.4 Validation

Requirements validation aims to validate the gathered requirements for consistency, completeness and accuracy. This is essentially a precondition for the product built with the created requirement set to fulfill the customer's wishes and expectations. A typical method for requirements validation is a *requirements review*, in which a group of people, after reading and analysing the requirements, meet to discuss observed problems and agree on resolving them.

Conventional requirements validation methods include requirements reviews and creation of requirement test cases.

In requirements reviews, a group of people is gathered to read and analyze the requirements and look for problems. The found problems are then discussed, and actions for addressing them agreed upon: requirements may be rewritten to be more exact, missing information requested from stakeholders, conflicts resolved by negotiation and unrealistic requirements deleted or modified.

In requirements testing, a series of tests is created to match the functional requirements. Ideally, the behavior of each system feature should be tested. Each test case should define a specific area of functionality to test against, and might define the output the system would give in response to a given input.

4.3.5 Management

Requirements management is the process of managing change to a system's requirements. Incompleteness or incorrectness of requirements may lead to severe problems and is often considered an option worse to not having a requirements document at all. It has been presented [Kotonya & Sommerville, 2004] that more than 50% of a system's requirements will be modified before it is put into service.

Essential for requirement change management is the *traceability* of requirements: who suggested the requirement and why, what requirements relate to it and which external information is related.

Managing change is the domain in which agile software development methodologies are considered to excel. The reason for this is the basis laid in the second principle of the Agile Manifesto, where it reads: *"Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."* In agile approaches, change is considered natural and requirements are assumed volatile from the very beginning.

4.4 Three dimensions of requirements engineering

Pohl [1994] studied the RE process. He divided it into three orthogonal dimensions: *specification*, *representation*, and *agreement* (Figure 4.5). This framework may be used to present the shift from *initial input* to *desired output*: from personal to common views, opaque to complete system specification and informal to formal representation.

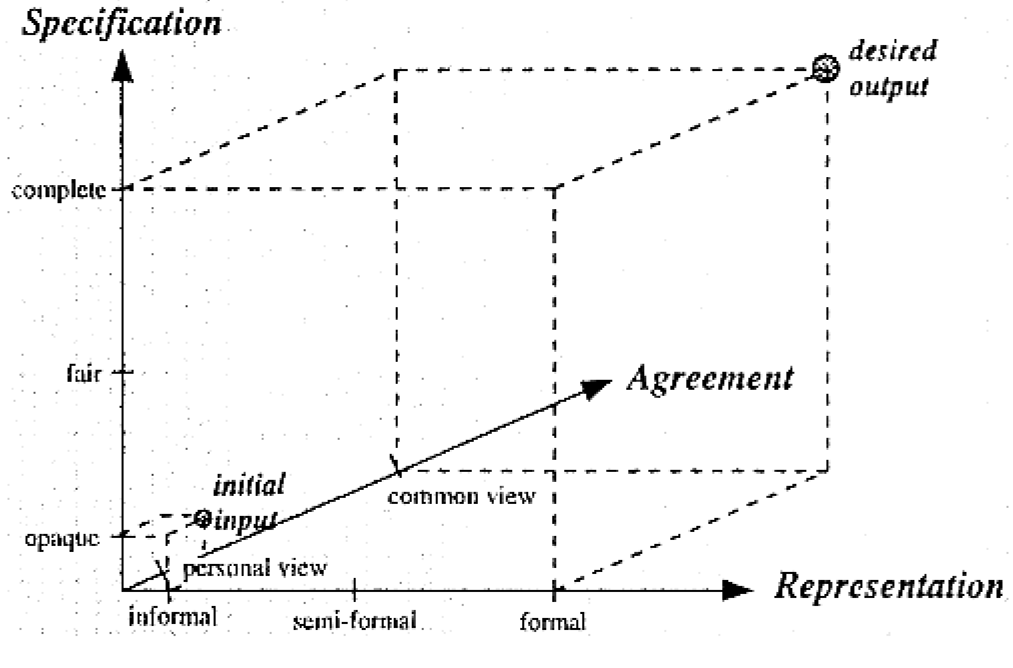


Figure 4.5 The three dimensions of Requirements Engineering [Pohl, 1994]

The main concerns of the *specification dimension* are completeness and understandability. This dimension relates to how well requirements are understood at a given stage.

The *representation dimension* relates to the formality and expression of requirements. When moving from informal to formal presentation of requirements, the types of requirement documents shift from user-oriented documents to system-oriented ones. Pohl thinks formality is important for people to be able to interpret the requirements in the same way; using an informal representation such as natural language might leave too much space for interpretation.

The *agreement dimension* focuses on the degree of the agreement the specification leads to. Here a shift from individual views to common agreement is desired. Pohl claims that it is beneficial to have various different views in that it supports the evolution to a common agreement on the final specification.

4.5 Lightweightness

Even with many existing techniques to ensure the quality of requirements, poor-quality requirements are reportedly still common in many software development projects [Boehm & Papaccio, 1988; Standish, 1994; Standish, 2009]. In the light of agile principles, it is enticing to think this is because of that these techniques are considered cumbersome, too complex, difficult to adapt or simply not being fun or rewarding enough to execute. Furthermore, what is often forgotten when specifying and modeling methodologies is the element of motivation among developers. Even if a process looks formally complete on paper, or specifically because it does, it is likely to attract opposition a result-oriented environment.

Hunt and Thomas [1999] suggest an approach of "simplest statement that accurately reflects the business need" be used, an idea which goes well hand-in-hand with one of the ideological cornerstones of the Extreme Programming (XP), namely doing "the simplest thing that could possibly work" [Wells, 2010; Venners, 2004]. It was in fact as early as the late 1980s that [Boehm & Papaccio, 1988] suggested avoiding a document-oriented approach such as the waterfall model to help in reducing software fixing and rework costs. Quite surprisingly, while they originally encouraged the use of prototyping approaches, many of the findings presented in the paper from an era today considered to be very formal and process-oriented are in line with the fundamental agile principles.

The demands presented for requirements management tools by Hoffmann et al. [2004] sound quite intimidating: "Since the requirements for complex systems are themselves complex information structures that must be handled in complex process scenarios, there are many strong requirements concerning a tool for managing them." While this claim suggests that not only that software projects are by nature complex but also that the only way to manage them is by the use of highly delicate and process-oriented tools, we believe in quite the opposite. I present that in order to make a practice attracting enough for it to be deployed and applied in practice, it is required for it to display direct enough benefits. This is in line with the idea of minimizing waste; all the parts of the process that do not directly contribute to the end cause should be omitted.

There is obvious demand for a model that makes the disciplines of RE more simple and less painful. Such a model of requirements documentation could be called *lightweight*: a model that is simple and powerful in that it does all that is expected; nothing less, nothing more. Minimizing complexity by striving for lightweightness both in terms of processes and tools will both enhance communication but also make work more rewarding and yield results in shorter time spans.

Zhang et al. [2010] applied the concept of lightweightness to the dimensions of the RE process presented by Pohl [1994]. The ideas presented focus on improving software quality rather than the quality of individual documents.

Lightweight specification is concerned with specifying requirements iteratively and incrementally throughout the project lifecycle. Requirements will be developed adaptively rather than predictably, as change cannot be avoided. Requirements should also be defined as simply as possible. This could be called *just-on-time specification.* Requirements need not be fully specified up front, when many aspects are still unknown and needs cannot be expressed consistently and correctly. Instead, they can be elaborated at the right time when they are selected for implementation. This approach is natural, as change is likely to emerge both in the business domain as well as in the customers' understanding of their own needs.

Lightweight representation embraces the concept of attaching requirements to working software. Instead of the degree of formality of requirements representation, the realization of the requirements may be demonstrated to the users at an implementation level. This may be done by using either actual, concrete software, or by the use of prototypes. Having such a context-enriched representation with implementation details enables a smooth transformation from the high level requirements description to the detailed implementation, enhancing the clarity and understandability of user requirements. This can further encourage active stakeholder participation and frequent feedback cycles.

Lightweight agreement emphasizes consensus among stakeholders. It is important to facilitate the evolution from the personal views to a common agreement on the final specification. A common agreement may be achieved by a number of means; besides the traditional requirements analysis, negotiation, and reviewing, the most important being the agile principle of providing feedback on small releases of working software. Contrasted to requirements expressed in a textual format, requirements placed in their actual context provide users with a better explanation of their contents.

4.6 Mapping the RE lifecycle to agile

RE as a process is not something that is commonly considered inclusive of agile software development methodologies. In fact, the situation is quite the opposite. The fact that the RE processes are well specified and thoroughly documented is often seen as a contrast to the agile ways of working, where one tries to avoid traditional heavyweight documentation in favor of doing more by documenting less.

Even though the importance of bringing simplicity and manageability is seldom questioned, the common practice indicates it remains a challenging task to combine the text-book way of working with the actual agile industry practices.

While this approach may seem natural considering the principles presented by the Agile Manifesto, it also presents a factor of uncertainty in software projects, as the existence and importance of requirements cannot be overlooked in any project. Regardless of the development model being used, requirements specify the functions and desired outcomes of the project under development. Failing to conduct the requirements process may directly or indirectly lead to discontent of stakeholders and prolongation of project schedule and budget.

Given the phases of RE, a mapping to common agile disciplines can be generated by the following classification:

Elicitation

- **Customer feedback throughout the project lifecycle.** As the elicitation of requirements requires close collaboration and involves various stakeholders, it is very natural for agile development approaches to continuously elicit and "dig for requirements". Frequent communication and close collaboration with the customer are considered one of the key aspects of agile development, which makes gathering and revising knowledge a seamless part of the process.

- **Use cases/scenarios and user stories.** Writing scenario-based descriptions of the goals of a user is a simple yet powerful approach of eliciting requirements. Bang [2007] reported success having used this technique to model a legacy system to be re-created from scratch, having some 50 user stories identified, specified and prioritized (which he estimated would suffice for two upcoming releases) in roughly three man-weeks.

According to Lowe [2001], the problem with conventional requirements processes are that they are largely designed to elicit requirements rather than to support the development of domain understanding or help to understand the impacts of a particular design. In agile approaches, this is at least partly mitigated by the fact that the elicitation, analysis and documentation phases are interleaved and feedback is constant and frequent.

In addition to use cases, *user stories* are also commonly used to capture and record system requirements. A user story is a requirement formulated in one or more short sentences in natural language, describing the activity a certain user wishes to achieve. User stories are typically associated with agile development efforts, as they are less verbose and simpler to manage than use cases, a subset of which they might be considered. The difference between a use case and user story is in fact frequently debated. Cockburn [2010] gave the following answer to the subject: *"A user story is the title of one scenario, whereas a use case is the contents of multiple scenarios."*

A common template used in formalizing a user story is as follows [Cohn, 2004]:

```
As a <role>
I want <feature>
so that <business value>
```

An example user story formalized using such template could be *"As an email user, I can search my archived emails for file attachments."*.

User stories resemble much the scenario approach of conventional RE elicitation. Bang [2007] reported positive experiences in applying user stories in an agile web development project. In his opinion, they are easy for people to understand without having to learn a new language. He also considered it a positive thing that the stories are kept simple and are not cluttered with lots of details that might contain an amount of uncertainty. On the other hand, he presented that when requirements are based on high-level descriptions, it is crucial to have the same key people available throughout the project. Otherwise, the same discussions might be had over again, which leads to waste of time.

Acceptance testing in Behavior Driven Development (Section 3.2.1) is typically done against user stories. The features estimated in playing Planning Poker, an estimation technique popularized by the Extreme Programming (XP) methodology, are also commonly in the form of user stories.

Analysis

- **Iterative planning**. The agile principle of doing frequent deliveries calls for elaborating on selected requirements before a new iteration. These meetings, such as the sprint planning meeting in Scrum, combine various elements in the RE domain. As requirements, often in form of backlog items or the like, are discussed, picked for inclusion and prioritized, requirements are practically *elicited* and *analyzed* simultaneously. The idea of iterative planning is comparable to the conventional RE phase of workshopping, but without being bound to a certain boundary or project phase.

  Seasoned software developers commonly seem to share a view on the attributes of good requirements. A good rule of thumb is that requirements should describe the need of the user, without going into the actual implementation details. Hunt and Thomas [1999] see that requirements should not be too detailed, because "abstractions live longer than details". Thus, a requirements document being too specific is considered a big risk. This claim is backed up by Boehm [1988], who claims that during the specification phase, extra functionality gets added to the list of requirements too easily, without a good understanding of how product's conceptual integrity and the required effort of the project are affected. They refer to this as the "Words are cheap" problem, leading to so-called *gold-plating* of products.
- **Iteration and prioritization.** As change is embraced and even late changes are welcomed, assessment of priorities is required frequently. A component is rarely considered finished, but may rather be refined and improved over the course of time. This makes the practice of *requirements analysis* constant and continuous.

Documentation

- **Working software.** Agile methods emphasize *working software over comprehensive documentation* [Beck *et al.*, 2001]. Having a working up-to-date version of the application deployed and readily available for stakeholders to use and comment encourages collaboration and lowers communication barriers. Following Boehm's [2000] IKIWISI (I'll Know It When I See It) principle, the application is best described by the application itself.

Validation

- **Small frequent releases.** Delivering frequent releases establish checkpoints upon which the requirements may be reflected and validated. Small releases lead to incremental builds of working software.

Change Management

- **Feedback on working software.** As with *documentation*, change is best illustrated on a recent installation of working software. Requirements are validated and the mutual knowledge updated frequently throughout the lifecycle of the project. Whereas prototyping is employed in the conventional RE elicitation phase and may still be used in agile projects, more weight is put on testing on actual, working software.

The conventional RE workflow is mostly linear with the phases following each other. Previous sequences may be iterated over when required. When the phases are mapped in an agile context, however, the sequence blurs. As the elicitation, change management and analysis phases are overlapped and often happen in parallel, it is no more possible to tell where an activity ends and another begins.

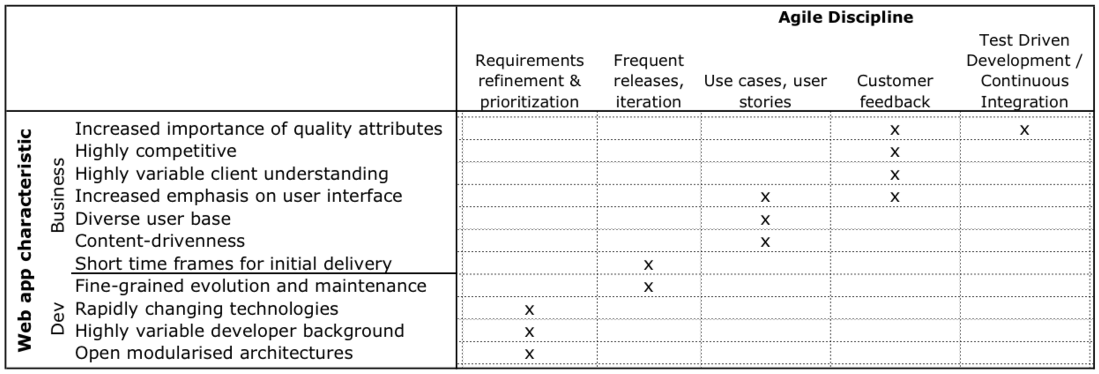
As an example, in a Scrum-style sprint planning meeting new requirements might be elicited, analyzed, documented and prioritized all in the same session. Furthermore, a mixture of the conventional RE techniques might be applied in a smaller scope within a single meeting or a workshop: brainstorming, document analysis and prototyping or UI design might all be practiced within the same session. As priorities change and new requirements are introduced, change management is also being done. Rather than chronological stages of action, in an agile context the RE disciplines become *cross-cutting aspects* being practiced throughout the lifecycle of a project.

5 Combining agile, RE and web

Fools ignore complexity; pragmatists suffer it; experts avoid it; geniuses remove it.
– Alan Perlis

5.1 Agile RE practices applied to web

The special characteristics of web applications indicate a demand for development disciplines providing support for these features. Table 5.1 presents a matrix mapping the special characteristics of web applications (as presented in Section 3.3) to common agile disciplines.



			Agile Discipline				
			Requirements refinement & prioritization	Frequent releases, iteration	Use cases, user stories	Customer feedback	Test Driven Development / Continuous Integration
Web app characteristic	Business	Increased importance of quality attributes				x	x
		Highly competitive				x	
		Highly variable client understanding				x	
		Increased emphasis on user interface			x	x	
		Diverse user base			x		
		Content-drivenness			x		
		Short time frames for initial delivery		x			
	Dev	Fine-grained evolution and maintenance		x			
		Rapidly changing technologies	x				
		Highly variable developer background	x				
		Open modularised architectures	x				

Table 5.1: Web app characteristics mapped to Agile Disciplines

The matrix is intentionally suggestive: the division of agile disciplines in five different groups is artificial, as is the rough categorization of web application characteristics between *development* and *business* ones. The motivation for the matrix is to indicate a mapping clearly exists between various web application specifics and common agile disciplines and practices. Even though the rows and columns are mapped in a *one-to-one* basis between the most likely matches, the reality is more likely to follow a *many-to-many* mapping, as most agile disciplines are applicable and contribute to more than just one area of a project.

One can see that a match exists in the group of agile disciplines for each of the challenges introduced by the intricacies of web applications. Thus, one may safely assume that the commonly known agile disciplines do in nature provide support for web application development context.

Continuous refinement and prioritization of requirements makes it possible to work in small feature sets and always provide the customer with new functionality that best answers the actual needs. Splitting requirements in small enough tasks makes it possible to make use of different kinds of components and technologies, as well as makes it possible to assign different tasks to different people based on where their specialities and capabilities are.

The agile way of *making frequent deliveries and continuously iterating* makes it possible to do a quick initial delivery, and also makes it possible to do fine-grained evolution.

Use cases and user stories are based on actual use scenarios and relate to the content the user is searching for in the web site or application. This approach also gives hints on which are the most frequently used functions in the interface. Visuality issues must naturally be taken care of separately.

Frequent customer feedback is what makes it possible to swiftly react to possible quality issues and make corrective maneuvers. It also gives an edge over competitors, as long as the business plan of the customer is sound. By being able to provide feedback on actual working software, the dangers of highly variable customer understanding leading to misconceptions and unrealistic expectations is mitigated. As visuality is commonly important to customers, it may also be reacted to by timely customer feedback.

Test driven development and continuous integration are solid techniques for maintaining quality, regardless of the process models being used.

Web application development, as opposed to more conventional software development, generally seems to further stress the importance of communication. The matrix presented in Figure 5.1 supports this idea. Practically all the agile disciplines revolve around communication in one way or another: interaction with stakeholders, meetings within the team, communicating requirements by use cases or user stories, communicating changes by frequent releases, communicating work order by prioritization and sprint planning, etc.

5.2 From documentation to communication

We set out to build on the combination of ideas adopted from agile disciplines and those of a conventional requirements engineering process presented above, extending them to cover the special characteristics of software development as practiced in web application development. Inspired by and building on the characteristics of the web context, we present a shift from **documentation** to **communication**, moving:

- from *fragmented* to *up-to-date*
- from *context-unaware* to *context-aware*

As well as added challenges, the web context also brings features that can be taken advantage of regarding requirements management. The nature of a client-server architecture brings along unique characteristics that affect development, deployment and availability. This is a particularly valuable asset to use in enhancing collaboration and communication. The main concepts of this model are built on the very characteristics of web applications.

The concept of embracing communication over documentation is backed up by Ambler's [2009] idea of active stakeholder participation.

Up-to-date

"Web-based distribution also avoids the typical two-inch-thick binder entitled Requirements Analysis that no one ever reads and that becomes outdates the instant ink hits paper. If it's on the Web, the programmers may even read it." [Hunt & Thomas, 1999]

Static documents are difficult to edit, update and share. It is a common problem in software projects to have requirements scattered around in different document versions, possessed by different people and even across different systems. There might not even be or a single authority maintaining and distributing the document, in which case the "peer-to-peer" nature of the process may lead to great differences in how different people perceive the issues and goals of the project. It goes without saying that in the long run, this is a likely cause of many quality and budget issues. (Requirements as project success factors are discussed in Section 4.1.1).

It makes sense to have one trusted system in which the requirements are stored and where they are maintained. Storing the requirements online in a commonly accessible place not only makes versioning and distributing them easier, but also contributes to achieving a shared understanding between the people involved in the development effort.

Due to the fact that there are typically not only a single but instead many different tools and systems employed in a software development project, it is unrealistic to require all of the documentation and requirements reside in one place (even if it would be an optimistic and ideal solution from many viewpoints).

Context-aware

Context-awareness reduces the risks of over-specifying by making the actual application visible at all times. Specification and implementation are both visible at the same place, at the same time. This makes the process more transparent and easily accessible to various parties.

This feature embraces the characteristic of web, as a centralized installation of a web application can be made available even during development as opposed to other kinds of software. Compared to a development project where seeing recent updates and changes may require first getting a copy of the latest binary, then installing it on a desktop or a mobile device, and then running it, the web is an ideal platform for context-aware requirements documentation and development.

These web-specific dimensions complement the work of Zhang et al. [2010], who applied the concept of lightweightness to the dimensions of RE, as presented in Section 4.5.

The shift originally presented by Pohl [1994] in *representation dimension* from informal to formal is comparable to the agile principle of preferring working software over comprehensive documentation. This is also the case with the *specification dimension*, where complete documentation does not provide value in itself, but in its contribution to the understanding and implementation of the system. The completeness of the specification is secondary to how the requirements can be reflected in the actual software. As far as the understandability of requirements is concerned, we present that there is hardly a better way to communicate a requirement than to show it in parallel with its implementation, e.g. in its actual context.

The common view in the *agreement dimension* is achieved by consensus made possible by the up-to-dateness of the requirements and the availability of the working software. When the requirements are always up-to-date and presented in the context of the working software, it helps to remove barriers in communication between the development team and the customers. This idea is well aligned with Lowe's [2001] theses on considerations regarding a web application development process. A context-aware environment supports the design process, as the business requirements and the actual product are brought closer together.

5.3 Building a conceptual model

A key element in implementing a more lightweight requirements process is to achieve a paradigm shift replacing the documentation with user stories, working software and changing requests, and in general shifting from documentation to communication efforts. [Zhang *et al.*, 2010; Arvela *et al.*, 2008] The requirements document will become less important on its own and gain more importance as the facilitator of a mutual understanding among the stakeholders about the goals of the development effort.

We have earlier proposed [Zhang *et al.*, 2010] a framework supporting analysis and documentation of requirements in agile projects. Figure 5.1 illustrates a reference model built on this approach. The entity-relationship notation captures *requirements*, *external documents*, *software artifacts* and *traceability links* connecting them.

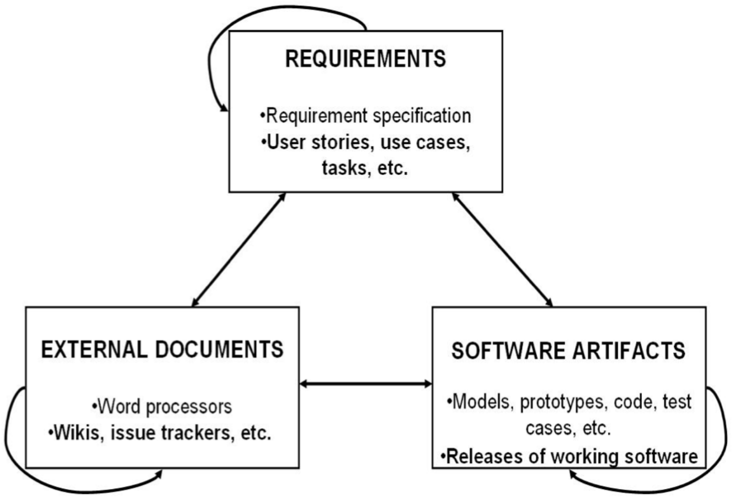

REQUIREMENTS
•Requirement specification
•User stories, use cases, tasks, etc.
EXTERNAL DOCUMENTS
•Word processors
•Wikis, issue trackers, etc.
SOFTWARE ARTIFACTS
•Models, prototypes, code, test cases, etc.
•Releases of working software

Figure 5.1 A reference model of a requirements analysis and documentation environment [Zhang *et al.*, 2010]

Requirements can be captured at different detail levels. Interacting with customers may lead to requirements of higher abstraction level, for example user stories. This model encourages an agile "just-in-time" approach to eliciting requirements. Preferably, requirements are refined and adjusted into detailed tasks as they are selected for inclusion in the next iteration. They are then implemented and evaluated within the same iteration.

Requirements should also cover the informal representations of users' conception of the system. This data may be stored as user stories (e.g. in the users' own words), use cases or scenarios that can be attached to the working software. This will make the requirements documentation process intuitive and encourage customer participation.

Requirements documents, prototypes and working software should consistently represent the actual needs of the customers. This approach provides a holistic view in which both the implementation details and their linked requirements are included. The authors present this could have a notable positive impact on timely evaluation and feedback.

External documents are the documents not stored in any requirements management tools. These documents are typically easy to create, but their static nature makes them hard for different stakeholders to collaborate on. There is a clear demand for a more straightforward approach in the lightweight requirements documentation.

The model proposes enhancing collaboration by adding *generic collaborative platforms* such as wikis and issue trackers. These can provide a way to communicate more flexible than possible with the use of separately exchanged static documents. The possibility of discussion and exchange of ideas happening in the context by various stakeholders can effectively reduce the details needed in requirements documentation, as the contextual information can be linked to the discussion on the collaborative platform. These platforms can adapt to support stakeholder participation in requirements elicitation and documentation.

Software artifacts are the by-products produced during the development of the software. They may include versions of working software, test cases or source code. They connect to the requirements through various traceability links.

A typical example of where software artifacts are born is the agile discipline of *frequent releases*, which results to distributing and/or deploying a version of the software developed for the customer to see. As the customers gets to see the current development process, the contextual information is synchronised between the stakeholders. This makes it possible for the developers to get constant feedback, which is an effective means of removing uncertainty.

As working software is in its actual context compared to a prototype where various compromises might have been made, risks of misunderstand are smaller. It has also been presented that this approach saves time, compared to throw-away prototypes that may be iterated but still get discarded in the end.

Traceability links connect the various instances of components. In doing so, they provide contextual information on the target system. They make it possible to tell for example which high-level user story a group of low-level tasks originate from, as well as which conversation, user story or initial requirement a line a source code file relates to.

As traceability links are likely to exist both within and between tools, it is important that an agile project has integration between various systems, e.g. in form of hyperlinks to reduce the fragmentation of information. The amount of manual effort required to maintain the traceability links and related information is often deemed a problem, as developers are unwilling to contribute time for that.

5.4 Overview of existing tools

In the industry there are countless large development projects in which large numbers of requirements are managed. Many different requirements management tools (*RMT*) have become available in the past years with varying approaches and features. Hoffman et al. [2004] present that the choice of such a tool is an "important and sensitive" issue which should be made carefully in the beginning of the project, as changing the tools in the middle of a project is a tedious and expensive effort.

In smaller companies, or in general projects with smaller scope, the desired model of working is usually in contrast very lightweight and hierarchy of the project team is intentionally kept as low as possible. However, one should note that agile methodologies and lightweight approaches have reportedly been successfully adopted in a number of large companies as well. Examples include Nokia [Laanti, 2009; Aalto, 2008], F-Secure [Palomäki, 2007], Philips [Abrahamsson, 2007] and more.

Today, the scope and extent of tools related to the handling of requirements has grown beyond that of just the traditional RMTs. Zhang et al. [2010] present the following categorization for requirements management and documentation tools:

- General-purpose document tools,
- collaborative tools,

- RMTs, and
- prototyping tools.

The tools available differ in many aspects: scalability, orientation, process or workflow support, business domain (system or software engineering), user interface, and the like. The following descriptions of the aforementioned groups of tools are as laid out by Zhang et al. [2010].

Characteristic to the *general-purpose document tools* is their widespreadness and ease of use. This category comprises tools commonly found in office suites such as Word, Excel and PowerPoint in Microsoft Office[1]. As these tools are not too specialized, they may be used for a number of documentation tasks. Even if not very sophisticated, many surveys [Hofmann & Lehner, 2001; Nikula *et al.*, 2000; Pohl, 1994; Manninen & Berki, 2004; Matulevičius, 2004] have found these tools most helpful in requirements documentation practices. The downside to using these tools is their shortcomings in supporting specific RE activities and in ensuring the quality of the created documents.

The *collaborative tools* provide a platform for assisting a group of users reach a common goal, usually by allowing collaborative editing of content. The most famous example of such systems is Wikipedia, an encyclopedia built by voluntary individuals from all over the world. These tools may be classified by their level of achieved collaboration, ranging from simple information sharing applications such as online chats and wikis to sophisticated groupware systems (e.g. knowledge management and project management systems). Rather than facilitating documentation, these tools are a lightweight solution for creating, editing, sharing and discussing information, while the latter improve the communication and collaboration for requirements analysis. In an agile context, collaborative tools such as wikis may be used widely for many project management tasks; recording notes and decisions from planning and status meetings, writing specifications, storing external documents such as graphics, spreadsheets, and text. Accompanied by extensions and plugins, it might be possible e.g. to do user interface sketches and model diagrams within the same program.

RMTs are the "heavyweight" tools providing an environment to support the different dimensions of the RE process. They are dedicated to managing large amounts of data collected during the RE process, as well as the volatility of the requirements [Kotonya & Sommerville, 2004]. These tools typically collect

[1] `http://office.microsoft.com`

the requirements in a database and provide a range of manipulation and access facilities for browsing, converting, and tracing requirements, as well as controlling change and generating reports. Those applications with support for the formal representation of requirements can also be used for checking and verifying their consistency and semantics. RMTs typically require comprehensive knowledge about the underlying processes. While empirical studies [Matulevičius, 2004] indicate that RMTs provide better support for the RE process and the quality of requirements documentation, there are many surveys [Hofmann & Lehner, 2001; Nikula *et al.*, 2000; Pohl, 1994; Manninen & Berki, 2004; Matulevičius, 2004] suggesting that the mainstream practice relies mostly on more lightweight tools, such as office suites and modelling tools, rather than dedicated RMTs. Furthermore, according to Zhang et al. [2010] survey reports have tended to contradict the use and the rationale of the ever rising number of dedicated RMTs. Widely used RMTs include tools such as CaliberRM[2], and IBM's[3] DOORS, RRC, and Requisite Pro.

Traditionally many RMTs have been built to support a specific process or workflow, often the desired case in organizations having a carefully specified and documented process model to be followed. Sometimes these tools have over the years grown to the extent where they attempt to cover many different aspects of a software project (Figure 5.2). While this approach may work for some projects, it in its feature-richness illustrates the demand for more lightweight alternatives.

Prototyping tools are used to rapidly build a prototype representing the important aspects of a software system. The prototype is used to demonstrate requirements to the stakeholders and collect feedback. These tools range from software used to create simple mock-ups - sometimes intentionally giving an unfinished look to emphasize the system being a prototype and nothing more - to very sophisticated ones to create and mimic actual interactive functionality and detailed user interfaces of the software system under development. Applications such as Axure RP[4], iRise[5], and ProtoShare[6] are used to create web-based prototypes. Also some general-purpose tools provide prototyping support for user interfaces and web design: examples include graphics design tools such as Illus-

---

[2] `www.borland.com/us/products/caliber`

[3] `http://www-01.ibm.com/software/awdtools/`

[4] `http://www.axure.com/`

[5] `http://www.irise.com/`

[6] `http://www.protoshare.com/`

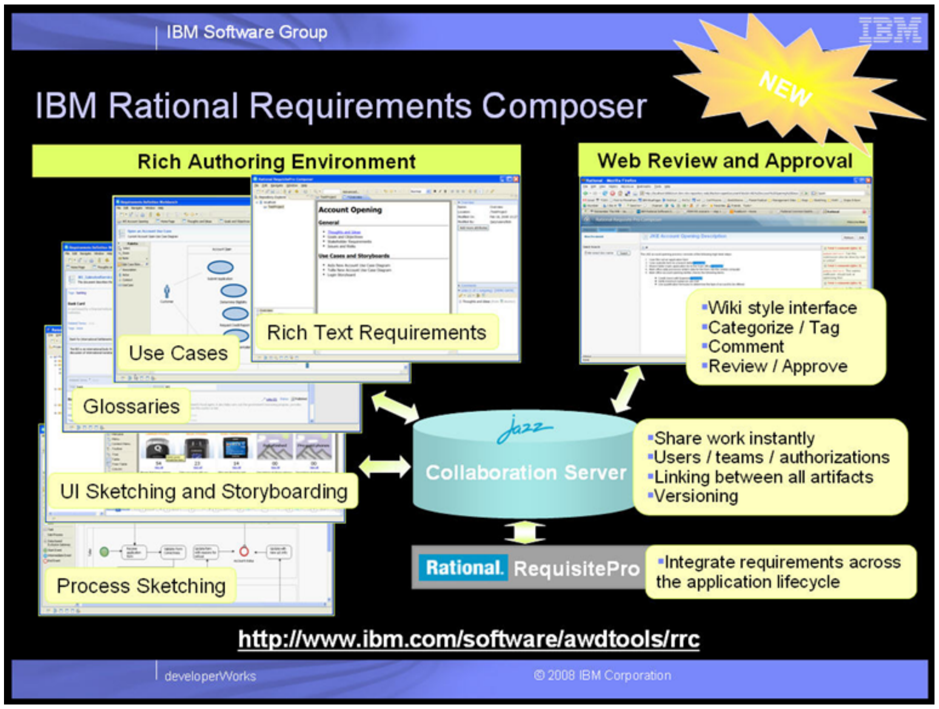


Figure 5.2 A presentation slide showing an overview of the features of IBM's Rational Requirements Composer[7]

trator and Photoshop, diagramming tools such as Visio or SmartDraw, and the visual and textual HTML tools such as FrontPage and DreamWeaver. Prototyping tools do not focus on specifying and managing requirements, but rather on providing stakeholders with a real experimental system. This approach effectively increases the understandability of requirements while avoiding requirements creep and rework.

In addition to the aforementioned four categories of tools for requirements documentation, there are also project management tools, such as Rally[8] and Scrumworks Pro[9], which allow editing a requirements backlog and generating reports. Requirements documentation tools in general provide support for requirements specification in various levels of formality. Also the collaborative tools provide

[7] `http://download.boulder.ibm.com/ibmdl/pub/software/dw/demos/RRC/RRCOverview.html`

[8] `http://www.rallydev.com/`

[9] `http://www.danube.com/scrumworks/pro`

more flexible support for external documents, whereas prototyping tools can provide links between the software artifacts and the requirements. [Zhang *et al.*, 2010]. However, none of them can cover the components specified in the reference model depicted in Figure 5.1.

The purpose of requirements documentation is communication among stakeholders. Although the general-purpose document tools have widespread availability, they lack adequate support for communication and collaboration in the RE process. Even if this deficiency of collaboration is compensated by the collaborative tools, these also still lack support for context-enriched representation and just-on-time requirements documentation. Also do the RMTs over-emphasize the rigid specification and the often bureaucratic workflow of the RE process. In doing this, communication between the developers and the customer suffers, as the close and continuous interaction is lost [Manninen & Berki, 2004]. The prototyping tools facilitates customer feedback by providing a system to test, but often comes short in not providing support for just-in-time specification. [Zhang *et al.*, 2010]

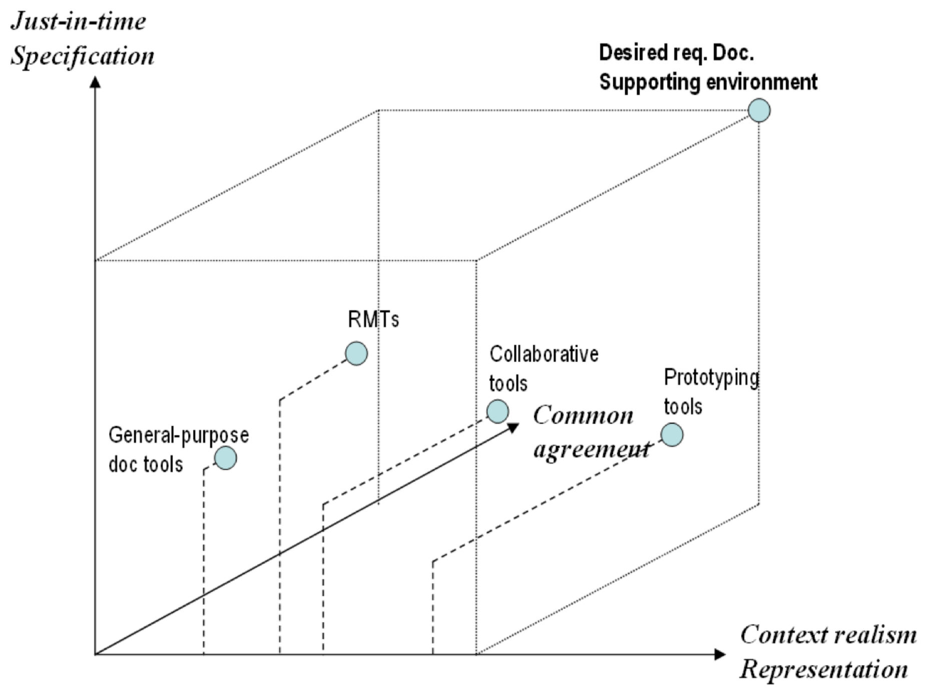

Just-in-time Specification
Desired req. Doc. Supporting environment
RMTs
Collaborative tools
Prototyping tools
Common agreement
General-purpose doc tools
Context realism Representation

Figure 5.3 Requirements tools laid out within the three dimensions of lightweight requirements documentation [Zhang *et al.*, 2010]

The support provided by these tools for the goals within the three dimensions of lightweight requirements documentation (as discussed in Section 4.5) is illustrated in Figure 5.3.

Additionally, the International Council for Systems Engineering (INCOSE), has since the 1990s gathered and maintained information on Requirements Management tools. They have created a suite of survey questions and keep in frequent touch with the tool vendors to compare and stay aware of the features of numerous different Requirements Management Tools. [INCOSE, 2010]

6 Vixtory

Many existing requirements management tools focus in the more traditional disciplines of requirements management in terms of process orientation and generating documentation artifacts and meta-data. In doing that, they are often document-oriented, possibly difficult to use and adopting them is sometimes considered a burden that slows down the project.

In this section we present a concept for a tool not only to complement the existing requirements management related tools, but also as a proof-of-concept of how the problem domain could be approached from a different angle. By no means is our purpose to claim superiority over the existing approaches, but rather provide a novel idea of annotating, recording and tracking requirements in the context of web development.

It is highly unlikely that any single combination of tools, let alone a single tool, would be a perfect - or even a decent - match for all people doing software development. On the contrary, we believe in the choosing the tools depending on the type of project, or the "right tools for the right job" approach. The importance of this approach is also stressed by Hoffmann [2004]; although, we would like to emphasize not relying on a single tool but instead combination of them for maximum efficience and comfort. Introducing a tool suite that is nearly a project management ecosystem in its own right (see the overview of IBM's RRC in Figure 5.2) leaves less space for individual developers to form their own toolbox of preferred software and may ultimately hinder productivity, even if originally built for the opposite in mind.

Although web application present a group of intricacies and special features of their own, they have little special demands as far as requirements are concerned. The core principles of the requirements management domain still apply, so where most difference compared to the existing requirements management tools may be made is in how the requirements are tied to their context and presented. As the creators of Vixtory, we want to emphasize that we do not consider Vixtory a one-tool-fits-all solution, but are rather excited by the possibilities of such an approach as we feel it has great potential but it is yet to be found practiced in the industry.

6.1 Background and concept

Vixtory was born by the initiative of Henri Sora, the CTO of a Finnish software company Ambientia Oy specializing in the development of customized web applications [Ambientia, 2010]. A number of requirements management and specification tools had been evaluated for use in web application development, but none of them seemed a good fit for the result-centric development process used. The tools available typically either focused on producing a requirements document or required building a *throw-away prototype*, which would usually be discarded when replaced by an actual implementation. As the goal was to be able to create the actual software right from the beginning, prototypes that would be discarded were considered unnecessary waste. Thus spun the idea for a novel tool, in form of the question: "what if requirements could be specified in the right context, the actual application in development?"

Vixtory is a requirements documentation tool designed specifically for web-based software development projects. It is a web application run on a server that allows users to attach requirements theoretically on any existing web site using a modern web browser. The idea in Vixtory is that the site being developed evolves as it normally would, and Vixtory is used to attach requirements "on top" without needing any changes to the product.

An important design decision was to make the tool simple and intuitive enough so that it could be used by developers as well by the customers. The very idea of the tool and the web as a targeted environment laid ground for conforming to the IKIWISI principle: "I'll Know It When I See It" [Boehm, 2000]. By being accessible and straightforward while presenting requirements in their actual context, it would facilitate collaboration and shared understanding among the shareholders contributing to the project.

6.2 Features

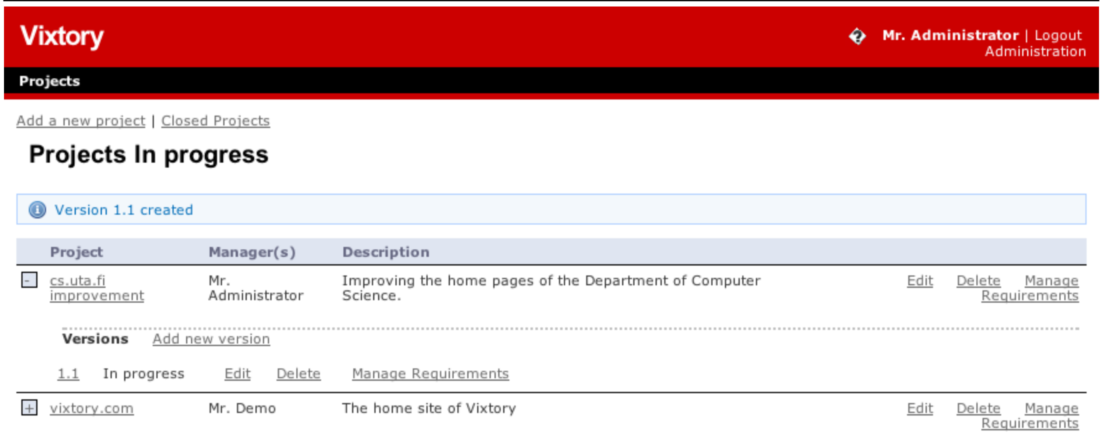
The feature set of Vixtory was intentionally kept to a reasonable minimum, allowing the development of the tool to concentrate on the concept. Features that were not considered relevant to testing the idea were dropped out. At the same time, it was desired not to enforce any particular workflow. The feature set became as follows:

- **Simple single-user authentication**. Only a single user role exists: all users have global access to all data stored in the system.

- **Transparently create and edit requirements directly into existing web sites**. Any web site having a publically accessible URL address will do. Requirements are simple items with title, description and creation date.
- **Allow prioritizing requirements**. A requirement can have a priority ranging from *trivial* (least important) to *blocker* (most important).
- **Manage change history with logical entities**. Each development *Project* consists of one or more *Versions* (installations) of the site/application.
- **Provide a quick overview of the requirements**. A listing displaying an overview of the requirements in the current version.

What differentiates Vixtory from many of the tools in the categories mentioned in Section 5.4 is that it doesn't directly fall into any category, but instead combines elements found in each of them.

Vixtory embraces the IKIWISI concept (I'll Know It When I See It) by applying the idea of context-awareness (as presented in section 5.3). This contributes to David Lowe's [2001] idea of developing a clearer understand of the relationship between business, information and technical architectures of the system, as the requirements are brought to the context of the actual web application.


Vixtory
Mr. Administrator | Logout
Administration
Projects
Add a new project | Closed Projects

Projects In progress

Version 1.1 created

	Project	Manager(s)	Description			
-	cs.uta.fi improvement	Mr. Administrator	Improving the home pages of the Department of Computer Science.	Edit	Delete	Manage Requirements
+	vixtory.com	Mr. Demo	The home site of Vixtory	Edit	Delete	Manage Requirements

Versions — Add new version

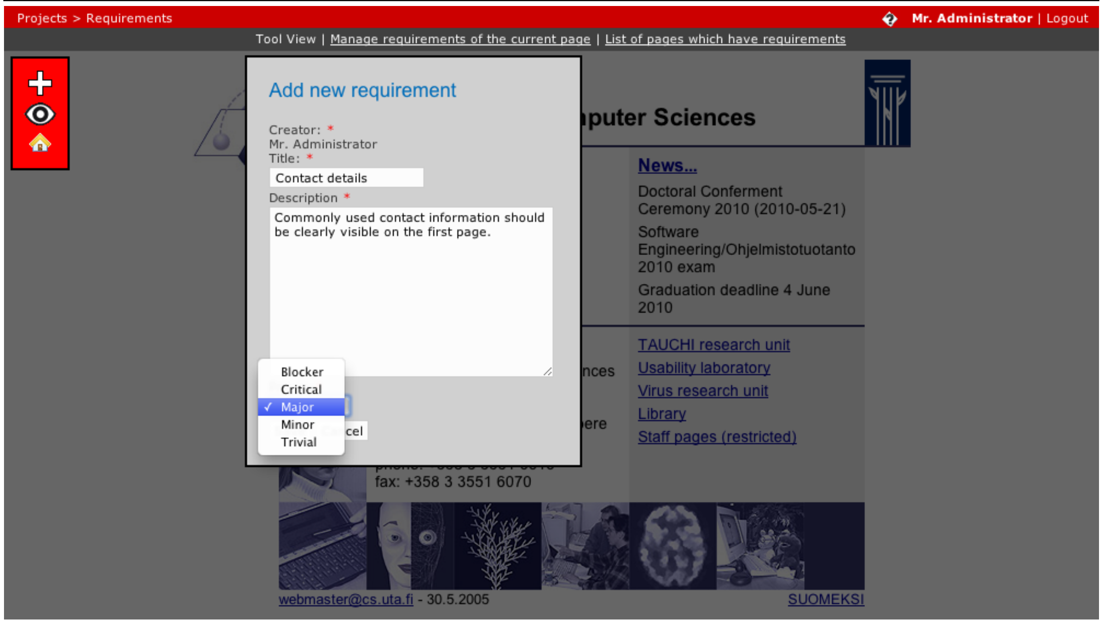
1.1	In progress	Edit	Delete	Manage Requirements

Figure 6.1 Vixtory: the projects view

6.3 Usage

The workflow of Vixtory is based around *projects* which correspond to the developed web applications. A project consists of one or more *versions*, each of which is an actual instance or a development version of a site, identified by a URL.

When a user logs on, the *Projects view* showing an overview of projects is presented (Figure 6.1). In this view projects and their versions can be added, edited and removed. Projects are divided in two simple states; *In progress* or *Closed*.

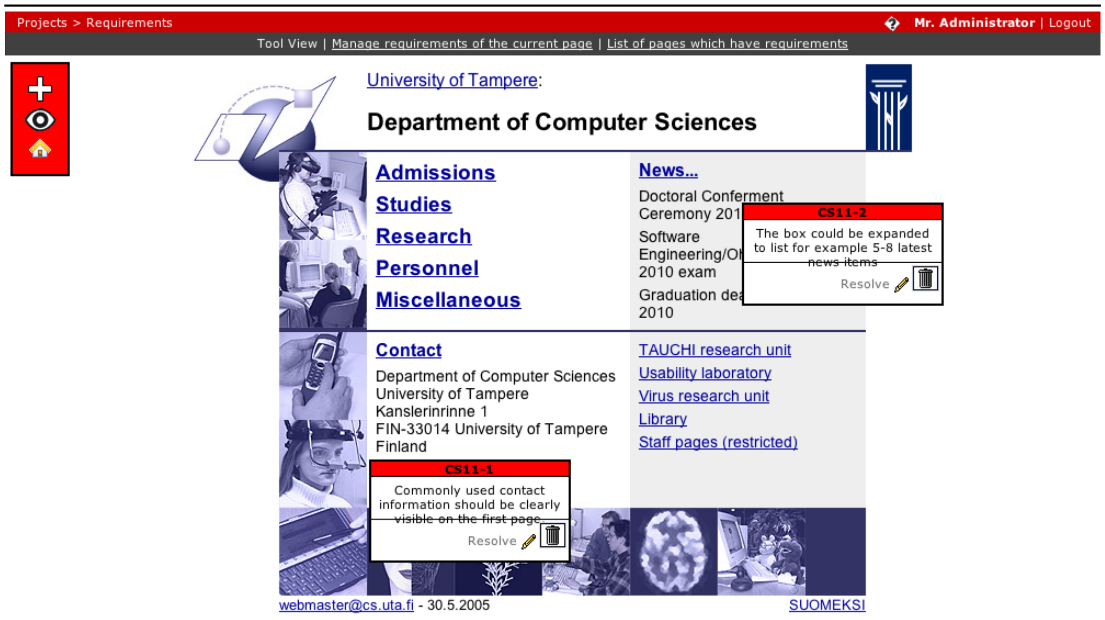

Projects > Requirements
Mr. Administrator | Logout
Tool View | Manage requirements of the current page | List of pages which have requirements
Add new requirement
Blocker
Critical
✓ Major
Minor
Trivial
cel
puter Sciences
News...
Doctoral Conferment Ceremony 2010 (2010-05-21)
Software Engineering/Ohjelmistotuotanto 2010 exam
Graduation deadline 4 June 2010
TAUCHI research unit
Usability laboratory
Virus research unit
Library
Staff pages (restricted)
nces
ere
fax: +358 3 3551 6070
webmaster@cs.uta.fi - 30.5.2005
SUOMEKSI

Creator: *
Mr. Administrator
Title: *
Contact details
Description *
Commonly used contact information should be clearly visible on the first page.

Figure 6.2 Vixtory: adding a new requirement

By selecting a version to work on (possibly first creating one, if required), the user is presented with the *Requirements View* (Figures 6.2 and 6.3). This is the actual working display in Vixtory. Most of the screen estate in this view is occupied by the view of the actual application. When this view is first loaded, the user is presented with with the contents of the URL defined for the Version. Here the only static user interface components are the Vixtory top bar and the floating toolbar with three functions, from top to bottom: *Add a requirement*, *Show/hide requirements* and *Return to Projects view*. The toolbar may be dragged to a different position on the screen.

While in this view, the user can browse the embedded web site as he would normally. Vixtory will load the specified URL and update the current contents of the view with those of the link clicked. By clicking on the "plus" icon on the toolbar, Vixtory enters a requirement adding mode and the screen will darken. By clicking on any point in the view, a dialog for adding a new requirement is



Projects > Requirements

Mr. Administrator | Logout

Tool View | Manage requirements of the current page | List of pages which have requirements

University of Tampere:

Department of Computer Sciences

Admissions
Studies
Research
Personnel
Miscellaneous

News...

Doctoral Conferment Ceremony 201

Software Engineering/O 2010 exam

Graduation dea 2010

CS11-2

The box could be expanded to list for example 5-8 latest news items

Resolve

Contact

Department of Computer Sciences
University of Tampere
Kanslerinrinne 1
FIN-33014 University of Tampere
Finland

TAUCHI research unit
Usability laboratory
Virus research unit
Library
Staff pages (restricted)

CS11-1

Commonly used contact information should be clearly visible on the first page

Resolve

webmaster@cs.uta.fi - 30.5.2005

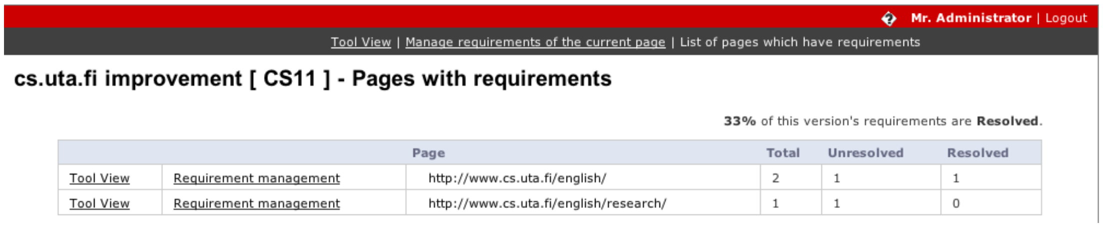
SUOMEKSI

Figure 6.3 Vixtory: Requirements view with two requirements on a page

presented (Figure 6.2). Here the requirement is assigned a title, a description and a priority. Once selected *Save*, an annotation rectangle indicating the added requirement will appear in the selected point. Once created, requirements can be edited, moved to another position by dragging and dropping, marked as *resolved* or removed. This *context-enriched* presentation of requirements makes them easier to understand and provides a feasible ground for discussion and further development.

As the user navigates from page to another, the view will always be updated to display the requirements added for the current location. Figure 6.3 shows a web page with two annotated requirements. As the requirements are saved directly to the database, it is possible for multiple users to browse the same version on different computers and add new requirements simultaneously. Venners [2004] presented that having a central repository in which all the requirements are stored increases the chances of a successful project. This concept is nicely supported in Vixtory with it embracing the idea of *up-to-dateness* presented in Section 5.2, allowing anyone with access to the system browse the most recent version of the stored information. Not requiring any other tools than a web browser found practically in all contemporary computers is a noticeable asset, considering the availability of data to different stakeholders.

The user may select to display an overview of pages with requirements (Figure 6.4). This view presents a simple overview of pages with the number of resolved requirements in each. There is also a convenience link for directly displaying the location in the requirements view.



Mr. Administrator | Logout

Tool View | Manage requirements of the current page | List of pages which have requirements

cs.uta.fi improvement [CS11] - Pages with requirements

33% of this version's requirements are **Resolved**.

		Page	Total	Unresolved	Resolved
Tool View	Requirement management	http://www.cs.uta.fi/english/	2	1	1
Tool View	Requirement management	http://www.cs.uta.fi/english/research/	1	1	0

Figure 6.4 Vixtory: Overview of pages with requirements

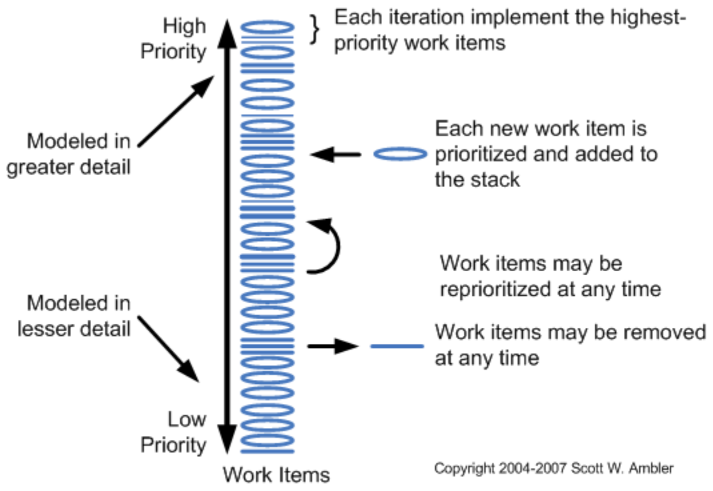

High Priority
Each iteration implement the highest-priority work items
Modeled in greater detail
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Modeled in lesser detail
Work items may be removed at any time
Low Priority
Work Items
Copyright 2004-2007 Scott W. Ambler

Figure 6.5 Requirements as a prioritized stack [Ambler, 2009]

Vixtory supports the idea presented by Ambler [2009] to treat requirements like a prioritized stack (Figure 6.5). In this approach, the items with a higher priority are modeled in greater detail ("Just-in-time"). Vixtory makes adopting this way of working possible by making it easy to attach new requirement items on the application and allowing to effortlessly edit and re-prioritize them at any time. This incremental specification of requirements allows for greater flexibility on focusing on the task at hand, instead of attempting to specify everything up front. Venners et al. [2004] emphasize that according to their research, completing the requirements during the project is more important than having all of them readily specified up front at the beginning of the project.

6.4 Development

Vixtory was developed as a student project on the annual Project Work course at the University of Tampere, Department of Computer Sciences [Poranen & Kajaste, 2008; Poranen, 2009]. It was developed from the scratch over the course of two semesters. During this time, a total of 18 people were involved in the creation.

The original project had two main goals. The primary objective was to create a *proof-of-concept* of how requirements could be managed in a lightweight way in web development projects. The secondary objective was to evaluate the technology, especially the Grails web framework [Grails, 2010], to find out how easy would it be for novice developers to adopt a completely new technology stack and web framework.

6.5 Known issues and technical considerations

There are a number of quirks and limitations related to the technical implementation of Vixtory.

A security scheme called *Same origin policy* [Mozilla, 2009] is applied in all modern web browsers. This scheme restricts a page loaded in the browser from manipulating the Document Object Model (DOM) structure of pages that reside in a different location. As this would effectively prevent the very idea of annotating requirements directly on an external web site (one with a publicly accessible URL), the security restriction was worked around with a proxy mechanism. The local proxy fetches the requested content from the external site as needed, allowing it to be manipulated locally. In practice, this isn't visible to the user at all.

One of the most notable shortcomings in the technical implementation relates to the level at which JavaScript is supported. Currently, Vixtory is practically not applicable to rich user interfaces implemented with JavaScript and AJAX (see Figure 2.2). Properly handling them would require delicate technical planning, for example in terms of determining the state of an application at a given moment, avoiding JavaScript namespace conflicts with the scripts loaded locally and on the remote site, hooking onto script call chains to be able to handle UI logic, etc. While it would be pessimistic to say it cannot be done, it definitely would not be easy.

A lesser, yet a fundamental shortcoming is the session handling and form submissions. Artificially maintaining user sessions is challenging to get done right, although still possible. Posting form data could cause a target page to be rendered differently depending on the parameters.

The requirements specified in various views are currently unique in that they only exist in the view in which they where created. It would make sense to be able to share requirements between various views, or specify so-called "global requirements".

On the other hand, the same page content might be displayed in multiple URLs. As the views are identified by their URL, Vixtory is currently not able to recognize the same view, if it is accessible with more than one address.

7 Evaluating Vixtory

7.1 Conducting research

Originally, the intention was to introduce and evaluate Vixtory in real-life software development projects. Targeted were web development projects on the annual Project Work course at the Department of Computer Science at the University of Tampere; the same course on which Vixtory was originally developed. The purpose was to gather first-hand experience of Vixtory and its possibilities in actual group development efforts. However, Vixtory was still suffering from the symptoms following a major rewrite and was not considered mature and stable enough for production use. Thus, the original plan unfortunately had to be abandoned.

As a secondary approach, Vixtory was included as a group work topic in the Requirements Engineering course in the autumn semester of 2009. Two student groups of four people chose to study Vixtory. The groups were given a presentation of the background, development, and current status of Vixtory to help them get started with the project. An installation of Vixtory was also made available for their evaluation. Both the groups were asked a total of six questions in three groups:

1. Features of a requirements tool for agile software development projects
   - What are needed features for a requirements tool that supports agile development projects?
   - Does Vixtory support the RE process (Elicitation, Analysis, Documentation, Validation)? How?
2. Usage scenarios of Vixtory
   - What kind of usage scenarios or projects do you think Vixtory suits best/worst?
   - How could different stakeholders use Vixtory together to improve their communication and collaboration?
3. Vixtory improvement
   - What would you improve or make differently?
   - Should Vixtory allow for some kind of social interaction (commenting requirements)?

The current technical shortcomings and other relevant considerations were made clear to the students in the initial briefing. They were also instructed to try not to concentrate on technical issues but rather to evaluate the concept with its pros and cons. The full assignment given to the students is presented in Appendix A.

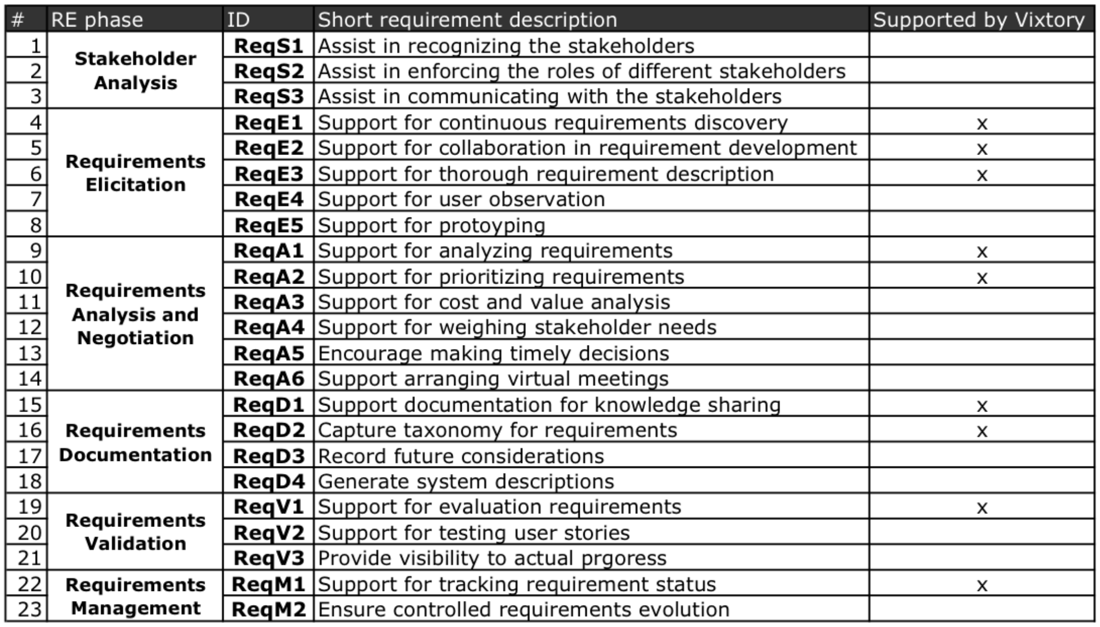
One of the groups [Karjalainen *et al.*, 2009] produced a rather comprehensive document analyzing the pros and cons of Vixtory. The other group more or less overlooked the given assignment and ended up sidetracks, mostly comparing the current feature set of Vixtory with other existing RE tools. Although some good observations were made, the discussion about the concept of Vixtory did not answer the provided questions well enough to be covered in further details.

7.2 Interpretation of results

Karjalainen et al. [2009] noted that traditional RE approaches and agile models share a common ground in their endeavors for customer satisfaction. They make valuable observations in work done by Paetsch et al. [2003] and Kernighan [2008], noticing an "anti-tool attitude" present in the agile world. While Paetsch et al. presented a shift in focus in agile methods from tools to emphasizing the importance of having highly skilled people available, Kernighan praised all-powerful simple tools that can be used for a wide range of purposes instead of a single one, contrasting the use of such tools to opening a paint can with a screwdriver. The point made by Kernighan about a "mechanical advantage" is very valid. He presents that a tool must do some task better than people can, augmenting or replacing our own efforts. Being able to do that the tool justifies its own existence. That is also the case with Vixtory, which was originally created for a need.

Based on the phases of the Requirements Engineering practice, the group identified a total of 23 requirements for "a perfect agile RE tool". These are presented in Table 7.1. The table is based on similar table created by the group, with an added column describing if each requirement is considered provided a solution to by Vixtory, based on the group's opinions and findings. Although one might ask if a tool implementing all the features presented in this table would allow for an agile way of working, it still illustrates well the problem domain, particularly the aspect of which features can be omitted in a requirements documentation tool while still maintaining usability.

Eliciting the requirements for such "perfect agile RE tool", the group iterated through the phases of RE, first identifying the requirements found for a tool in each of them, and then contrasting them to the features of Vixtory. We next



#	RE phase	ID	Short requirement description	Supported by Vixtory
1	**Stakeholder Analysis**	**ReqS1**	Assist in recognizing the stakeholders	
2		**ReqS2**	Assist in enforcing the roles of different stakeholders	
3		**ReqS3**	Assist in communicating with the stakeholders	
4	**Requirements Elicitation**	**ReqE1**	Support for continuous requirements discovery	x
5		**ReqE2**	Support for collaboration in requirement development	x
6		**ReqE3**	Support for thorough requirement description	x
7		**ReqE4**	Support for user observation	
8		**ReqE5**	Support for protoyping	
9	**Requirements Analysis and Negotiation**	**ReqA1**	Support for analyzing requirements	x
10		**ReqA2**	Support for prioritizing requirements	x
11		**ReqA3**	Support for cost and value analysis	
12		**ReqA4**	Support for weighing stakeholder needs	
13		**ReqA5**	Encourage making timely decisions	
14		**ReqA6**	Support arranging virtual meetings	
15	**Requirements Documentation**	**ReqD1**	Support documentation for knowledge sharing	x
16		**ReqD2**	Capture taxonomy for requirements	x
17		**ReqD3**	Record future considerations	
18		**ReqD4**	Generate system descriptions	
19	**Requirements Validation**	**ReqV1**	Support for evaluation requirements	x
20		**ReqV2**	Support for testing user stories	
21		**ReqV3**	Provide visibility to actual prgoress	
22	**Requirements Management**	**ReqM1**	Support for tracking requirement status	x
23		**ReqM2**	Ensure controlled requirements evolution	

Table 7.1: Table of requirements for the "perfect agile RE tool" and if they are met by Vixtory, based on work by Karjalainen et al. [2009]

discuss the most interesting of these aspects from the viewpoint of Vixtory.

Stakeholder analysis

Regarding the practice of stakeholder identification, the group questions the agile idea of customer collaboration; agile models could at least in theory ignore other potential stakeholders, leading to incompleteness of requirements. The same issue was noted by Paetsch et al. [2003], who referred to this as the agile methods often assuming an "ideal" customer representative. This is analogous to the "happy path" of use cases - a scenario where everything goes as planned and no unexpected errors occur. In such a case, the customer representative employed in an agile project knows the answer to all the questions posed by the developers and also has the power to make binding decisions. RE, on the other hand, has a less idealized picture of stakeholder involvement.

However, as also Paetsch et al. noted the focus of RE being in identifying all the requirements *before* starting the project, this could also lead one to think as a mitigating factor; even if there would be requirements that would not have been discovered before starting the actual development work, they are likely to be encountered soon enough and embraced, as change is welcomed at all stages. Furthermore, one could argue that the importance of stakeholder analysis lies greatly in understanding the big picture of the project together with the interests and goals of the stakeholders. When looked at from this perspective, it would make sense to think that single requirements do not necessarily matter as long as the project team is constantly working with the ultimate goals of the project in mind. This also emphasizes the idea of having skilled individuals who are experts in knowing how to best accomplish the desired results. Naturally, whether it is feasible to put this much weight in the skills of individual developers, also depends on the project and company.

The group observes Vixtory's omittance of fine-grained user authorization and authentication. As described in Section 6.2, this was an intentional design decision, as in the development it was deemed more important to focus in the core features. The idea presented of being able to manage users and their roles to allow and disallow specific tasks (ReqS1, ReqS2) would definitely be a useful feature in the final product. For example, adding new requirements or changing their priorities could be restricted for certain users, some stakeholders might be given access to financial details of the project, etc.

Also presented is a feature to assist in communication between stakeholders (ReqS3). We believe that the context-awareness of Vixtory actually makes it possible to promote Vixtory as a communication channel between stakeholders of various interests, i.e. business and development people. Vixtory could be used as well in workshop-like meeting with people sitting together around a table in a conference room, as well as asynchronously with different people accessing it from the comfort of their own workspace. Seeing the requirements attached in their context allows everyone to understand how and where the requirements relate to the system under construction. Although NFRs are not explicitly supported, it is likely that they could be discovered and discussed alongside the use of Vixtory, and recorded e.g. by using external systems.

Requirements elicitation

The group presents that the due to the feature-drivenness of agile development projects, planning of non-functional requirements is often left unaddressed, and suspect that architecture and component interfaces may need to be restructured in performance-critical systems. While this might be true in some cases, practices such as test-driven development (Section 3.2.1) are all about designing and implementing interfaces-first. Also, although agile methods are rather result-oriented than document-oriented, it doesn't mean that documentation should be omitted, but rather to only document in detail the parts that are actually needed and critical for the system.

The basic way of annotating requirements in their context in Vixtory is seen by the group as very easy and straightforward. They also think browsing the existing web site while discussing future development provides a good starting point for elicitation and thus supports the principle of continuous requirement discovery (ReqE1). While the group presents that the provided template for requirement descriptions could be more versatile (ReqE1, ReqE3), we present that this issue is argumentative; the simpler the approach, the more freedom it allows for. The idea is decent overall, but implementing it feasibly without bringing too much complexity would require careful planning.

Per the report, support for collaboration in requirement development (ReqE2) is supported at the level of passing requirements from clients to developers, in which Vixtory also improves the mutual understanding among the parties. However, what the group presents should be added is an ability to chat, or better, arrange video conferences using Vixtory. This is an intriguing scenario but indeed software for screen sharing purposes already exists in various tools such as NetViewer[1], and duplicating such functionality would not be feasible. The same applies as far as a chat functionality is considered, and providing one in-application would not necessarily bring much additional value. Also proposed is way of linking requirements to user stories, which is indeed a nice idea, but the question of how user stories could be introduced in the system besides requirements, still remains open.

In support for user observation (ReqE4), a possibility of recording the navigation history of the user is presented. This would be fairly straightforward to implement, but it remains unclear whether this data could actually be used to reap benefits, as the case of collaboratively navigating the site with Vixtory is probably unlikely to resemble a real use case, unless the person browsing the is given a test-case like task to conduct.

The group made an interesting proposition of providing support for prototyping (ReqE5). They point out that experimenting with the web site and trying out new things is currently not possible. They present Vixtory could allow for adding so-called "activity notes" which would be used to change the way certain buttons

[1] `http://www.netviewer.com/en/`

behave or how the web site navigated. This could be complemented with the possibility of adding new temporary web pages to the project. The ideas presented could actually work rather well, provided that the difference between already existing and "stubbed" pages would be obvious enough to the user. However, this function could in part be answered to by allowing traceability links to external systems which could be used for prototyping or planning upcoming functionality.

Requirements analysis

The group suggests that Vixtory in general provides little support for requirements analysis and negotiation (ReqA1). Handling of the requirement notes is easy, but in their view Vixtory lacks in the support for prioritizing the requirements, or analyzing or negotiating the features with the stakeholders. Having the requirement notes flowing on top of the web page improves the understanding of their contents, but no structures means exists for analyzing the requirements; one needs to browse through them to get an overview. Even though there are lists which allow browsing the names of the requirement notes, the context is blurred when their containing page is not shown at the same time. The group proposes that support for a checklist or interaction matrix be added, should Vixtory be decided to provide a more comprehensive support for analysis activities.

The means of requirements prioritization (ReqA2) in Vixtory is assigning a requirement an importance category. The group suggests the idea of further being able to manage Scrum-style requirements backlogs: a sprint backlog containing the requirements that are going to be implemented in the next iteration, and a product backlog containing a list of all the requirements in the system. This is actually a very nice idea, as it handles prioritization and a lightweight method of planning future development, and it would probably integrate well with the interface and workflow of Vixtory.

Practically, cost and value analysis (ReqA3) as such is not needed, as in agile projects requirements are prioritized continuously so that the most important features are developed first. This is illustrated in the requirements stack presented in Figure 6.5, where the most important requirements are always prioritized on the top of the stack. This makes the prioritization more straightforward and also leaves less space for interpretation. Much of the same applies to the support for weighing stakeholder needs (ReqA4). Agile methodologies often prefer an approach where a person in a role such as that of the *product owner* in Scrum is ultimately (from the viewpoint of the development team) responsible for including such requirements for implementation that the needs of all the stakeholders get represented. As such, it may be out of the core scope of Vixtory to go any deeper into product management details. Support for built-in calendar and scheduling functions (ReqA5) are also suggested.

While the idea presented for supporting virtual meetings and recording their decisions and outcomes (ReqA6) is an intruguing one, the same results could likely be achieved by the use of an external software and a flexible enough means of documenting the decisions, tracked backed into Vixtory by the use of traceability links.

Requirements documentation

It is presented that the RE tool should provide support for comprehensive documentation and knowledge sharing. We as the creators of Vixtory believe in a more straightforward "right tools for the right job" approach, trying to avoid creating a tool that tries to address all worlds. Hence, following the ideas presented earlier in our reference model (Figure 5.1), a wiki system could well provide the need for more extensive sharing of knowledge (ReqD1), leaving the RE tool to fulfill a more specific task in associating the requirements in their context, with additional/metadata stored in external systems. This also applies to the proposed case of recording future considerations (ReqD4). This could also provide a simple yet powerful remedy for the problem reported by Kajko-Mattson [2008] of agile methods leading to an oral burden. The importance of having an "up-to-date" way of storing documentation mentioned in Chapter 5.2 is also emphasized in here, as contrasted to traditional files lying around the hard drives and email inboxes of people, a centralized wiki installation is an effective means of communicating up-to-date information.

Vixtory is considered to have support for daily knowledge sharing (ReqD2), as all the requirement notes are stored and listed in two different views. However, the documentation format is light, and the report discusses the problem of "living presentation", where software engineers are in possession of so-called hidden knowledge, that is not recorded anywhere but is instead memorized in the heads of the people. They also pointed out studies in which this had been acknowledged as an actual risk. An idea of the system automatically generating system descriptions to mitigate this could well be applied in practice. Together with the traceability links provided in the reference model, it would probably be feasible to make Vixtory to allow linking to external documents (i.e. wiki pages) and also write e.g. some kind of changelog for all the views and requirements.

The group also suggest that it partially looks as if system requirements and agile work tasks have somehow gotten mixed up, but at the same time it describes the strive for simplicity of Vixtory pretty well; as the group themselves point out (ReqD2), Vixtory requirement notes would probably work best as a reminder about some task to be done. Here they also mention the possibility of linking to actual, more detailed requirement documentation. Again, the design decision is debatable, as the intention was not to enforce any specific documentation format or workflow. Thus, it would be sensible to allow linking to a number of different documentation systems and formats with as little effort as possible.

Curiously, the proposed idea of generating system descriptions out of the information contained in Vixtory (ReqD3) was actually something that was originally considered to be included in the first versions of Vixtory. Originally a "print" view existed that could be used to print out a document containing a small picture of each of the views together with a list of associated requirements. However, the techniques used to accomplish this feature made Vixtory dependent on Internet Explorer and was abandoned, as it hindered other development. Having such a feature would still be useful in some cases.

The idea of supporting future considerations (ReqD2) bears a great resemblance to the suggested prototyping support proposed as a part of requirements elicitation support (ReqE5); creating stubs of new pages would allow to place requirements and ideas on pages that do not still exist.

Requirements validation

The group proposes a set of commonly agreed acceptance tests be used to compensate for the missing requirement specification that could otherwise be used as a requirements validation baseline.

Vixtory supports viewing end evaluating individual requirements (ReqV1), but the available requirement documentation format is deemed as insufficient in detail, as in (ReqD2). It is also proposed that to prevent the views from cluttering as a result of lots of requirements being added, only the ones relevant to the current iteration/phase could be shown. This relates to the ideas presented about requirements prioritization (ReqA2).

Regarding the indication of project status and progress (ReqV2), the paper suggests that the progress of the static features on the developed application could be made more visible. An example of providing "before" and "after" views to the web site is given; in this approach, static screenshots could be taken of the sites each time a new version is created. The evolution and progress of the developed site would then be better illustrated, and the different versions compared with each other.

The group points out a problem in requirements validation such as that the developers may never really know if the customer has properly evaluated their work. They propose the RE tool could support testing user stories (ReqV3). This could once again integrate with the proposed backlog functionality, if implemented. The group proposes the possibility of creating and storing formal usage scenarios, from which user stories could be generated from.

Requirements management

Typical to agile projects, the group observe, is that the evolutionary path of a requirement is not recorded and thus cannot be followed. They propose the tool should provide support for continuous tracking of requirement status (ReqM1). Vixtory currently only classifies requirements as "unresolved" or "resolved", and the group present adding phases such as "identified", "evaluated", "committed" and "closed" to reflect requirements management practices and stages.

The group notes that in a volatile environment where time and market requirements change rapidly, it may be futile to practice change management in general, and back this claim up with research by Paetsch et al. [2003], who had reported that no proof has been shown that change tracking would provide any economic benefits. The group also suggests the tool should ensure controlled requirements evolution (ReqM2), which could probably be achieved by not only having the tool record change history but also by agreeing on common project practices, privileges and responsibilities. This is, however, once again a matter of communication and should not necessarily be restricted by the tool itself. In general, Vixtory is hugely permissible. In some cases at least, having available a changelog showing who changed what and when (or even with a revision history), could in a rather simple way bring enough control and security over tracking change - an approach comparable to that of a wiki - without forcing the users to familiarize themselves with various access control mechanisms.

The proposal of treating project versions as releases that are going to be delivered makes much sense. In this approach, progress could be calculated based on the requirement effort estimates, and by adding just a little metadata in the requirements, statistics such as burn-down charts could be added.

7.3 Summary

Overall, despite the lack of resources available for research and evaluation, the results were surprisingly good both in the terms of the quality of the work done by the group given the context, and also the feedback given to Vixtory by them.

Karjalainen et al. [2009] give Vixtory a rather good score even with its shortcomings: *"it is even surprisingly well adjusted to the target environment and it really fulfills its original purposes"*. They further proclaim in Kernighan's [2008] terms that Vixtory indeed provides a *mechnical advantage* for handling the requirements for web site development, contrasted to printing web page layouts on the wall and attaching Post-It notes on them. According to the group, Vixtory can be used by customers to define requirements and validate results at least on some level. It can also facilitate understanding of how the development should proceed. For developers, Vixtory provides means of better understanding of the requirements and a notice board for helping with the implementation.

Other pros include Vixtory being flexible to use and intuitive, with no features hidden behind menus, making learning to use it effortless. As it is usable with a normal web browser, it is also easily accessible.

On the downside, the group suggests, Vixtory might be an overkill for passing around simple task notes, and it could provide better support for commonly recognized agile processes as Scrum and XP. As a step towards this, they present it should be possible to manage the full life cycle of a requirement, from annotating a newly discovered feature to marking it released. They also see the prototyping features as something that Vixtory would clearly benefit from.

The current version of Vixtory already provides some support for the model presented in Section 5.3. The "requirement notes" of Vixtory are generic in kind in that they do not directly relate to (nor do they force) any specific model of requirement documentation. This greatly emphasizes the idea of providing traceability links to both external documents and software artifacts, as Vixtory does not try to handle everything by itself. Instead, it handles a rather specific task of annotating certain views of a system. Considering this, contrasting Vixtory with a traditional RMT would be rather foolish, as their approach is quite different from each other! While the other tries to cover every possible aspect concerning the lifecycle of requirements in a project, the other concentrates on a very specific domain, namely web application development, and also only gives a very limited feature set while not enforce any particular workflow.

Given the context of Vixtory, it looks like one of the most remarkable shortcomings in Vixtory is the lack of support for linking to external systems and documents, namely the *traceability links*. It would make sense for future development efforts to provide a better support for them. Providing a general-purpose tool with support for flexibly linking to other tools and documents would allow individual developers much freedom in choosing their preferred tools. This is already often the case in software development, where different people are likely to prefer different editors, integrated development environments (IDE) and modeling tools.

7.4 Features for an ideal tool

As in software development projects in general, the dangers of requirements creep are inherently present also in this case. Often when for pursuing perfection, the danger of trying too hard concretizes in an overwhelming feature list, which is also the case in here. Adding to the features to the system may not only prove a burden in the development efforts, but also end up causing confusion for the end user. Instead of adding new features, omitting requirements might be an equally strong and important design decision. Make no error, this is not an easy decision! Knowing what to include and exclude are probably the hardest parts in developing a new kind of tool. Being a tool developed specifically for web-based development, Vixtory might benefit from some web-specific tools and capabilities such as support for inspecting and manipulating HTML, CSS and Javascript. Doing that, however, Vixtory would probably shift focus a bit from being a general-purpose documentation tool to being a development tool. Here also lies a danger in that providing a very limited feature might be worse than not providing a feature at all, as having a tool that does not do what it's expected to might easily lead to a different tool being chosen for the task. Having more features will also mean more maintenance in the future.

Reflecting on the ideas and feedback given by the group, a listing of requirements for an "ideal web-based RE tool" based on the ideas presented by the group, and building on personal opinions on lightweightness, is presented below. The difference with the listing of the features for a "perfect agile tool" differs from the one presented by the group in Table 7.1 in that it mostly represents the personal views and experiences of the author and takes a somewhat more pragmatic approach than the one presented by the group, whose requirements were elicited from the list of existing RE practices. The following features are founded on the assumption that they could be built on the top of what Vixtory is today:

Attaching requirements on a web page on the fly. This is essentially the core and what Vixtory does already.

Managing various development projects and their versions. As above, this is provided by Vixtory. Additionally, more support could be provided for creating deliveries and releases.

Intuitive user interface. A simple and intuitive user interface that does not get on the way and contains all the most frequently used functionality accessible at all times.

Browser-based. As Vixtory, the system should be accessible with a standard web browser for ease of accessibility.

WYSIWYG. For embracing the IKIWISI principle [Boehm, 2000], the tool would need to display the web page exactly the way it is displayed normally. The user should not need to know the technical implementation details, but rather be able to browse the web application as he would do normally.

Filtering of requirements. Requirement notes could be filtered by various criteria so that the screen would not get too cluttered with all the requirements shown at all times. Filtering criteria could include things such as project status (only showing requirements relevant to the current iteration), priority (only show critical requirements), etc.

Support for traceability links. It should be easy to attach external documents and links in a project and to requirement notes. Naturally, it would also require support from the other end in order to make the links birectional.

Extendability via plugins. The system could incorporate a plugin architecture for users for whom the core features would not be enough. This could allow for adding custom document templates and providing the users with additional functionality, e.g. add other types of entities to attach to web site views than just the requirement notes currently supported by Vixtory.

Support for scrum-style sprint and product backlogs. The system could provide basic support for maintaining a backlog for both current iteration, as well as the big picture of the system being developed. These backlogs could also be used for prioritizing tasks by dragging-and-dropping and assist in making deliveries.

Real-time change support for seamless collaboration. It would enhance collaboration for multiple users to work on the same view (application state) at the same time. If a requirement would be displayed for two different users at the same time, moving the requirement on one screen would also show it moving on the other. This would also apply for all the other changes in the system.

Social features for enhancing collaboration. Work items such as requirement notes could be commented. The home screen could incorporate a dashboard showing a summary of what has been changed since the last login.

Support for global requirements. There could be "global requirements" that would apply for the whole application. In practice they might often be NFRs.

Multiple views per requirement. Requirements and views should be in many-to-many relation; a requirement could show up in multiple different views.

Support for dynamic functionality and changing web application states. Rather than views, the system could support the concept of a "state". When a user opens a menu in a web application, for example, the system might be in a different state than if the menu was not opened. This feature would be crucial, as most web applications of today provide desktop application like dynamic functionality.

Generating requirement documentation. Stored requirements content could be exported for example as a PDF file. This could be used for various purposes such as archiving, sharing or printing it out.

Flexible access control by user groups and roles. Users could be assigned various roles and privileges and their access in some projects or parts of them restricted.

Basic prototyping functionality. Stubs or placeholders for new views could be added. Prototyped or mocked stubs would be hilighted so that they would not get mixed up with existing, "actual" views.

Support for project status tracking. Projects could be assigned a status or it could be calculated based on the iteration and completeness status of existing requirements and tasks.

Illustrating differences between versions. Differences between project versions could be illustrated, and differences in versions could be compared side to side.

Maintaining a change history. The system could automatically write a wiki-style list of all the changes, and also provide a mechanism for reverting back to an earlier version, functionality comparable to a version control tool. This way, an accidental deletion of the requirements of a view could simply be reverted, or the change history of a requirement would be easily accessible.

8 Conclusions

We set out to discover whether it was possible to find a mapping between RE practices and common agile principles, and find out the features a tool for agile web project requirement documentation would benefit from. Essentially, both of these questions were found answers.

Considering the absence of contributions regarding agile web development in general, let alone studies combining the domains of agile, web, and RE, I am very satisfied drawing this thesis to a conclusion. I believe it might well be the first of its kind not only attempting to draw together these three fields, but also in presenting and analyzing a concept and ideas for a tool to assist in such ventures. I do acknowledge that it is unlikely to be able to directly apply the findings in the industry to reap immediate benefits. I would, however, consider the contribution significant, as the presented way of documenting requirements in Vixtory is a novel one with no tools offering similar functionality. Also I feel the linkage found between conventional methods of requirements engineering and agile development helped cross a gap that is commonly perceived to exist between the "traditional RE world" and that of agile development.

Even though the evaluation and research conducted on Vixtory was less thorough than originally planned for, it still seemed to support the authors' thoughts on the possibilities - as well as shortcomings - of Vixtory. The already existing list of ideas for the future development of the tool lays good ground to build on. The task would definitely not be an easy one, but I strongly believe in the possibilities of the concept, as even at this proof-of-concept stage Vixtory was received surprisingly well.

While the development of Vixtory itself is currently frozen, the conceptual model presented in Section 5.3 provides a solid ground for future work and expansion. Furthermore, there is obvious demand for a tool with a feature sent identical to that laid out in Section 7.4.

References

[Aalto, 2008] Juha-Markus Aalto. Large scale agile development of S60 product software: A few hundred synchronized Scrums - setup and experiences. *OO Days 2008 at the Tampere University Of Technology, Finland*, pages 1–7, Nov 2008. `http://www.cs.tut.fi/tapahtumat/olio2008/esitykset/aalto.pdf`.

[Abrahamsson, 2007] Pekka Abrahamsson. *Agile software development of mobile information systems.* Springer Berlin / Heidelberg, Jan 2007.

[Abrahamsson, 2008] Pekka Abrahamsson. End Of Agile. *Conference presentation, OO Days at the Technical University of Tampere, Finland*, pages 1–25, Dec 2008.

[Ambientia, 2010] Ambientia. Ambientia Oy. *Company Web Site*, 2010. `http://www.ambientia.net/portal/en`.

[Ambler, 2009] Scott Ambler. Agile Modeling. 2009. `http://www.agilemodeling.com/`.

[Arvela *et al.*, 2008] Mike Arvela, Matias Muhonen, Matias Piipari, Timo Poranen, & Zheying Zhang. AgileTool - Managing requirements in Agile WWW projects. *The 7th International Conference on Perspectives in Business Information Research 2008, Gdansk, Poland*, 2008.

[Bang, 2007] Tom J. Bang. An agile approach to requirement specification. In *Proceedings of 8th International XP Conference, Como, Italy, June 18-22, 2007*, 2007.

[Baskerville *et al.*, 2003] Richard Baskerville, Balasubramaniam Ramesh, Linda Levine, Jan Pries-Heje, & Sandra Slaughter. Is Internet-Speed Software Development Different? *IEEE Software*, 20(6):70–77, Feb 2003.

[Beck *et al.*, 2001] Kent Beck, James Grenning, Robert C Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Andrew Hunt, Steve Mellor, Hermann Josef Matula, Arie van Bennekum, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler, & Brian Marick. Manifesto for Agile Software Development. 2001. `http://agilemanifesto.org/`.

[Beck, 2002] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.

[Bethke, 2003] Erik Bethke. *Game Development and Production.* Wordware Publishing, Inc., 2003.

[Boehm & Papaccio, 1988] Barry W. Boehm & Philip N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10):1–16, Jan 1988.

[Boehm, 2000] Barry Boehm. Requirements that Handle IKIWISI, COTS, and Rapid Change. *Computer*, 33(7):99–102, 2000.

[Brooks, 1987] Frederick P. Jr. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[Chelimsky *et al.*, 2010] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, & Dan North. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends.* The Pragmatic Bookshelf, Jul 2010.

[Cockburn, 2001] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley Professional, 2001.

[Cockburn, 2010] Alistair Cockburn. A user story is the title of one scenario, whereas a use case is the contents of multiple scenarios. 2010. `http://alistair.cockburn.us/A+user+story+is+the+title+of+one+scenario+whereas+a+use+case+is+the+contents+of+multiple+scenarios`.

[Cohn, 2004] Mike Cohn. User Stories Applied: For Agile Software Development. 2004. `http://www.mountaingoatsoftware.com/books/2-user-stories-applied`.

[Dahl & Nygaard, 2008] Ole-Johan Dahl & Kristen Nygaard. How Object-Oriented Programming Started. 2008. `http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html`.

[Dominguez, 2009] Jorge Dominguez. The CHAOS Report 2009 on IT Project Failure. 2009. `http://www.pmhut.com/the-chaos-report-2009-on-it-project-failure`.

[Fowler, 2006] Martin Fowler. Continuous Integration. 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`.

[Fowler, 2008] Martin Fowler. Business-readable Domain Specific Language. Dec 2008. `http://www.martinfowler.com/bliki/BusinessReadableDSL.html`.

[Garrett, 2005] Jesse James Garrett. Ajax: A New Approach to Web Applications. 2005.

[Gat & Heintz, 2009] Israel Gat & John D. Heintz. John Heintz on the Lean & Kanban 2009 Conference. 2009. `http://theagileexecutive.com/tag/scrumban/`.

[Gorman, 2006] Jason Gorman. Post-Agilism – Beyond the Shock of the New. 2006. `http://www.itarchitect.co.uk/articles/display.asp?id=280`.

[Grails, 2010] Grails. The Grails Web Framework. 2010. `http://grails.org/`.

[Hall *et al.*, 2002] Tracy Hall, Sarah Beecham, & Austen Rainer. Requirements problems in twelve software companies: an empirical analysis. In *IEE Proceedings*, volume 8, pages 7–42, 2002.

[Hoffmann *et al.*, 2004] Matthias Hoffmann, Nikolaus Kühn, Matthias Weber, & Margot Bittner. Requirements for requirements management tools. In *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*, pages 1–8, Jan 2004.

[Hofmann & Lehner, 2001] Hubert F. Hofmann & Franz Lehner. Requirements Engineering as a Success Factor in Software Projects. *IEEE Software*, pages 1–9, Aug 2001.

[Hunt & Thomas, 1999] Andrew Hunt & David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. The Pragmatic Bookshelf, Jan 1999.

[Hunter, 2005] Michael Hunter. Fail Fast. 2005. `http://blogs.msdn.com/micahel/archive/2005/08/17/FailFast.aspx`.

[INCOSE, 2010] INCOSE. INCOSE Requirements Management Tools Survey. 2010. `http://www.incose.org/ProductsPubs/Products/rmsurvey.aspx`.

[Jobs, 2010] Steve Jobs. Thoughts on Flash. *Web article*, 2010. `http://www.apple.com/hotnews/thoughts-on-flash/`.

[Kajko-Mattson, 2008] Mira Kajko-Mattson. Problems in agile trenches. *Proc. of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 111–119, 2008.

[Karjalainen *et al.*, 2009] Reeta Karjalainen, Bartlomiej Kutera, Xiaozhou Li, & Youming Zhang. Supporting RE in Agile Web Application Development - A Case Study on Vixtory. *Course assignment report in the Department of Computer Science, University of Tampere*, 2009.

[Kautz & Nørbjerg, 2003] Karlheinz Kautz & Jacob Nørbjerg. Persistent Problems in Information Systems Development: The Case of the World Wide Web. In *Proceedings of ECIS*, Jan 2003.

[Keil *et al.*, 1998] Mark Keil, Paul E. Cule, & Kalle Lyytinen. A framework for identifying software project risks. *Communications of the ACM*, 41(11):76–83, Jan 1998.

[Kernighan, 2008] Brian Kernighan. Sometimes the Old Ways Are Best. *IEEE Software*, 25(6), 2008.

[Kohl, 2006] Jonathan Kohl. Post-Agilism: Process Skepticism. 2006. `http://www.kohl.ca/blog/archives/000166.html`.

[Kotonya & Sommerville, 2004] Gerald Kotonya & Ian Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, 2004.

[Laanti, 2009] Maarit Laanti. Scaling Agile - Towards The Agile Enterprise. *Conference presentation, OO Days at the Technical University of Tampere, Finland*, 2009. `http://confluence.agilefinland.com/download/attachments/2819423/Scan-Agile+2009+-+Maarit+Laanti+-+Scaling+Agile+-+Towards+the+Agile+Enterprise.pdf?version=1`.

[Lowe & Eklund, 2002] David Lowe & John Eklund. Client needs and the design process in web projects. *Journal of Web Engineering*, 1(1), Jan 2002.

[Lowe, 2001] David Lowe. Web system requirements: an overview. *Requirements Engineering*, 8(2):102–113, Jul 2001.

[Manninen & Berki, 2004] Ari Manninen & Eleni Berki. An evaluation framework for the utilisation of requirements management tools - maximising the quality of organisational communication and collaboration. In *Proceedings of BCS Software Quality Management 2004 Conference*, 2004.

[Matulevičius, 2004] Raimundas Matulevičius. How requirements specification quality depends on tools: A case study. *Advanced Information Systems Engineering*, 2004.

[McDonald & Welland, 2001] Andrew McDonald & Ray Welland. Web engineering in practice. In *Proceedings of the fourth WWW10 Workshop on Web Engineering*, pages 21–30, Jan 2001.

[Mozilla, 2009] Mozilla. Same origin policy for JavaScript. 2009. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[Murugesan & Ginige, 2005] San Murugesan & Athula Ginige. Web Engineering: Introduction and Perspectives. *Web Engineering: Principles and Techniques, Chapter 1/15*, pages 1–30, Jun 2005.

[Murugesan *et al.*, 2001] San Murugesan, Yogesh Deshpande, Steve Hansen, & Athula Ginige. Web Engineering: A New Discipline for Development of Web-Based Systems. *Web Engineering: Managing Diversity and Complexity of Web Application Development*, 2016, 2001.

[Neemuchwala, 2007] Abid Ali Neemuchwala. Evolving IT from "Running the Business" to "Changing the Business". *Web White Paper*, pages 1–9, Oct 2007. `http://www.tcs.com/SiteCollectionDocuments/White%20Papers/DEWP_05.pdf`.

[Nikula *et al.*, 2000] Uolevi Nikula, Jorma Sajaniemi, & Heikki Kälviäinen. A State-of-the-practice Survey on Requirements Engineering in Small-and Medium-sized Enterprises. *IEEE Software*, 20(6):81–89, 2000.

[North, 2003] Dan North. Introducing BDD. 2003. `http://blog.dannorth.net/introducing-bdd/`.

[Oracle, 2010] Oracle. The History Of Java Technology. 2010. `http://www.java.com/en/javahistory/`.

[Overmyer, 2000] Scott P. Overmyer. What's different about requirements engineering for web sites? *Requirements Engineering*, Jan 2000.

[Paetsch *et al.*, 2003] Frauke Paetsch, Armin Eberlein, & Frank Maurer. Requirements engineering and agile software development. In *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Jan 2003.

[Palomäki, 2007] Pirkka Palomäki. F-Secure Transition to Agile. *Conference Presentation, Scrum Alliance 2007 Fall Gathering in London*, 2007. `http://www.scrumalliance.org/resources/286`.

[Patton, 2009] Jeff Patton. Kanban Development Oversimplified. 2009. `http://www.agileproductdesign.com/blog/2009/kanban_over_simplified.html`.

[Pigoski, 1997] Thomas M. Pigoski. *Practical software maintenance: best practices for managing your software Investment.* Wiley, Jan 1997.

[Pohl *et al.*, 2005] Klaus Pohl, Günter Böckle, & Frank J Linden. *Software Product Line Engineering.* Springer, 2005.

[Pohl, 1994] Klaus Pohl. The three dimensions of requirements engineering: a framework and its applications. *Information Systems*, 19(3):243–258, 1994.

[Poranen & Kajaste, 2008] Timo Poranen & Ilari Kajaste. Software Projects 2008. *University of Tampere, Department of Computer Sciences, Report D-2008-8*, 2008.

[Poranen, 2009] Timo Poranen. Software Projects 2008-2009. *University of Tampere, Department of Computer Sciences, Report D-2009-6*, 2009.

[Rosson *et al.*, 2004] Mary Beth Rosson, Julie Ballin, & Heather Nash. Everyday programming: Challenges and opportunities for informal web development. *2004 IEEE Symposium on Visual Languages - Human Centric Computing (VL-HCC'04), Rome, Italy*, pages 123–130, Jan 2004.

[Schneider, 2002] Kurt Schneider. What to expect from software experience exploitation. *Journal of Universal Computer Science*, 8(6), Jan 2002.

[Shore, 2005] James Shore. Fail Fast. *IEEE Software*, 21(5):1–5, Feb 2005. `http://martinfowler.com/ieeeSoftware/failFast.pdf`.

[Standish, 1994] Standish. The CHAOS Report. 1994.

[Standish, 2009] Standish. CHAOS Report 2009 Press Release. 2009. `http://www1.standishgroup.com/newsroom/chaos_2009.php`.

[Sugimori *et al.*, 1977] Y. Sugimori, K. Kusunoki, F. Cho, & S. Uchikawa. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *International Journal of Production Research*, 15(6):553–564, Jan 1977.

[TIOBE, 2010] TIOBE. TIOBE Programming Community Index for May 2010. 2010. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[Venners, 2004] Bill Venners. The Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V. 2004. `http://www.artima.com/intv/simplest.html`.

[Venners, 2009] Bill Venners. Twitter On Scala - A Conversation with Steve Jenson, Alex Payne, and Robey Pointer. 2009. `http://www.artima.com/scalazine/articles/twitter_on_scala.html`.

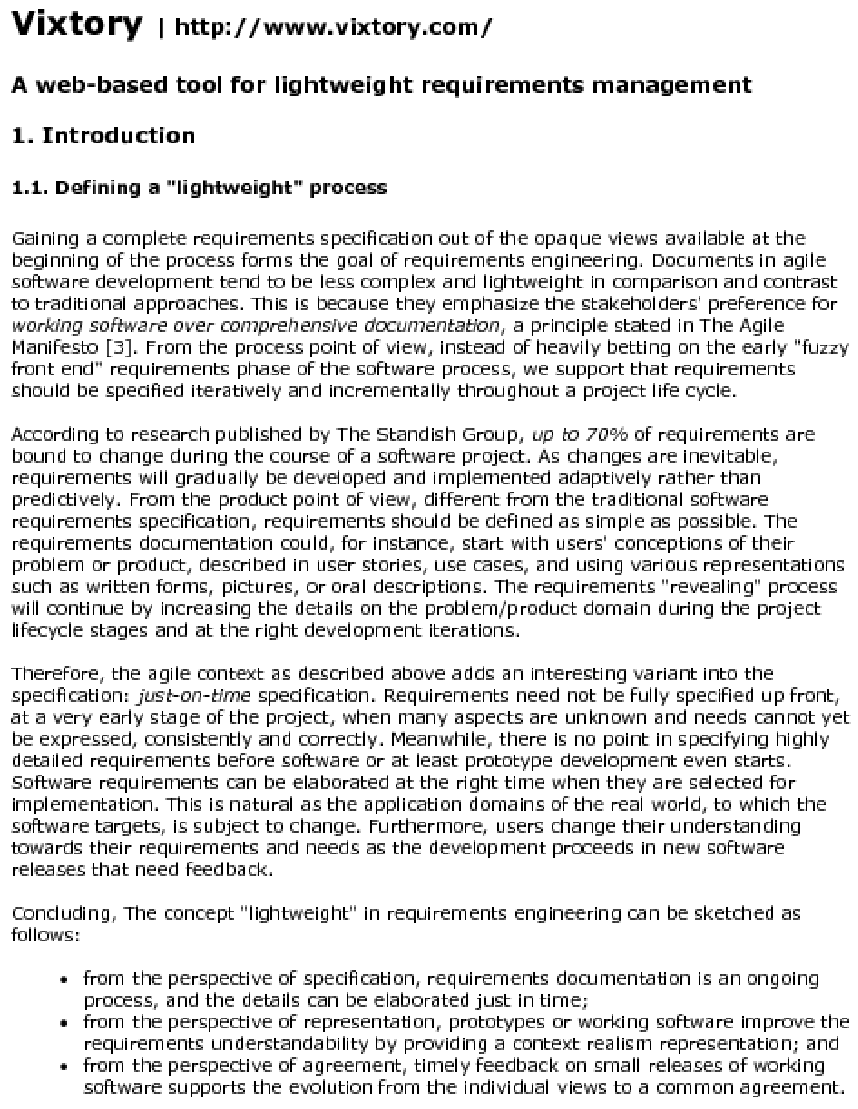
[Wells, 2010] Don Wells. Extreme Programming Wiki: Simplicity is the Key. 2010. `http://www.extremeprogramming.org/rules/simple.html`.

[Young, 2004] Ralph Rowland Young. *The requirements engineering handbook*. Artech Print on Demand, Jan 2004.

[Zakon, 2010] Robert Zakon. Hobbes' Internet Timeline 10. 2010. `http://www.zakon.org/robert/internet/timeline/`.

[Zhang *et al.*, 2010] Zheying Zhang, Mike Arvela, Eleni Berki, Matias Muhonen, Jyrki Nummenmaa, & Timo Poranen. Towards lightweight requirements documentation. *Submitted to the International Workshop on Requirements Analysis 2010, in review*, pages 1–10, 2010.

Appendix A: Course assignment



Vixtory | **http://www.vixtory.com/**

A web-based tool for lightweight requirements management

1. Introduction

1.1. Defining a "lightweight" process

Gaining a complete requirements specification out of the opaque views available at the beginning of the process forms the goal of requirements engineering. Documents in agile software development tend to be less complex and lightweight in comparison and contrast to traditional approaches. This is because they emphasize the stakeholders' preference for *working software over comprehensive documentation*, a principle stated in The Agile Manifesto [3]. From the process point of view, instead of heavily betting on the early "fuzzy front end" requirements phase of the software process, we support that requirements should be specified iteratively and incrementally throughout a project life cycle.

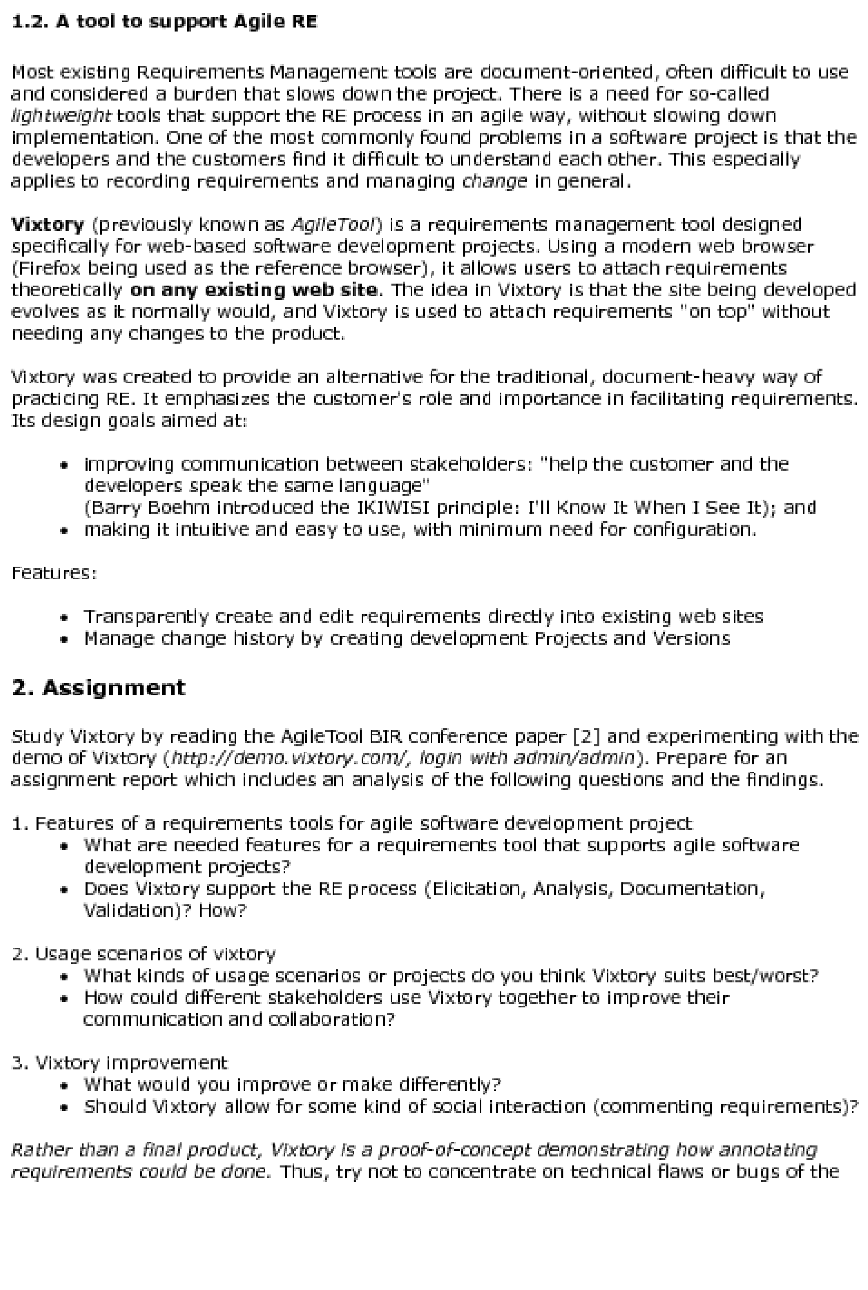
According to research published by The Standish Group, *up to 70%* of requirements are bound to change during the course of a software project. As changes are inevitable, requirements will gradually be developed and implemented adaptively rather than predictively. From the product point of view, different from the traditional software requirements specification, requirements should be defined as simple as possible. The requirements documentation could, for instance, start with users' conceptions of their problem or product, described in user stories, use cases, and using various representations such as written forms, pictures, or oral descriptions. The requirements "revealing" process will continue by increasing the details on the problem/product domain during the project lifecycle stages and at the right development iterations.

Therefore, the agile context as described above adds an interesting variant into the specification: *just-on-time* specification. Requirements need not be fully specified up front, at a very early stage of the project, when many aspects are unknown and needs cannot yet be expressed, consistently and correctly. Meanwhile, there is no point in specifying highly detailed requirements before software or at least prototype development even starts. Software requirements can be elaborated at the right time when they are selected for implementation. This is natural as the application domains of the real world, to which the software targets, is subject to change. Furthermore, users change their understanding towards their requirements and needs as the development proceeds in new software releases that need feedback.

Concluding, The concept "lightweight" in requirements engineering can be sketched as follows:

- from the perspective of specification, requirements documentation is an ongoing process, and the details can be elaborated just in time;
- from the perspective of representation, prototypes or working software improve the requirements understandability by providing a context realism representation; and
- from the perspective of agreement, timely feedback on small releases of working software supports the evolution from the individual views to a common agreement.

Figure 8.1 Course assignment, page 1



1.2. A tool to support Agile RE

Most existing Requirements Management tools are document-oriented, often difficult to use and considered a burden that slows down the project. There is a need for so-called *lightweight* tools that support the RE process in an agile way, without slowing down implementation. One of the most commonly found problems in a software project is that the developers and the customers find it difficult to understand each other. This especially applies to recording requirements and managing *change* in general.

Vixtory (previously known as *AgileTool*) is a requirements management tool designed specifically for web-based software development projects. Using a modern web browser (Firefox being used as the reference browser), it allows users to attach requirements theoretically **on any existing web site**. The idea in Vixtory is that the site being developed evolves as it normally would, and Vixtory is used to attach requirements "on top" without needing any changes to the product.

Vixtory was created to provide an alternative for the traditional, document-heavy way of practicing RE. It emphasizes the customer's role and importance in facilitating requirements. Its design goals aimed at:

- improving communication between stakeholders: "help the customer and the developers speak the same language"
  (Barry Boehm introduced the IKIWISI principle: I'll Know It When I See It); and
- making it intuitive and easy to use, with minimum need for configuration.

Features:

- Transparently create and edit requirements directly into existing web sites
- Manage change history by creating development Projects and Versions

2. Assignment

Study Vixtory by reading the AgileTool BIR conference paper [2] and experimenting with the demo of Vixtory (*http://demo.vixtory.com/, login with admin/admin*). Prepare for an assignment report which includes an analysis of the following questions and the findings.

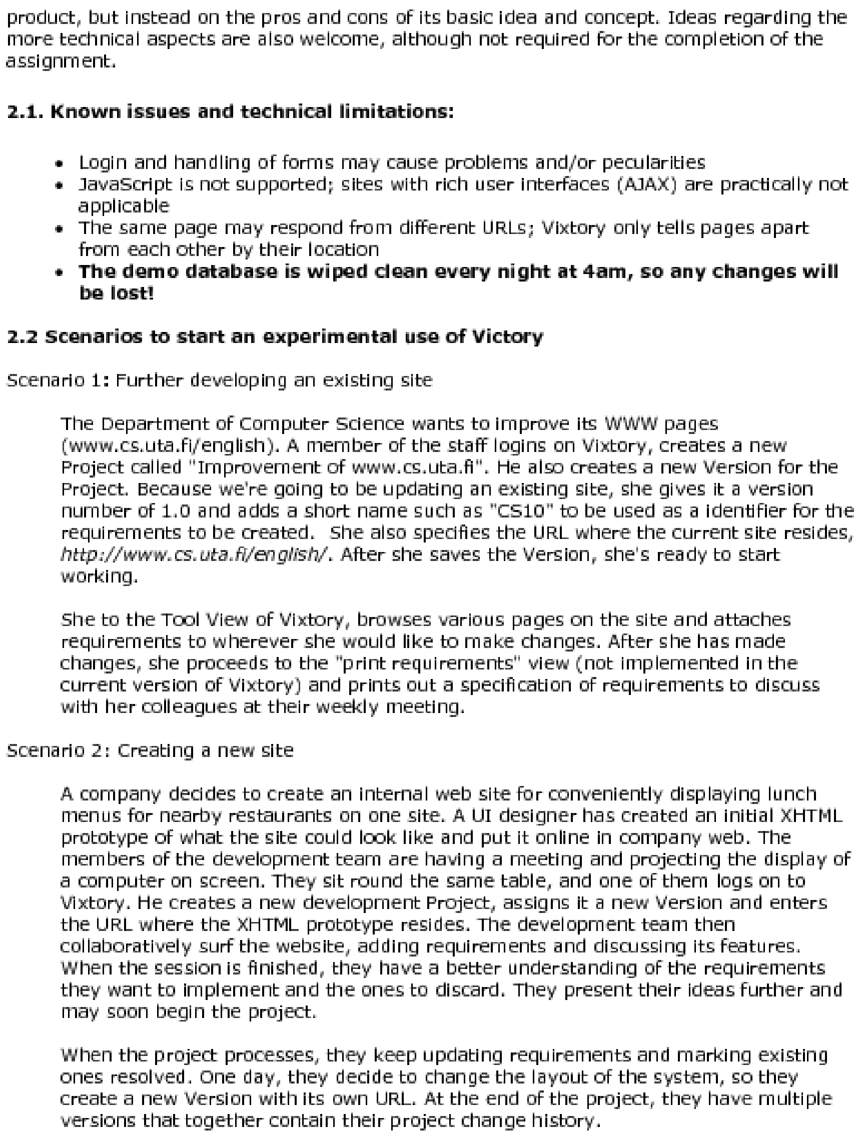
1. Features of a requirements tools for agile software development project
   - What are needed features for a requirements tool that supports agile software development projects?
   - Does Vixtory support the RE process (Elicitation, Analysis, Documentation, Validation)? How?

2. Usage scenarios of vixtory
   - What kinds of usage scenarios or projects do you think Vixtory suits best/worst?
   - How could different stakeholders use Vixtory together to improve their communication and collaboration?

3. Vixtory improvement
   - What would you improve or make differently?
   - Should Vixtory allow for some kind of social interaction (commenting requirements)?

Rather than a final product, Vixtory is a proof-of-concept demonstrating how annotating requirements could be done. Thus, try not to concentrate on technical flaws or bugs of the

Figure 8.2 Course assignment, page 2



product, but instead on the pros and cons of its basic idea and concept. Ideas regarding the more technical aspects are also welcome, although not required for the completion of the assignment.

2.1. Known issues and technical limitations:

- Login and handling of forms may cause problems and/or pecularities
- JavaScript is not supported; sites with rich user interfaces (AJAX) are practically not applicable
- The same page may respond from different URLs; Vixtory only tells pages apart from each other by their location
- **The demo database is wiped clean every night at 4am, so any changes will be lost!**

2.2 Scenarios to start an experimental use of Victory

Scenario 1: Further developing an existing site

The Department of Computer Science wants to improve its WWW pages (www.cs.uta.fi/english). A member of the staff logins on Vixtory, creates a new Project called "Improvement of www.cs.uta.fi". He also creates a new Version for the Project. Because we're going to be updating an existing site, she gives it a version number of 1.0 and adds a short name such as "CS10" to be used as a identifier for the requirements to be created. She also specifies the URL where the current site resides, *http://www.cs.uta.fi/english/*. After she saves the Version, she's ready to start working.

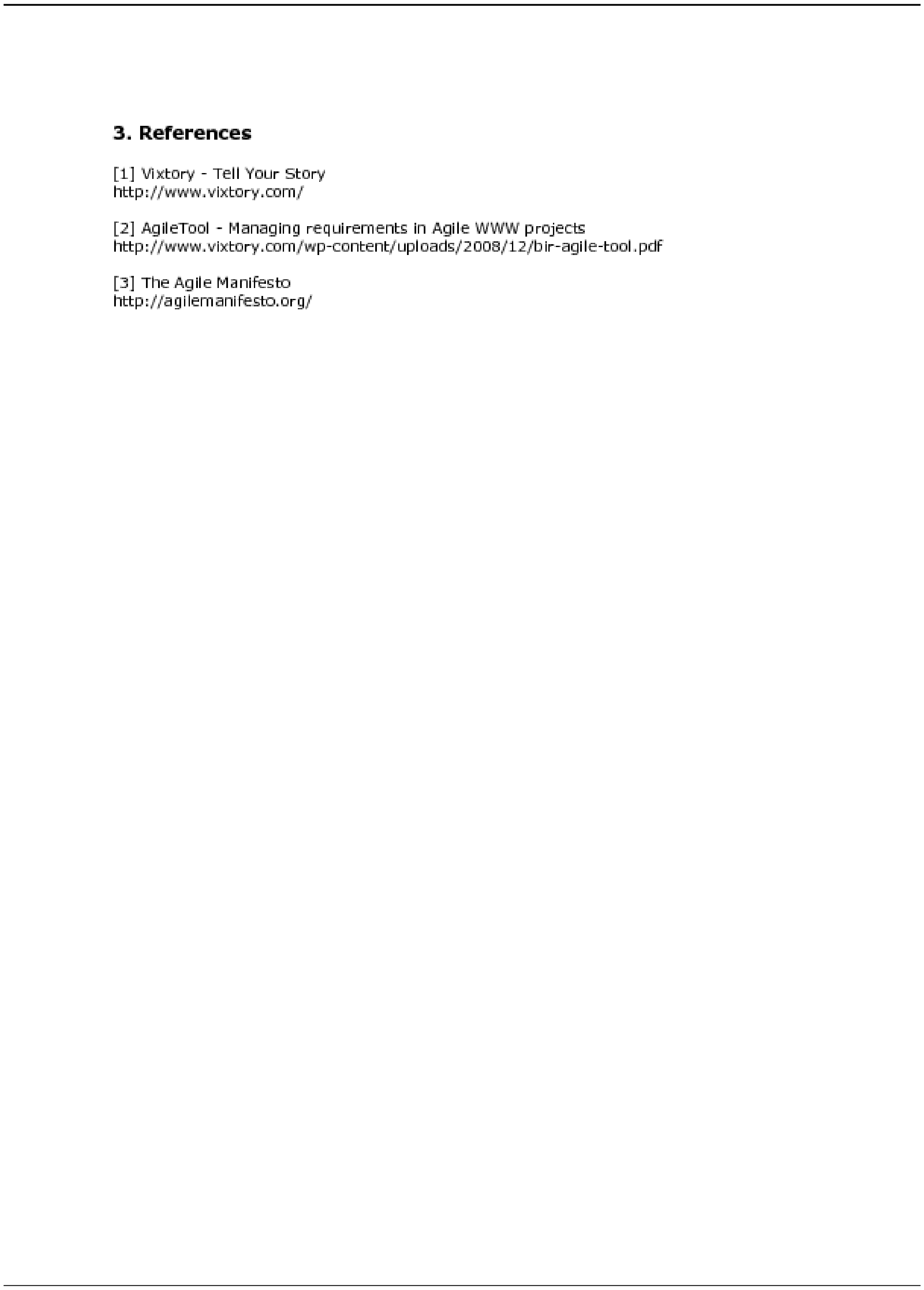
She to the Tool View of Vixtory, browses various pages on the site and attaches requirements to wherever she would like to make changes. After she has made changes, she proceeds to the "print requirements" view (not implemented in the current version of Vixtory) and prints out a specification of requirements to discuss with her colleagues at their weekly meeting.

Scenario 2: Creating a new site

A company decides to create an internal web site for conveniently displaying lunch menus for nearby restaurants on one site. A UI designer has created an initial XHTML prototype of what the site could look like and put it online in company web. The members of the development team are having a meeting and projecting the display of a computer on screen. They sit round the same table, and one of them logs on to Vixtory. He creates a new development Project, assigns it a new Version and enters the URL where the XHTML prototype resides. The development team then collaboratively surf the website, adding requirements and discussing its features. When the session is finished, they have a better understanding of the requirements they want to implement and the ones to discard. They present their ideas further and may soon begin the project.

When the project processes, they keep updating requirements and marking existing ones resolved. One day, they decide to change the layout of the system, so they create a new Version with its own URL. At the end of the project, they have multiple versions that together contain their project change history.

Figure 8.3 Course assignment, page 3



3. References

[1] Vixtory - Tell Your Story
http://www.vixtory.com/

[2] AgileTool - Managing requirements in Agile WWW projects
http://www.vixtory.com/wp-content/uploads/2008/12/bir-agile-tool.pdf

[3] The Agile Manifesto
http://agilemanifesto.org/

Figure 8.4 Course assignment, page 4