Cost-Effectiveness of Test-Driven Development

Master's Thesis

Dennis de Bode

Cost-Effectiveness of Test-Driven Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dennis de Bode
born in Nijmegen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

ISM eCompany
Van Nelleweg 1
Rotterdam, the Netherlands
www.ism.nl

Cost-Effectiveness of Test-Driven Development

Author: Dennis de Bode
Student id: 1027328
Email: ddebode@gmail.com

Abstract

Test-Driven Development (TDD) is a software development practice in which test cases are written before code implementation. ISM eCompany, who specializes in web solutions, is adapting their development process to become more agile. One of those adaptations is that ISM wants to introduce TDD into their development process, to have projects with a better code quality and a regression test suite. ISM only worries whether the writing of test cases do create too much overhead for the developers and using a TDD approach does not have a large effect on the time spent on fixing bugs and rework. This can cause the projects to overrun and that makes the practice TDD not cost-effective. This thesis will describe a case study of the introduction of TDD in two different projects at ISM. We have proven that TDD can be introduced into the development process of ISM. By introducing TDD in two projects of ISM we tried to determine whether TDD is cost-effective by determining the Return on Investment (ROI). This cost-effectiveness could, due to the insufficient dataset, only be assessed in a limited way. In the monitored projects the developers took 31% more time for developing work items. On the other hand, the developers spent less time on rework and bugs than in projects where no TDD is used. However, it is hard to conclude that this is due to the usage of a TDD approach, because of the low percentage of work items implemented with a TDD approach (17% of the work items). The developers struggled with TDD, because of the lack of experience with writing test cases.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Phil. H.-G. Gross, Faculty EEMCS, TU Delft
Company supervisor:	Ir. J. Appelo, ISM eCompany
Committee Member:	Ir. B. R. Sodoyer, Faculty EEMCS, TU Delft

Preface

First of all I want to thank all the people at ISM eCompany for their cooperation during this project. I would like to thank my supervisor at ISM Jurgen Appelo for the chance to conduct my thesis research at ISM and his guidance throughout this project. Special thanks go to Alexander, Mikael, Raymond, Peter and Tuba for participating in this case study and for Antonio and Yulia for keeping me company during the lunch breaks.

I would also like to thank my supervisor at TU Delft Gerd Gross for the feedback he has giving me on my work and for guiding me in the right direction. Special thanks goes to Roland Voets, who has always reviewed my work for grammatical errors and for keeping me awake during the trips to Delft.

Many thanks for my brothers and friends, who supported me and giving me the confidence to continue. Finally I want to thank my parents, who always morally and financially supported me throughout my study.

Dennis de Bode
Delft, the Netherlands
July 2, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Test-Driven Development

Test-Driven Development (TDD) [5] is a design practice that uses short development iterations, in which the developer writes a test before he writes just enough code to make the test pass (see section 2.3 for a more elaborate explanation). After the test passes, the programmer can refactor the code if necessary or write another test and then the cycle starts again. This cycle ends when no further tests can be defined. It is a practice which is recommended by the agile community [40] because it reduces fault introduction, improves code quality, and (according to some reserachers [26] [28] [16]) the code can be written faster when using TDD than when using a more traditional approach in which testing come at the end (see section 3.1).

1.2 Scope of the Thesis

This thesis describes the introduction and evaluation of TDD in the context of the company: ISM ECompany. We tried to asses whether TDD is cost-effective in their projects.

ISM eCompany is a full-service web company. They focus on different web activities such as: internet technologies, e-learning, Enterprise Resource Planning (ERP) and Content Relation Management (CRM) implementations. Through different takeovers, they have now more than 200 employees and they opened a development department in Zhitomir Ukraine. Customers of ISM eCompany include Heineken, AKO, Vitae, A.S. Watson and Akzo Nobel.

ISM is developing the Sana Software platform which is divided in three different categories: Content Manangement Solutions, Ecommerce Solutions and Elearning solutions. Customers can use these solutions to manage, build, and publish their websites, webshops or elearning courses. These products can be bought as an off-the-shelf product or can be integrated with a custom build web application. These customizations are performed by the ISM project development teams.

ISM has adapted their project life cycle to the practices of Scrum [41], so it is better suited for their dynamic development process. The development teams, who work on the

Sana Software, already write unit tests for their code. They released new versions of their Sana Software solutions, which makes it technically better possible to write unit tests in the customization projects (newer .net version, interfaces available for mocking [32] etc). ISM wants to introduce the writing of unit tests following a TDD approach in the project development teams, so the testing of their web applications will improve, and a regression test suite is build.

However, the question remains whether a TDD approach really has more benefits than disadvantages. The projects at ISM have very tight deadlines, because they are in a highly competitive market. Writing unit tests in a web application, which is integrated with a third party software, can be complex and therefore can create extra overhead. Furthermore, the projects at ISM have short duration (almost all projects have an average size of less than hundred working days). Do the investments put in TDD be returned in such a short notice? If this is not the case, it may cause the projects to overrun in terms of time and money.

To asses this we have determined whether TDD can be applied in a cost-effective way in the context of ISM. This was determined by determining the Return on Investment (ROI) of TDD in two different projects at ISM. ROI is the ratio of money gained or lost after an investment [39]. In our case it was investigated how many hours are saved on working on bugs and rework after investing hours in TDD.

1.3 Research Questions

The main research question is:

1. Is Test-Driven Development cost-effective in the researched context?

With context is meant the development of a customized web application integrated with an existing and evolving software product at ISM eCompany. Sub-questions are:

1. How can Test-Driven Development be implemented into the development process of ISM?
2. What are the costs in terms of overhead of Test-Driven Development in the context of a project at ISM?
3. What are the effects on the effort on fixing bugs and rework of Test-Driven Development in the context of a project at ISM?

1.4 Research Approach

A case study is performed on the introduction of TDD in the projects of the project development teams at ISM. By collecting quantitative (metrics) and qualitative data (interviews), we tried to determine what the costs and the effects are of a TDD approach in a project at ISM.

First existing literature was researched about TDD and the different case studies already performed on TDD. Then the development process of ISM was researched and whether unit

testing is technically feasible in combination with Sana Software. Our findings is described in different documents (see appendix B. for an example) and training material is written about TDD for the developers at ISM.

Then the practice of Test-Driven Development was introduced in two different project teams. Quantitative data was collected by monitoring the issue trackers of the teams. This data is compared with another project of ISM, where no TDD approach is used. After collecting the quantitative data, qualitative data was collected by interviewing the members of the project development team, who participated in the case study. By holding interviews we got more insight in the quantitative data that was collected.

After all the data was collected, it was analyzed and evaluated whether the practice TDD is cost-effective in projects here at ISM by determining the ROI.

1.5 Outline

The next chapter describes the practice TDD and what the benefits and disadvantages are of using a TDD approach. In chapter 3 an overview is given of similar case studies performed at TDD in an academic or industrial context. Chapter 4 describes the current development process and testing process of ISM. Furthermore, we describe how the new development process will look like. Chapter 5 shortly discuss the architecture of the Sana Software. Chapter 6 describes our case studies and we present our results from the case studies. Chapter 7 describes the limitations of our case study and in chapter 8 we discuss the different points of this paper. Finally in chapter 9 we summarize our conclusions and give recommendations for further research.

Chapter 2

Test-Driven Development

2.1 Agile Software Development

Agile Software development refers to a group of development methodologies with the same characteristics, such as extreme programming (XP) [48], Dynamic Systems Development Method (DSDM) [10] and Scrum [41]. Representatives of the different methodologies met with each other in 2001 to form the "Agile Manifesto" that contains the values and principles that underpin agile development [29]. These values are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

These values can be found in every agile software development methodology. Following these values, the different agile software development methodologies have a number of similar characteristics [2].

2.2 Characteristics of Agile Software Development

- Development Iterations

The development of the product is divided in a series of small iterations. At the end of each iteration there should be a prototype for the customer to review. These iterations are usually between 15-30 days.

- Face-to-Face Communication

As the physical distance between people increases, the effectiveness of their communication decreases [47]. This happens when people send emails to each other or communicate

requirements through documents. It creates delays and misunderstandings. Agile methods emphasize on colocation off all the stakeholders [8]. All stakeholders of a team should be physically near each other during a project, including the customer, so issues can be quickly resolved [44]. In a field study by Teasley et al. [43] at a Fortune top 100 company they noticed a significantly higher productivity when a development team was located in one room.

- Small, self-organizing, cross-functional Project Teams

Larger teams need more communication and generate more process overhead. This will reduce the overall productivity [47]. Therefore Agile methods advocate small teams. If the teams become too big, they have to be divided in smaller teams. The different methodologies advice different team sizes.

Teams in agile software development are self-organizing and cross-functional, without any corporate hierarchy [47]. The team members make their own planning and estimations. This means that they are all responsible and that the project is a team effort. If necessary developers will test and testers will develop. The team is completed by an on-site customer representative. This person is appointed by the stakeholders and is available for the developers to ask domain specific questions.

- Change is welcome

Agile methodologies encourage changes, instead of discouraging it [22]. Instead that the requirements are determined before design, agile methods leave room for the customers to add changes to the project unless the changes violate the broad scope, schedule, and cost constraints set by the purchasing customer (or management). The changes can be added at fixed points in the project life cycle (usually before a new iteration).

- Testing

Agile methodologies recommend to test early and often. That is why they promote TDD and automated testing [47][8]. When the tests are run early in the development process, the bugs can be found quickly and therefore they are easier and cheaper to fix. If all the tests are automated, they can be run fast and often, for example by every build of the system.

2.3 Test-Driven Development

2.3.1 Process

One of the practices which are made popular by the agile community and by Extreme Programming [4], in particular, is Test-Driven Development (TDD) [5]. TDD is a test first approach, where the test cases are written before implementation. When the implementation is done, new test cases are written and this drives the development forward.

This practice is closely related with unit testing, because in most cases when developers use TDD, the test cases are written with the assistance of xUnit frameworks [5] [18]. Unit

tests test the smallest testable part of the source code. They provide a strict, written contract that the code must satisfy. Usually the unit tests exercise a particular method in a particular context [24]. These unit tests must be all independent of each other and from external sources (like databases, GUI's etc). This makes the tests better maintainable and fast.

TDD has a cyclic approach where in small steps additional functionality is added. This functionality is completed when no further tests have to be added (see 2.1).

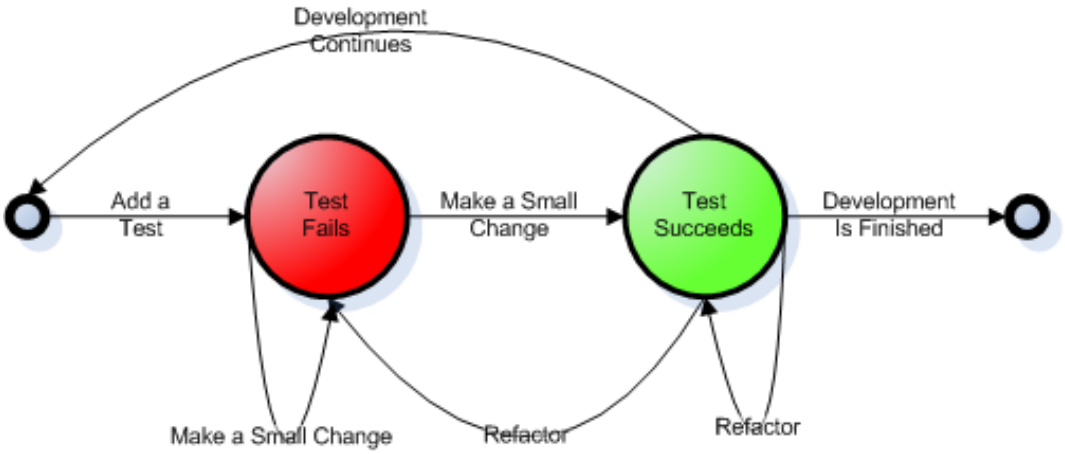


Figure 2.1: Test-Driven Development Cycle.

The TDD cycle consist of three phases:

1. Red Phase.

In the Red Phase, the developer writes a test for the new feature he wants to implement. An important rule in TDD is: "If you can not write a test for what you are about to code, then you should not even be thinking about coding"[7]. All old tests should succeed, except the new one.

2. Green Phase.

In the Green Phase, the developer writes the smallest possible piece of code to make the test succeed. This way, the developer makes small incremental steps of development. If he makes bigger steps, there is an increased risk that faults slip in the code.

3. Refactoring.

When the new test succeeds, the developer must make his code has clean as possible in order to make the test succeed. In other words, all duplication, created in making the test pass, must be removed. By running all the tests you can ensure that the refactoring activities do not break anything. Now if additional tests can be written, the cycle starts again. If this is not the case, then development stops.

When the developer has finished the TDD cycle, his implementation is done and a regression test suite has been built to test it.

2.3.2 Benefits

TDD is not a testing practices but a design practice. However, TDD supports the testing process by the regression tests it builds [27]. Other benefits are:

- TDD increases the confidence of the developer in existing code [27].

When using TDD all functionalities are tested [31]. When a new feature is added, you can easily verify if something is broken by running the tests.

- Bugs are quicker found and fixed.

With TDD bugs are quicker found and therefore easier to fix. It is easier to find and fix a bug after writing ten lines of code than when writing thousand lines of code [6].

- TDD reduces defect injection [49].

When a bug is found during maintenance and debugging of software, a patch is made to fix this. Unfortunately such fixes and small code changes are 40 times more error prone than new development [23]. TDD can facilitate the introduction of new functionality in the code base, by the regression test suite build during the use of a TDD approach. In addition, when using TDD, the developer has to think what is really expected from his code. He only makes small development steps and all old and new functionalities are directly tested.

- TDD increases code comprehension.

An additional benefit of unit testing is that it positively influences the program understanding (comprehension) [45]. 40% to 60% percent of the time spent on software maintenance is used on program understanding [11] [38]. Unit tests explain the behavior of the code that is tested and therefore developers can look at the unit tests to see how the code should behave.

- TDD makes testing more a satisfying process [50].

Testing provides negative feedback. The developer gets the message from the testing process that he has failed to build something that works correctly. The testing process usually ends in failure. By reversing the process, by first testing and then coding, the developer gets the message that he has succeeded to pass the test.

2.3.3 Shortcomings

There also some disadvantages about TDD:

- Lack of design

In the philosophy of TDD practitioners, the test cases are a replacement of the requirements. TDD often does not include any upfront design [21]. This increases the risk in a project because there is no normal defense as with explicit design and documentation [45]. However, the company ISM does already not make an elaborate upfront design. All functional requirements are described in user stories [9] (see section 4.1).

- Applicability of practice

Some code is hard to unit test (for example graphical user interfaces) [21] [1]. This can be an extra challenge for introducing TDD at ISM, because web applications tend to have a lot of GUI's. Besides the customization project has many dependencies on the Sana Software solution. This will make it hard to write unit tests which touch as less as possible functionalities of Sana Software. Unit tests must be independent from external sources to keep them fast and flexible. Functionalities of Sana Software must therefore be mocked in the unit tests [32].

- Skill level

Developers need high level of experience and determination to write test cases for code that is hard to test. Average developers may lack the discipline to maintain the test cases [21]. Furthermore, TDD is reported to have the steepest learning curve of all agile practices [1].

The benefit for a TDD approach at ISM is that less bugs are injected to the code. If a bug introduced, it can be quicker found and therefore quicker be fixed. Also the developers have more confidence to implement new functionalities, because the regressions test suite will check if any old functionality did not break. Disadvantages are that most developers do not have any experience with TDD and it will take a lot of effort to write the unit tests.

Chapter 3

Related Work

There are many case studies performed in the domain of TDD, which can be separated into academic controlled experiments and industrial case studies. In the academic controlled experiments, the works of students in lab courses are monitored to see what the results are of using TDD versus traditional approaches. In the industrial case studies, TDD is introduced into the development process of professional developers and the results are monitored.

3.1 Academic Controlled experiments

The academic case studies differ between studies which have a short duration (12-40 hours) and studies which have a much longer duration (week-months). Most of these studies focus on the productivity of the developers and the quality of the code (see table 3.1). This can be measured by the defect density [15] [33] or by focusing more on software metrics like class size, code coverage, coupling, cyclomatic complexities etc. [26] [28].

The results of the different studies really differ: Some indicate that there is more overhead on development when using TDD [19] and some indicate that developers using TDD are more productive [26] [28]. The outcomes of most academic studies suggest that the code quality improves when TDD is used [26] [28] [15] [19] [51], but some conclude that TDD does not have a positive effect on the code quality at all:

Lech Madeyski [33] measured the quality of the code by measuring the number of acceptance tests that passed and when using the TDD approach, instead of a test-last approach, the external code quality (this is assessing the code quality by looking from the outside instead of looking inside the code) was lower. The case study by Erdogumus et al. [16] concluded that in the context of their case study(a small programming assignment) TDD did not deliver a better quality code then a test-last approach.

3.2 Industrial case studies

There are also case studies performed on TDD in an industrial context. A main difference with the academic controlled experiments is that the time span of the different studies is much longer (from 3 hours up till 5 years) and the average code size of the projects is larger.

Table 3.1: Academic Controlled Experiments.

	Researchers	**Duration**	**Productivity**	**Code Quality**	**Year**
1.	David Janzen [26]	n/a	100% increased productivity	Lower computational complexity, higher coupling and higher test coverage.	???
2.	Reid Kaufmann, David Janzen [28]	n/a	50% increased productivity	Better Design and more confidence in code.	2003
3.	Stephen H. Edwards [15]	2/3 weeks	n/a	45% fewer defects.	2003
4.	Lech Madeyski [33]	10 hours	n/a	Lower external Code Quality.	2005
5.	H. Erdogumus [16]	13 hours	Increased productivity based on the higher number of tests.	No significant difference.	2005
6.	S. Yenduri, L. Perkins [51]	n/a	317 hours less then traditional development.	50% lower faults during Acceptance Testing.	2006
7.	Boby George [19]	1-4 hours	16% increased effort	Better External Code Quality.	2007

What furthermore can be noticed, that in more cases the developer has a lower productivity when using TDD, however almost all studies indicated a higher code quality (see table 3.2).

The results of the studies differ much from each other because every experiment must be seen in its own context. However, if the overall trend of the results of the industrial case studies is considered, it can be noticed that there is an increase of development effort, but it results in a better code quality. However, not all industrial case studies on TDD have results in which the application of TDD produces better code quality.

P. Abrahamsson et al.[1] describes a case study in which the introduction of TDD is a failure. The case study concerns a team of three students with industrial experience and one experienced professional developer. They developed a mobile application for the global market. The team should develop the application with a TDD approach, but were reluctant to adopt the approach. In the first iteration the team spent 30% of the time on TDD and in the next iterations this percentage even dropped. One of the reasons of why the team was reluctant to adopt TDD was: the team believed that TDD is not suitable for the kind of application that the project involved (with a strong focus on GUI's). Another reason was the inexperience of the developers with the platform and with TDD.

The lessons to be learned from this case study are:

Table 3.2: Industrial Case Studies.

	Researchers	**Duration**	**Productivity**	**Code Quality**	**Year**
1.	Randy A. Ynchausti [52]	8.5 hours	Up to 100% increased effort	38-267% fewer defects.	2001
2.	Laurie Williams, Boby George [20]	3-6 hours	16% increased effort	18% more black-box test cases succeeded.	2003
3.	Laurie Williams et al. [49]	n/a	No significant difference.	40% lower defect density.	2003
4.	E. Maximilien, L. Williams [34]	n/a	n/a	50% lower defect density.	2003
5.	Thirumalesh Bhat, N. Nagappan [6]	4 months	35% increased effort	62% lower defect density.	2006
6.	Thirumalesh Bhat, N. Nagappan [6]	7 months	15% increased effort	76% lower defect density.	2006
7.	Lars-Ola Damm, Lars Lundberg [12]	1-1.5 years	Total project cost reduced with 5-6%.	Decreased Fault Slip Through rates (5-30%) Lower Avoidable Fault Costs (60%).	2006
8.	Julio Cesar Sanchez et Al. [25]	5 years	19% increased effort.	40% lower defect density.	2007

- The team has to be motivated to use TDD.
- TDD has one of the steepest learning curve of all agile practices.
- Good tools for TDD must be available.
- A mentor with experience of TDD must be added to the team.

3.2.1 Uniqueness of our research

As can be seen from the tables 3.1 and 3.2, there are a lot of different case studies performed on TDD. However, most of them focus on the claims made about TDD:

1. Productivity of a developer should not be lower when he is using TDD instead of a more traditional development approach.

2. When using TDD, the developer produces code with a better quality.

In our work we are not going to focus whether using TDD produces better code, but the cost-effectiveness of TDD (which should be a result of better code). This research is related to [12] where they researched the Return on Investment of TDD, but our research has a completely different context:

- The duration of the projects in [12] were between 1-1.5 years. The projects at ISM are much shorter (as mentioned in section 1.1).
- Totally different kind of applications: Components for a mobile operator network versus web applications.
- In [12] they use TDD at a component level and in this thesis a more traditional approach of TDD is used. The developers try to write at least one unit test for each method, which he is going to implement.
- The projects in [12] have a different testing/development approach then projects in our research.

Besides the context, another approach is used to determine the cost-effectiveness:

- The costs of training the developers in TDD is not included. The training of the developers in our case study did not take more than one day. This training happened outside the projects of the case study and therefore it is not included. Besides, because of the short duration of the training, the costs will not have any significant influence on the cost-effectiveness of using TDD at ISM.
- In [12] they estimate the costs of fixing faults (Avoidable Fault Costs). An estimation is made of the additional cost when a fault is discovered at different phases in the project. Then by counting how many faults are found at each phase, they could determine the costs. We measure the costs of fixing faults by determining the number of hours spent by the developer.
- In [12] they assume that using TDD does not lead to extra costs, because of the extra work of designing/running/updating test cases. In their case, designing test cases replaced other designing activities (they did not mention whether these new activities needed the same amount of effort as the replaced activities). In our case, writing test cases is not a replacement of another activity and we are going to measure/estimate the increased effort of using TDD.

When you have better code, you should have lesser bugs and rework on work items. TDD is cost effective, when the overhead for writing/running/updating the test cases upfront is less than the time saved on rework and fixing bugs.

Chapter 4

Development Process

The information in this section was acquired by studying documents of ISM, by conducting interviews with different persons with different roles in the development process and by participating as software tester in different projects. For the interviews, different persons with different roles were selected to get a good coverage among the different teams and roles.

There are four project development teams at ISM in the Netherlands, which work in project-based teams at developing web applications and two product development teams which work on the Sana Software solutions. The next section only describes the structure, development process and test process of the project development teams.

A project development team can work on different projects at the same time. When this is the case, the development team is splitted into several smaller project teams. The teams are using an adaption of Scrum [41] to develop their web applications.

4.1 User Stories

All functional requirements of the projects at ISM are defined in User Stories [9] and the graphical design is defined in a separate document. ISM believes it is unnecessary to document implementation details in the functional requirements, because the developer will probably misunderstand them or knows a better way to implement the requirements. User Stories are stripped from all implementation details. If the developer wants details about the User Story, he can better communicate directly with the Product Owner (a Scrum Role) to avoid misunderstandings. All these User Stories are listed on a Product Backlog as Product Backlog Items. These User Stories are defined by the Product Owner in cooperation with the customer.

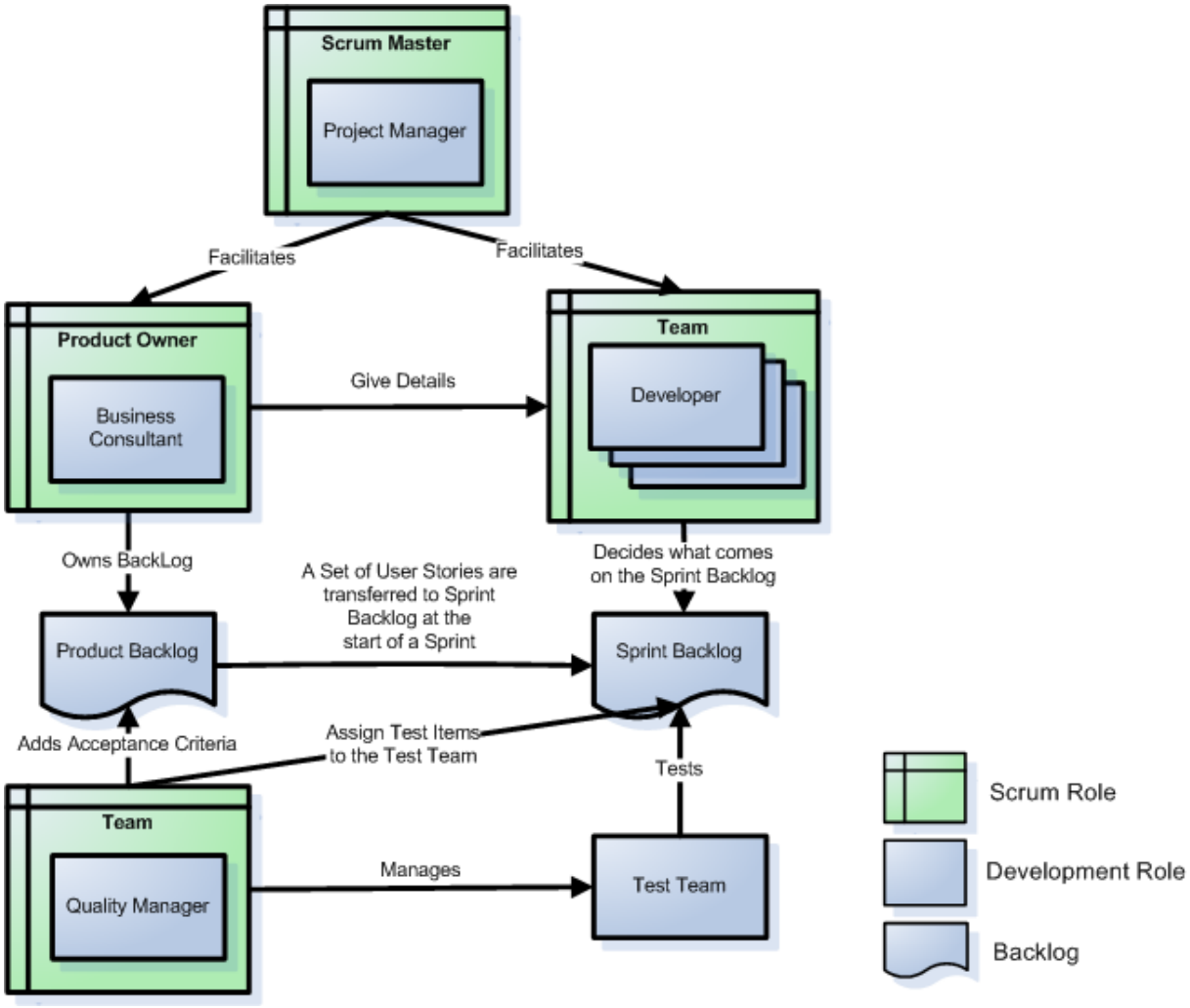


Figure 4.1: Roles at ISM.

4.2 Roles

4.2.1 Business Consultant

Because the development teams have multiple customers at the same time it is hard to have a customer as Product Owner on colocation. That is why the role is fulfilled by the Business Consultant. A Business Consultant makes the user stories in cooperation with the customer and is responsible that end product fulfills the customer requirements. He decides whether the implemented User Story meets the acceptance criteria.

4.2.2 Quality Manager

The Quality Manager defines in collaboration with the Product Owner the acceptance criteria for the different product backlog items. His responsibility is also assigning testers from the test team to test whether the implemented product backlog items meet the acceptance criteria. ISM considers that the Quality Manager is a part of the Scrum role ”The Team”. The Quality Manager is present at the daily stand-ups, sprint review meetings and sprint planning meetings, however he does not decide which Product Backlog items are transferred to the Sprint Backlog and does not give estimates. He only gives commitment if the items can be tested during the sprint.

4.2.3 Project Manager

The Project Manager has the role of Scrum Master. He facilitates the team and removes impediments. The planning of the projects and the decisions on which days developers work on which project are made by him, but the planning within a sprint is made by the team itself. He is responsible for projects remaining within their planning and budget.

4.2.4 Development Team

The team consists of junior, medior and senior developers, who work together on the same project. The team makes their own planning and estimation. They decide in cooperation with the Product Owner which User Stories will be put on the sprint backlog. Some teams have a lead developer which can make the final call on implementation issues.

4.2.5 Test Team

The test team is located in the Ukraine. The Test Team tests all projects and all developed products from ISM. The Quality Manager manages the test team from the Netherlands and makes sure that there are testers available to test a project. Because the testers are not on the same location as the rest of the team, they are not present at any of the meetings.

4.3 Project Life Cycle

Because the projects at ISM have a short duration and the customers want much more input during the development process, ISM decided to have sprint cycles of a week. All the teams will start on Monday the new cycle with a sprint planning meeting. The customer can have input on the implementation on a weekly basis, instead after thirty days, the usual length of a Scrum cycle. The Product Owner will communicate with the customer during the week. Before the next sprint planning meeting he will update the priorities on the product backlog. The User Stories with the highest priority will be implemented first, if the team is willing to commit to it.

In the Sprint Planning Meeting the team decides which User Stories they can implement the coming week. When the User Story is too big, it is split into multiple work items. The team gives an estimate on how long it will take to implement the work items. If there is

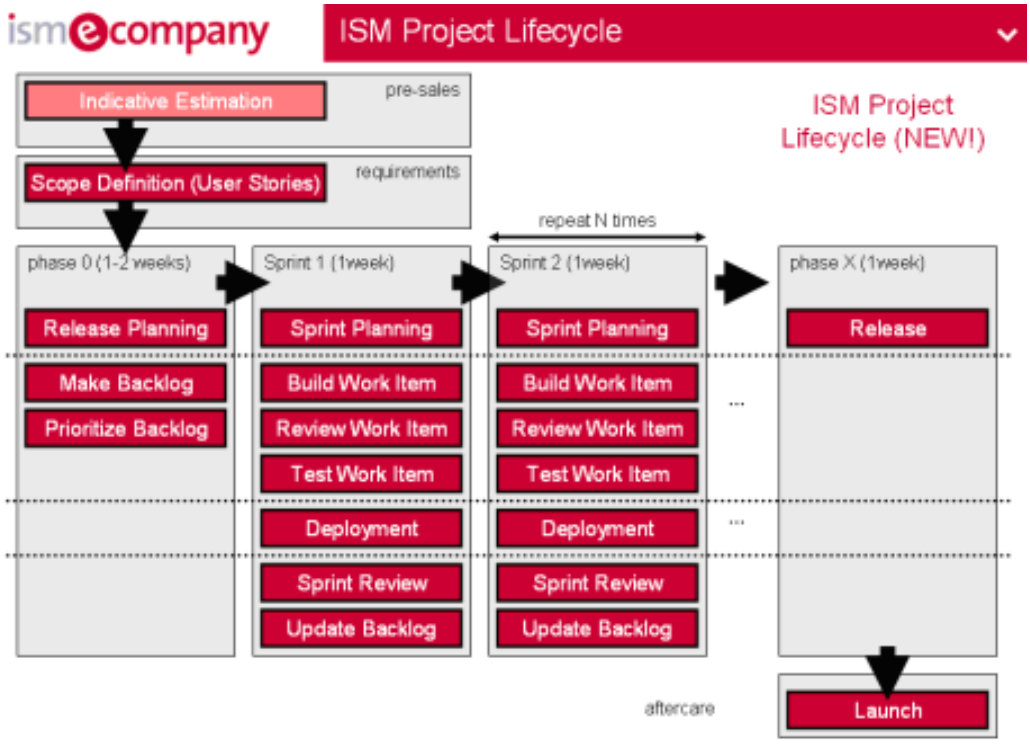


Figure 4.2: Project Life Cycle ISM.

something unclear on how to implement a User Story, the Product Owner can explain it. When all available hours of the team members are used, then the meeting is over. All the selected User Stories/work items will be put on the sprint backlog.

After the Sprint Planning Meeting the team has a week to implement all the User Stories on the Sprint Backlog. The Sprint Review Meeting is held just before the next Sprint Planning Meeting. The team presents the result of the sprint to the Product Owner and he then decides whether the work item meets the acceptance criteria. All rework and not finished work items that sprint will have a higher priority than other User Stories in the next sprint.

4.4 Development Process

4.4.1 Tools

The web applications developed by the teams are written in .Net [36]. The teams use Visual Studio 2008 Team Foundation Server (VSTS) to write their code. This program has a built in source control [37]. This way the code of the developers is available for the whole team.

Furthermore, a Scrum process template add-in is added to the Team Foundation Server (TFS). With this template the team can easily add, assign and update sprints, product backlog items, work items and bugs (see fig. 4.3).

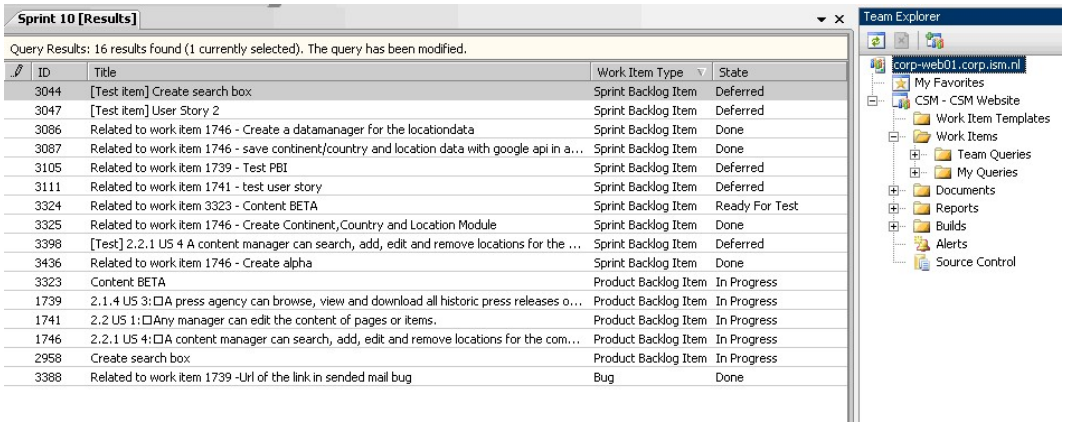


Figure 4.3: Scrum Template in TFS.

4.4.2 Development Process

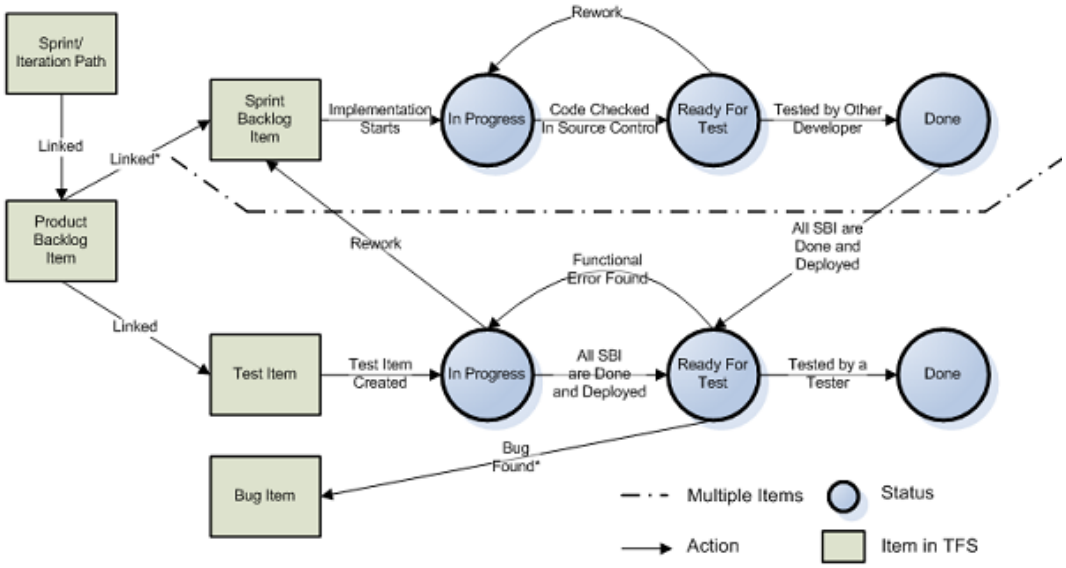


Figure 4.4: Development Process + Testing Process.

During the sprint planning meeting the teams decides which Product Backlog Items are going to be implemented. In TFS these Product Backlog Items are linked to a specific sprint. A developer decides which Backlog Item he his going to implement. He creates (multiple) Sprint Backlog Items and those are linked to the original Product Backlog Item. He has to estimate for each individual Sprint Backlog Item how much time he is going to take to implement the item. In TFS he puts the Sprint Backlog Item on the status ”In Progress”, when he starts with the implementation.

To implement the Sprint Backlog Item, the developer can use the User Story and the

acceptance criteria defined for it. If this is not sufficient to implement the Sprint Backlog Item, he can ask for information from the Product Owner.

When he is finished with his work, he checks in the code and sets the status of the Sprint Backlog Item to ”Ready For Test”. Then another developer in the team has to pick this item up and do a code review and a functional test. When there is rework to be done, the Sprint Backlog Item is assigned back to the first developer. If not, the work item is put on ”Done”. When the work item is done, it can be deployed to the Alpha Server, so it can be tested by the test team and reviewed by the Product Owner.

All the items in the Team Foundation Server are also placed as post-its on a whiteboard (see fig. 4.5). The items in TFS and on the whiteboard must be up to date. When using a whiteboard, the team can easily see what the status of the sprint is.

Figure 4.5: White Board of project team at ISM.

4.5 Testing Process

When all the Sprint Backlog Items of the Product Backlog Item are done, then the User Story itself can be tested. The Quality Manager creates a test item which is linked to the Product Backlog Item. Then the Quality Manager assigns it to a tester from the test team and supplies the tester with the acceptance criteria. The tester tests wheter the user story

meets these acceptance criteria and after that, the tester does some exploratory testing to see wether there are some functional errors which are missing specification items.

In case an error has been found, which applies on the User Story, it is considered as rework. The issue is described in the test item and then assigned to the Lead Developer. He decides which developer has to pick up the rework. For this rework a new sprint backlog item is created.

When an error is found, which does not apply to the functionality of the user stories which are implemented during the sprint, then it is considered a bug. A bug item is created by the tester and it is added to the Product Backlog list. The Product Owner decides when it has to be fixed.

The Team, Business Consultant and the Customer agree on a couple of Beta Deployments during the project. On the Beta Server the customer can test the web application himself. Before the customer tests the Beta website, the test team does a Beta Review of the whole website. All User Stories implemented on the Beta are checked and, in addition, some exploratory testing is performed to find bugs. This process is also repeated when the website is deployed to a Live Server.

4.6 New Development Process

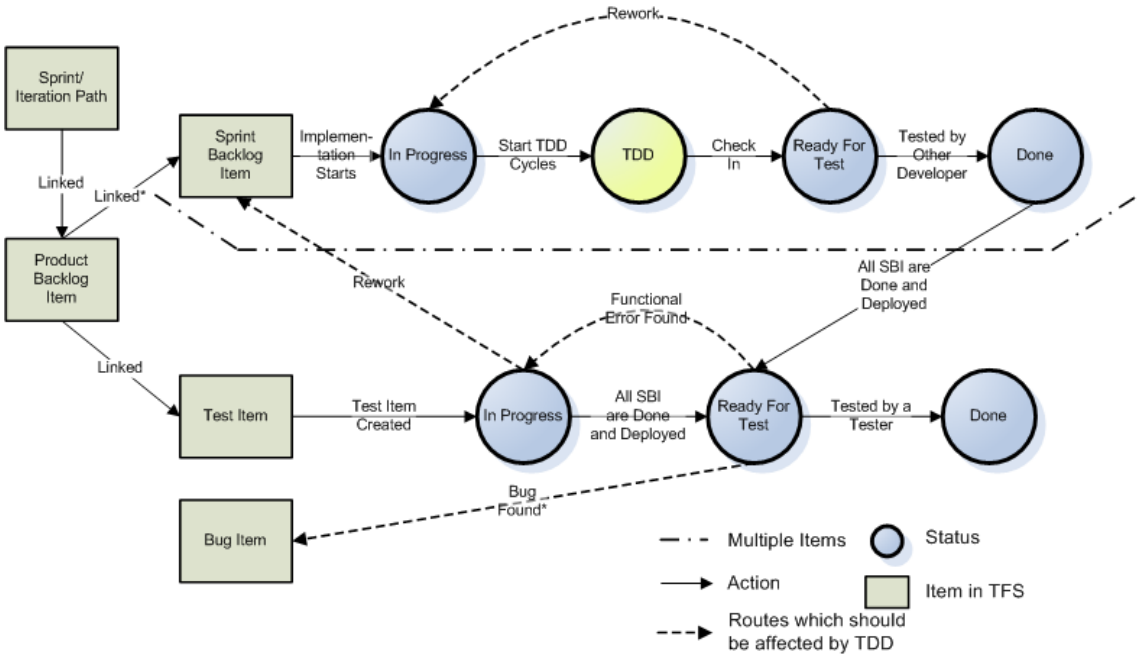


Figure 4.6: New Development Process.

Part of this project was the introduction of TDD into the current development process. As can be seen in figure 4.6, not much is changed of the whole development process. The only thing that changes, is how the developer develops his code. Instead of only writing

code, he has to write also automated unit tests following the TDD cycle described in section 2.3.1. The developer has a User Story and the acceptance criteria as specifications. However, these specifications are very high level, therefore the developer himself has to determine what is expected from his code and extract the unit test cases from that. In most cases, the developer starts with an empty method and incrementally adds functionalities to the method by using a TDD cycle.

Using a TDD approach by the developers, aims at the following results:

1. Lesser rework is found by developers.
2. Lesser functional errors are found by Testers, which result in lesser rework items.
3. Lesser Bug Items are created.

Chapter 5

Sana Software

5.1 Introduction

Sana Software is an independent business unit of ISM eCompany. Within this business unit a set of related products is distributed and developed [14]. The new products/releases of Sana Software are built with .Net C# and are developed by different product development team consisting of mostly developers from ISM Zhitomir (Ukraine). Clients can decide either to use Sana Software to make there own customization or order a web application which uses Sana Software.

When the client chooses for the second option, then the customization of the Sana solution is performed by a project development team of ISM (these development teams are in the Netherlands). Then the project development team is more or less a client of the Sana Software bussiness unit. The different products are developed by following the Scrum methodology and there is every 3 weeks a new release. Depending on the changes in the new release, the clients can decide to update their own versions.

5.2 Solutions

Sana Software has three categories of solutions which can contain different subsolutions. The three categories are: Content Management System Solutions, Ecommerce Solutions and Elearning Solutions. These categories are all an own business unit and are developed completely independent from each other.

We will only focus on the solutions Sana Sites (a CSM Solution) and Sana Commerce Live (an Ecommerce Solution), because these frameworks are used in the case study.

5.2.1 Sana Sites

Sana Sites is categorized as a CMS Solution. This is a software solution for online communication. With this solution customers can easily manage and publish their websites. This can be done with a WYSIWYG editor (see figure 5.1).

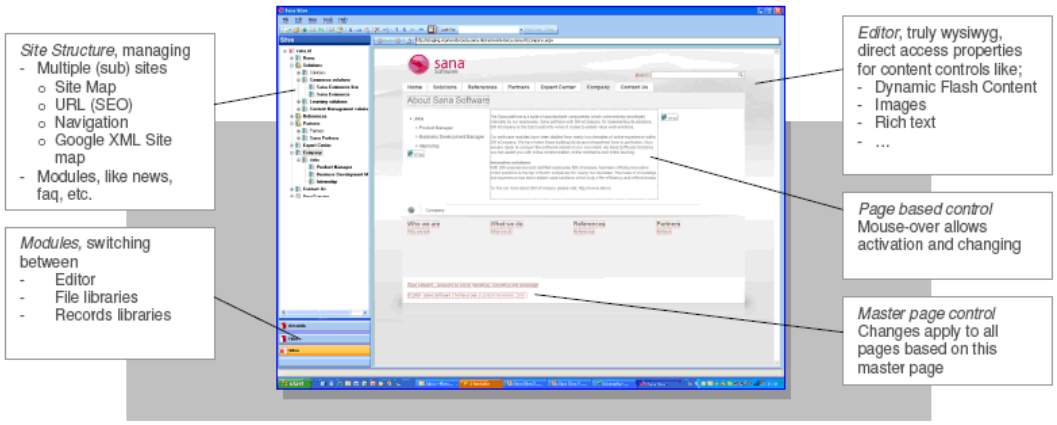


Figure 5.1: Sana Sites Editor

Sana Sites consists of a client-side application which communicates with a server-sided application through web services. Customers, who want to manage their websites, can install a standalone application on their pc or start the editor from a website.

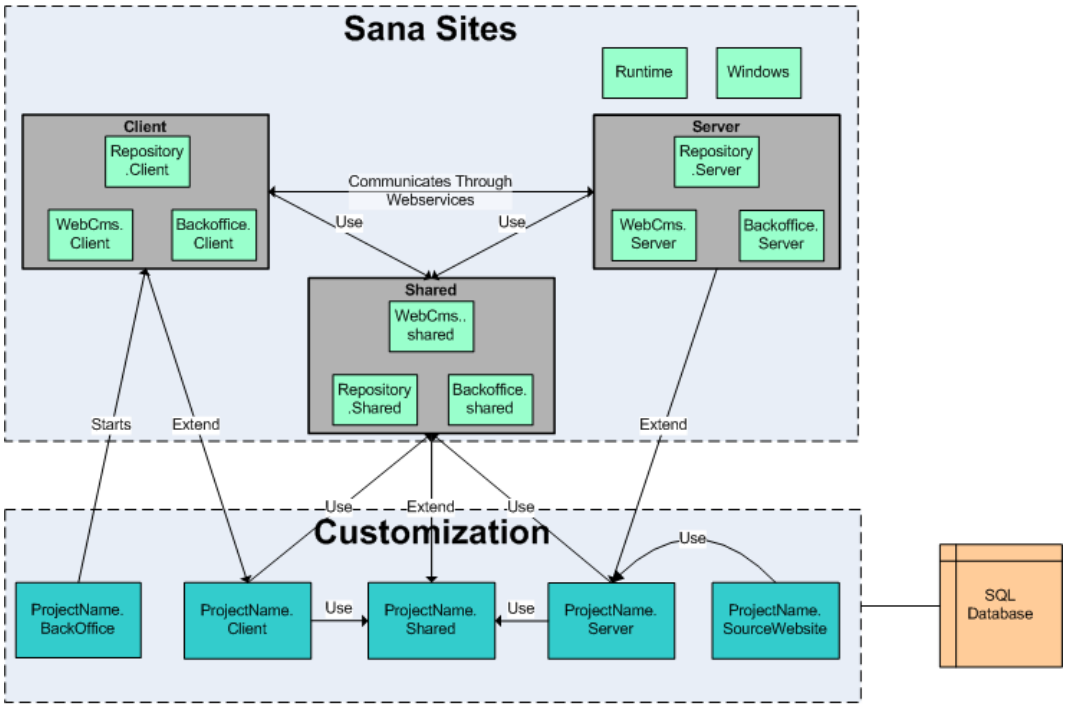


Figure 5.2: Sana Sites Architecture

Sana Sites is build in C#. Figure 5.2 shows an overview of the architecture of Sana Sites. Sana Sites is separated in three main packages, which are divided in smaller packages:

Server
This package runs on the server side. The Server package provides methods for the client to manage the back office and the site structure. It also handles all the calls to the database.

Client
This package runs on the client side. It presents the user with winforms to manage the website. It communicates with the server through web services.

Shared
Contains data used by both the server and the client. Therefore it is present at the client side as on the server side. It contains Type Metadata, Properties, Constants and Classes used by both Server and Client.

WebCms
This package controls all functionality for managing the site structure.

Back Office
This package controls all the functionality for managing the back office (like creating or editing records, channels etc).

Repository
Gets items from the database and provides ways for other classes to access these.

5.2.2 Sana Commerce Live

Sana Commerce Live is categorized as an Ecommerce Solution. With the Ecommerce Solutions customers can easily manage their webshop. Customers can choose from 2 solutions: Sana Commerce and Sana Commerce Live. Sana Commerce is only a webshop and can be linked to the customer's financial and logistical software. Sana Commerce Live is a package which delivers a webshop which is linked to Microsoft Dynamics NAV (also known as Navision). The customer can handle the stock, pricing, discounts, business rules and order handling at one central place.

On the frontend (registered) users can search, view and buy products. Order handling is all being carried out by Navision. The customer who ordered the website can log in on the back office with their internet browser. The back office has also a Microsoft Dynamics look and feel (see fig. 5.3). In the back-office the customer can edit products and content which are shown in the frontend. Furthermore, registered users can be managed.

Figure 5.4 shows the architecture of Sana Commerce Live. The Sana Commerce Live framework is developed by the product development team. The whole SDK is made available for the project development team which can customize everything outside the Sana Commerce Live framework. Sana Commerce Live consists of the following components:

SCL Startersite: Is the frontend where registered users can order products.

SCL Backoffice: Is where admin can manage the frontend.

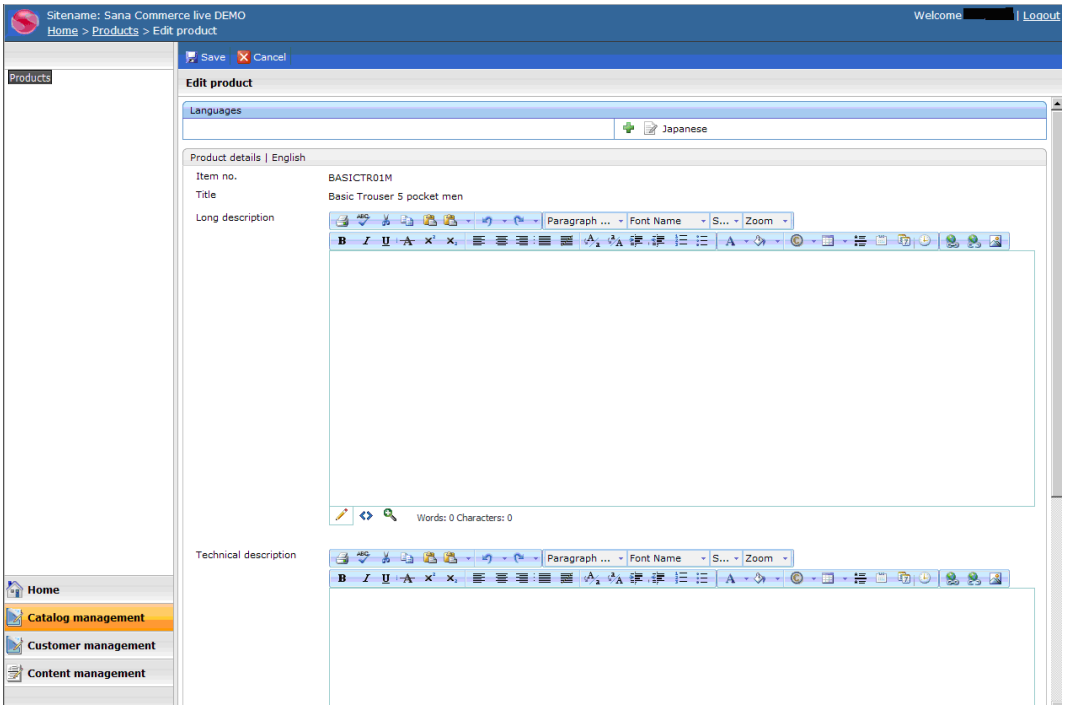


Figure 5.3: Sana Commerce Live Editor

SCL Business facade: Handles the business logic and gets data from the Navision and the SQL server. The project development team can implement their own business logic by placing a component between the SCL Business facade and the web applications.

SCL Web Business Layer: Handless all business logic related to the webshop (for example: sessions, users, masterpages, sitecontext etc).

SCL Navision dataprovider: Handles the calls to the Navision Application Server. They use the Windows Communication Foundation Service to communicate with each other.

SCL Content dataprovider: Handles all the content of the web applications (text, images, flash etc).

This whole solution is unit tested with VSTS test framework.

5.3 Customization of a Sana Solution

Figure 5.5 describes globally the architecture of customization projects. The architectures of the various Sana solutions differ from each other, however the customization projects works very similar. The developers of the project development team use the libraries of

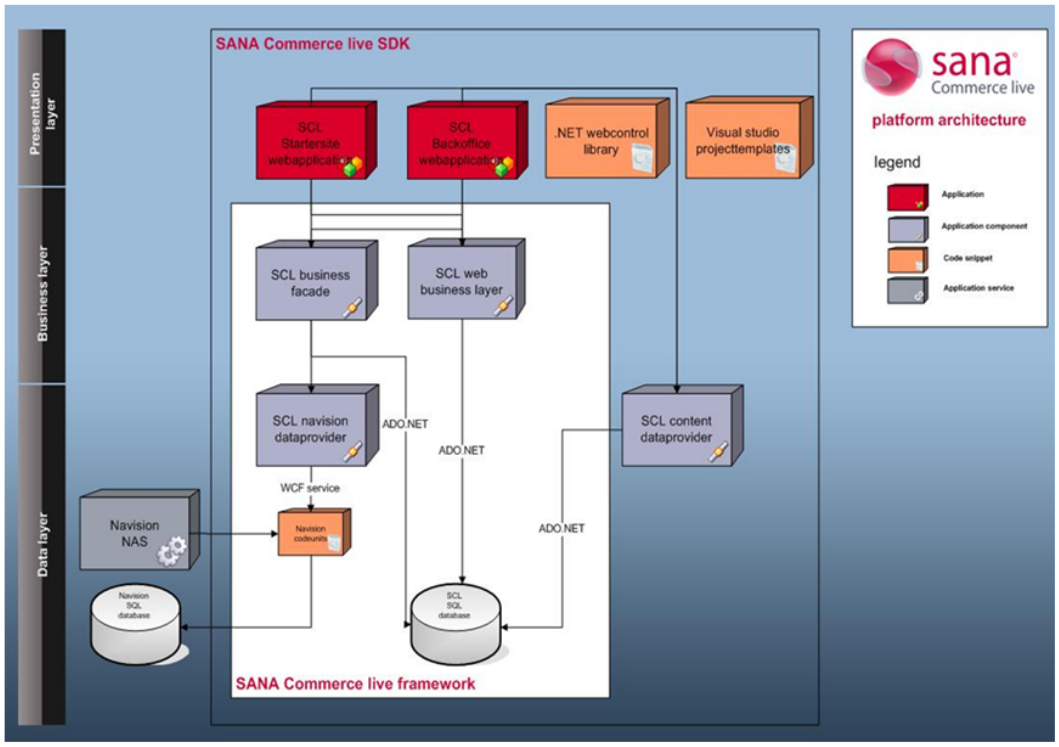


Figure 5.4: Sana Commerce Live Architecture

the Sana Solution to build their own customization and web pages. The libraries contain different C# classes which the developer can use or extend.

In case of a new version of the Sana Solution, the project developers simply update the libraries of their customization project. Furthermore, the project developers have access to the source code of the Sana Solution. With these files the developers can debug the Sana source code while running their own project. The project developers can not change the source code of the Sana Solution.

Web Pages are mostly .Net pages which contain HTML, Web Controls, User Controls, Master Pages, java script, flash and code behind files. The developers try to put most of the business logic outside the web pages and into the classes which extend or use the Sana Solution. In the config of the customization project a connection is defined to a sql database. The Sana solution can read the config and access the database.

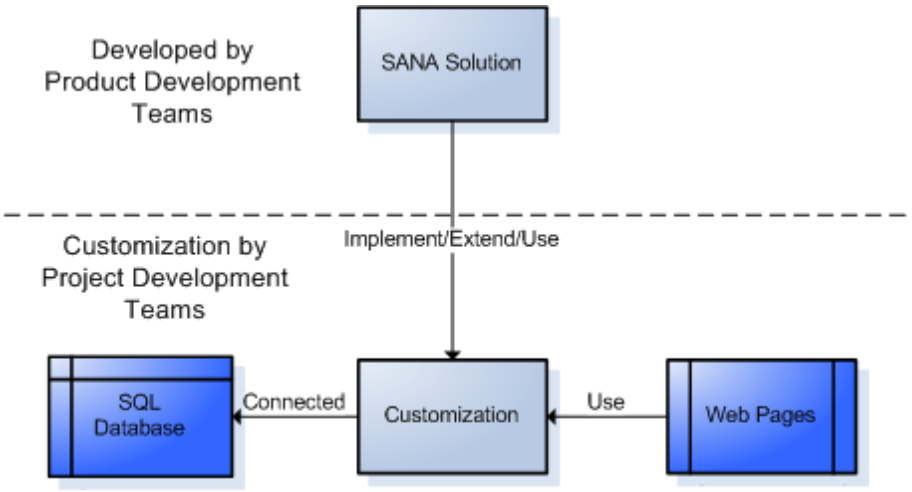


Figure 5.5: Global architecture of a customization project.

Chapter 6

Case Study

6.1 Case Study Goals

The objective of the case study is to determine the cost-effectiveness of a TDD approach in the projects of ISM. This will be assessed by comparing a project of ISM were no TDD approach is used with two projects of ISM in which a TDD approach is used. Additional information is gathered by interviewing the subjects of the case study.

6.2 Characteristics

The introduction of TDD in two projects, handled by two different project development teams, is going to be monitored. Both teams do not have any experience with TDD. The characteristics of the teams and the project they are working on are shown in table 6.1.

Table 6.1: Characteristics of the Case Studies.

ID	**Nr. of Developers**	**Sana Solution**	**Level of Developers**	**Unit Test/ Mocking Framework**	**Code Size**	**Duration Study**
1.	2	Sana Commerce Live	Medior/ Junior	VSTS Unit Tests/ Rhino Mocks	132.427 LOC	3 weeks
2.	3	Sana Sites	Junior/ Medior/ Medior	NUNit/Rhino Mocks	200.946 LOC	3 weeks

The progress and the results of these two projects will be compared with another project of ISM, in which the developers did not use TDD, to determine the effects of using TDD. This project has approximately the same characteristics as the projects which are used in our case study (see table 6.2).

Table 6.2: Characteristics of Project without TDD.

ID	Nr. of Developers	Sana Solution	Level of Developers	Code Size	Length of data gathering
1.	2	Sana Sites	Medior/ Medior	182.212 LOC	6 weeks

6.3 Cost-Effectiveness

6.3.1 Cost and Effects

The only cost involved for using TDD we identified is the overhead during development for the developer. The developer has to create test cases before he starts developing. Creating test cases is an activity which did not take place in the old situation at ISM. Therefore the time spent on creating, updating and running these test cases creates extra overhead for the project.

To determine whether TDD is cost effective in the new situation, we have to compare it with the old situation where no TDD is used in the development process. The major cost of a customization project at ISM is the time developers spent on the project. Therefore we want that the developers only spend time on building things and not on fixing bugs or doing rework.

All the effort that the developers spend on bugs and rework can be considered as extra costs, which should be avoided as much as possible (for example with TDD). In other words we want the internal failure costs [42] as low as possible. Internal failure costs are all the costs of rework, bugs and retesting before the product is shipped to the customer. This case study is only going to focus on the costs of rework and fixing bugs.

One of the assumed effect of TDD is better code quality. As mentioned in section 2.3.2, TDD should reduce the defect injection and bugs should be quicker found and fixed. Therefore the effects of using TDD in the projects at ISM should be:

1. Developers have to spend less time on rework.

2. Developers have to spend less time on fixing bug items.

6.3.2 Extra Costs Old situation

We calculated the time spent on rework plus time spent on fixing bugs each sprint in a project of ISM, in which the developers do not use TDD. This amount is divided by the total hours spend by the developers on relevant sprint backlog items and multiplied with 100 to get the percentage of time spent on rework and bugs. With relevant sprint backlog items is meant items which involve developing code. Items like ”creating alpha server” are excluded, because these items can not be tested by testers or reviewed by other developers.

Formula 6.1 is used to calculate the percentage of time that the developers spend on rework and bugs in a project in which the developers do not use a TDD approach. The costs are not calculated in monetary values, because all projects at ISM has different cost structure depending on commercial agreements made with the customers. Therefore comparison between projects can only be made on hours spent.

$$ExtraEffortP(x) = \frac{\sum_{x=0}^{n}(REWORKtime(x) + BUGtime(x))}{\sum_{x=0}^{n} TotalHours(x)} * 100 \tag{6.1}$$

$x =$ Relevant work items from a project developed without TDD.

$n =$ The amount of relevant work items.

$REWORKtime =$ Hours spent on rework of a item.

$BUGtime =$ Hours spent on fixing bugs.

$TotalHours =$ Hours spent on work item.

$ExtraEffortP =$ Percentage of time spent on rework or bugs during a project.

6.3.3 New Situation

When using TDD, the overhead of creating/updating/running test cases will contribute to the effort put in a project. However, the developer should spend less time on rework and bugs (see section 6.3.1). Therefore the extra effort of a project can be lower in the new situation, making TDD cost-effective. The effort of using TDD is calculated with formula 6.2. The overhead of TDD is going to estimated by the developers (see section 6.4).

$$TDDEffort(y) = \sum_{y=0}^{n} OVERHEADtime(y) \tag{6.2}$$

$y =$ Relevant work items from a project developed with TDD.

$n =$ The amount of relevant work items.

$OVERHEADtime =$ Overhead of TDD while working on a work item.

$TDDEffort =$ Total hours of overhead created by TDD in a project.

Then the effects of using a TDD approach is calculated. First the amount of extra effort in a project which uses a TDD approach is calculated with formula 6.3.

$$ExtraEffort(y) = \sum_{y=0}^{n}(REWORKtime(y) + BUGtime(y)) \tag{6.3}$$

$ExtraEffort$ = Total hours spend on rework and bugs.

The effect of TDD should be that extra effort should be lower. Therefore the benefit of using TDD is shorter time spent on extra effort. But first we have to calculate how much time supposedly would be spent on extra work when not a TDD approach is used. In formula 6.1 the percentage of time on extra work in a project without TDD is calculated, so that percentage can be used to calculate how many hours that would be in the projects which use TDD.

$$ExtraEffortV(y) = \frac{\sum_{y=0}^{n} DEVELOPMENTtime(y)}{100 - ExtraEffortP(x)} * ExtraEffortP(x) \tag{6.4}$$

$DEVELOPMENTtime$ = The time spent on developing a work item before it has the status "Ready for Test".

$ExtraEffortV$ = The hours supposedly spent on ExtraEffort when not a TDD approach would be used.

With this estimation of extra effort when no TDD approach is used, we can calculate how many hours are saved (or gained) when using TDD (see form. 6.5).

$$TDDBenefit = ExtraEffortV - ExtraEffort(y) \tag{6.5}$$

$TDDBenefit$ = Hours saved or gained in a project by using TDD.

With the costs (in our case this is the effort spent on TDD) and benefits calculated it can be determined whether the practice is cost-effective by determining the Return of Investment. We use the formula suggested in [39] [46] (Benefit - Cost / Cost). Cost is replaced by TDDEffort (see formula 6.6).

$$ROI = \frac{TDDBenefit - TDDEffort}{TDDEffort} \tag{6.6}$$

In [12] they also use the investment cost to calculate the ROI. We did not calculate the investment costs, because we think that on the long run the investment costs would not have any significance influence. The only investment cost we identified was the training of the developers and that did not take more than one day. In addition, the training was performed by the developers on special "academy" days. Every developer at ISM has a fixed number of work days, on which he can study new practices. Therefore the hours for the training were not included in the projects of the case study. The tools used for writing/running the test cases and for mocking objects are freeware, or ISM had already a license for them.

The ROI is the amount of time saved after investing time in TDD. If the ROI is greater than or equal to zero, then the practice is cost effective. Because if the ROI is zero: no time is saved, but also no time is lost and in return a regression test suite is build, which can be used for future RFC's.

We make here an important assumption: That the DEVELOPMENTtime is going be relative the same in all the projects of the case study. When the developers take a longer time to put work items on the status ”ready for test” with TDD, that extra time is going to be considered as ”Overhead” and therefore stays DEVELOPMENTtime relative the same. We do not think that developers will develop items quicker then before with TDD, because the developers have to write/update/run test cases (which they did not do in the old situation). Therefore if the extra effort on fixing bugs and rework goes down when using a TDD approach and it compensates the overhead created by TDD, then the developer spent less time on the project. For example (this example is completely fictive):

- In old situation:

Building a Press Release Manager take 18 hours. 10 hours developing + 8 hours rework. $ExtraEffort(x) = \frac{8}{18} * 100 = 44.44\%$.

- In new situation:

Building a Press Release Manager take 12 hours developing + 4 hours rework. Then the overhead is 2 hours, because in the first situation (without TDD) the developer took about 10 hours to develop the same item:

$TDDEffort(y) = 2$
$ExtraEffort(y) = 4$
$ExtraEffortV(y) = \frac{10}{100-44.44\%} * 44.44\% = 8$
$TDDBenefit = 8 - 4 = 4$
$ROI = \frac{4-2}{2} = 1$

From the ROI can be concluded that for every hour invested in TDD, you gain 1 hour. In the new situation 2 hours was invested in TDD and it cost 2 hours less to complete the Press Release Manager.

In this example we could also conclude that the practice is cost-effective by looking at the total hours spent on the item. This is not possible in the case study, because the work items in each project differ too much from each other and the workload is different.

6.4 Case Study Setup

Before the developers started with using a TDD approach, a hands-on-lab was designed for the developers. In these labs, the developers are introduced to the practices of TDD, unit testing and mocking objects. These documents were made available on an internal website of ISM.

We studied how to setup the unit tests in a customization project. One of the reasons why it is hard to write unit tests for a customization project is the dependencies on the Sana Software. Another reason is that some business logic which must be tested is in the code behind pages of the web pages. We studied what the best way is to mock the different

entities of the project (Sana Software, database connection, site context etc). Furthermore, we researched what the best way is to configure the unit tests in a customization project so everything is accessible and can be tested.

These findings were bundled in a document and made available for the developers (see appendix B for an example). Furthermore, some unit tests were written by us different customization projects. The developer used these tests as real life examples and reused the code.

For our own data gathering, three fields were added to the Sprint Backlog Item (SBI) of the process template of visual studio (see Figure 6.1). In the first field ”Unit Tested” the developers can mark if the sprint backlog item is developed using a TDD approach. The second field ”Time Spent” is the total time (in hours) the developer spent on the item before it sets the status to ”Ready for test”. In the third field they have to provide an estimation on how big the overhead was when using a TDD approach. In figure 6.1 the developer spent four hours to implement the item and he thinks the overhead created by TDD was one hour.

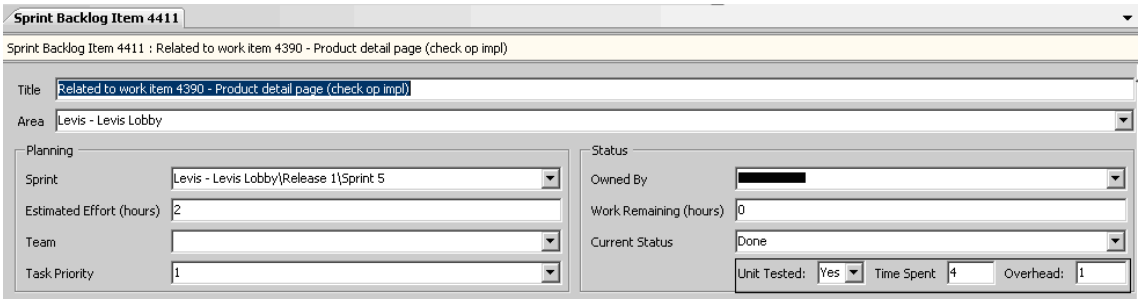


Figure 6.1: Edited Sprint Backlog Item

The quantitative data was gathered from the projects by monitoring the process template in Visual Studio. Every day a snapshot was made of the status of the sprint and put in an excel sheet. After the end of the sprint the following data is collected:

1. Hours spent by the developers on the project this sprint.
2. Hours spent on rework this sprint.
3. Hours spent on bug items this sprint.
4. Hours spent on work items using TDD.
5. Hours overhead is created by using TDD.

With this data we are going to determine whether the practice TDD is cost-effective.

6.5 Extra Effort of Project without TDD

The metrics of six sprints of a similar project as the ones in which the developers were using TDD was gathered. This project was a customization of Sana Sites and with two

developers who took part in the case study. Because the project has approximately the same characteristics as the projects in which TDD was introduced, we can assume that the data from this project can be used to be compared with the data from our case studies.

The cost percentage was calculated with formula 6.1, defined in section 6.3.2 and present the results in Table 5.

Table 6.3: Metrics of Project without TDD.

6 Sprints:	No TDD used	Hours	Percentage %
	Total Hours	247	
	BUGtime	2.50	
	REWORKtime	52.50	
	ExtraEffort	55	22.27

As can be seen in Table 6.3, the developers spent 22.27 percent of the time working on rework or fixing bugs.

6.6 Quantative Results

6.6.1 Quantitative Results Project 1

In this section the results of the introduction of TDD in the project with ID. 1 in table 6.1 is presented. The metrics of this project was collected from VSTS and are presented in table 6.4.

Table 6.4: Metrics of Project 1.

Sprint	**TotalHours**	**TDDEffort**	**ExtraEffort**	**ExtraEffortV**	**TDDBenefit**	**ROI**
Sprint 5	51.25	5.50	4	14.68	10.68	
Sprint 6	6.25	0.50	0	1.79	1.79	
Sprint 7	74.75	0.50	12	21.41	9.41	
Total:	132.25	6.50	16	37.88	21.88	2.37

Table 6.4 shows that the ROI is 2.37, which means that investing 1 hour in TDD results in 2.37 hour less work on bugs and rework. However, when we look at the effort that is put in TDD, it is hard to conclude that the reason less time is spent on bugs and rework is due to TDD. Metrics were collected from 36 relevant work items during 3 sprints and only 6 work items were implemented with a TDD approach (see table 6.5).

It can be concluded from table 6.5 that overhead created by TDD is around 31.03%. The developers take a longer time to implement the work items with a TDD approach. That percentage is something what we already expected, because TDD does not replace any

Table 6.5: Overhead created by TDD in Project 1

Sprint	Total hours spent on Work Items with TDD	Number of Work Items	TDDEffort	Overhead on Work Item	Overhead on Project
Sprint 5	14.55	4 of 13	5.50	37.80%	10.73%
Sprint 6	3.50	1 of 3	0.50	14.29%	8%
Sprint 7	3	1 of 20	0.50	16.67%	0.67%
Total:	20.95	6 of 36	6.50	31.03%	4.91%

other activities. Probably this percentage will become lower when the developers get more experience with TDD.

6.6.2 Quantitative Results Project 2

In this project the introduction of TDD was a bit more problematic. The developers had a hard time to write unit tests, due to their inexperience and because writing unit test in a customization project of ISM is challenging. They were afraid to not make the deadline of the project. That is why the lead developer of the project team decided to drop the practice and to restart with writing unit tests in a new project. He also wanted to gain more knowledge about TDD and introduce the practice after a knowledge session with the whole team.

The team has started with a new project and started with unit testing their work items following a test-last approach (this means after implementation is done, then the unit test is written). Because the new project uses the code base of the previous project, they wrote in the first sprint unit tests for the sections of the old code they believed could be broken by the new implementations.

Because no TDD was used, we do not get any relevant metrics from this project.

6.7 Qualitative Results

After the gathering of quantitative data, some interviews were conducted among all the developers who participated in the case study. The interview was a survey with nine closed answered questions, but the interviewees could motivate their answers. Table 6.6 presents the results of the survey. In total five developers were interviewed.

As can be concluded from table 6.6, most interviewed developers believe that indeed TDD will positively affect the time the developers spent on bugs and rework. However, writing unit tests was not really cost-effective for the projects subject to the case study, because the developers were still getting used to the practice. Writing a unit test produces too much overhead. The interviewed developers expect that after gaining more experience the overhead will be less and thus the practice of TDD will be cost-effective.

Table 6.6: Survey Results Case Study.

	Questions	**% Persons Agreed**
1.	When using TDD, less bugs are introduced?	80%
2.	When using TDD, there is less rework?	80%
3.	When using TDD, does it takes more time to implement work items?	100%
4.	Do you think 22.27% is representative number for how much time is spent on rework and bugs in a project?	80%
5.	Would the overhead of TDD be compensated, by the less time spent on rework and bugs?	80%
6.	With more experience in TDD, would TDD create less overhead?	80%
7.	Do you think TDD is useful in the projects of ISM?	60%
8.	Do you think TDD is a better practice then a test-last approach?	60%
9.	Are you going to use TDD in future projects of ISM?	60%

The reason for the low number of work items which were implemented with a TDD approach was also because of inexperience. It took too much effort and because of tight deadlines the practice was sometimes abandoned. In addition, in project 1, many of the work items were related to Navision and to design issues, which can not be unit tested. Still most developers were positive about using TDD and they think they are going to use this practice in future projects.

One developer still had doubts about using TDD or a test-last approach. Because there is not much specification available for projects, it will take a lot of effort to come up with the tests beforehand and maybe during implementation the design will totally change. Then the written unit test may be useless and the developer has to refactor the remaining unit tests. However, he believes that writing unit tests is useful for the projects at ISM, but at the moment he rather wants to use a test-last approach. Only one developer thinks that TDD and writing unit tests for projects at ISM is not efficient. It creates too much overhead and it slows the project down. He also believes that most bugs still slip into the code even when using TDD, because with TDD the developer test the known scenarios and bugs slip in through the backdoor.

6.8 Evalution Case Study

Executing a case study in the context of ISM was challenging, because it is not a controlled environment. An example of this is that the teams in the case study were under pressure to make their deadlines. This makes them reluctant to experiment with a new practice and rather postponed the introduction of TDD. They were afraid that introducing a new practice would add to much workload to their current project and therefore they would not make the deadline.

In project 1 the problem was that the project was halted about every week. ISM had to wait for information of third parties or had to wait on decisions of the customer. This made it hard to get good data, which could be compared with other projects. In some sprints, the developers did so little work, that the data in that sprint was not really useful. In addition, these issues make it hard to make a good planning for the case study.

Furthermore, the developers in the project development teams had no experience in writing unit tests at all. Due to this inexperience, when trying to introduce TDD in the development process of the teams, the developers had a lot of things coming at them at the same time: unit testing, mocking, TDD etc. They were not sure how to write unit tests, especially in the sections of the code which were intertwined with the Sana Software and sections which are always hard to unit test (GUI's, code behind pages etc.).

Before starting with the case study, we first researched how the unit tests could be organized in the customization projects and put the findings of this research in different documents. Nevertheless even with the assistance of these documents and training material, the developers needed a lot of support when it came to writing unit tests. Because the lack of experience with unit testing, the developers found especially hard to write a unit test before they start coding. The developers did tend to switch back to a test last approach. First they finished the implementation of the work items and after that they tried to find out how they could unit test their code.

When starting this case study the learning curve of unit testing and TDD was underestimated. It had maybe been wiser to first introduce unit testing and when the developers have enough experience, then introduce TDD. On the other hand, it maybe would have been harder to persuade the developer to let go of his test last approach and use TDD.

Other solutions for this problem could be a more intensive training course or introduce the practice in an internal test project, where there is more room for experimentation. However, these solutions can be hard to set up in an industrial setting. It depends whether the company is willing to invest the time and money into it.

Chapter 7

Threats to Validity Case Study

7.1 Reliability

7.1.1 Correctness of Data

We retrieved data of an old project from TFS, to determine what the extra effort on rework and bugs is in the old situation, when they did not use TDD. With this data we could compare it to the data of the new situation, where they tried to use TDD. Because we did not closely monitor that project we have to assume that the data in TFS is correct. Besides we do not know if this percentage is representative for all the projects at ISM. However, at ISM there are so many different projects it is hard to determine what the representative number for the extra effort is in the projects at ISM. The project we chose had almost the same characteristics as the case study so we can assume the data is in that context representative. Furthermore, as can be seen in table 6.6, most developers, who participated in the case study, think that 22% is a representative percentage.

Nevertheless the data contains a lot of uncertainties. To exclude any special circumstances in a project which could affect the data, more data from different projects could be gathered. Then, we should get a clearer picture about how much time is spent on rework and bugs in the projects of ISM.

7.1.2 Estimations of Overhead

We let the developers estimate how much overhead is created by using TDD. Because this variable is not measured but estimated, uncertainty is introduced. We do not know if the developers have correctly estimated the overhead. However, because not many estimations were given, this is at the moment, not a problem. If more estimations were given, the estimates of the different developers could be compared with each other, to see if there is a big difference between the estimations. Anomalies between the estimations could be found and could be corrected.

7.2 Internal Validity

7.2.1 Selection of Projects

We had to select projects to compare with each other. These projects were chosen because they had the same kind of characteristics: two to three developers, same level of expertise with TDD and developed with the new Sana Solution versions. Still, it is hard to compare the projects, because they are not identical.

Other kinds of customizations are made, which may have different kind of complexities or the developers have different level of expertise with these customizations. Another issue is that the developers are getting more experienced with the new Sana Solutions versions. Due to this experience they may work faster and make less faults. This may influence the comparison of data between the old project and the new projects.

In addition, the external factors of the projects can differ. An experiment performed in an industrial setting is not a very controlled experiment due to the external factors (pressure of management, pressure of deadlines, other activities of developers), which can influence the experiment. For example a project development team could not have sufficient or correct information about a project, which can lead to overhead and faults.

7.2.2 Duration Case study

The duration of the case study was too short for the developers to get good acquainted with TDD and for us to get enough data to prove whether TDD is cost-effective. However, if we look at the academic case studies, the developers/students had much shorter time to get experience with TDD and there TDD produced better results then a test-last approach. Likely the causes are: the complexity of writing unit test in the customization project, the inexperience of the developers with unit testing and the pressure of finishing the project before the deadline.

7.2.3 Process Conformance

To use a TDD approach in the projects of ISM is something what the Development Managers at ISM decided. Therefore it is not necessarily something what the developers themselves think they need in their projects. Using TDD needs a lot of discipline and it would be probably easier if the developers themselves decided to use TDD. This motivation ”issue” may have influenced the data negatively from the case study.

From the interviews, however, we can conclude that most developers where positive about the practice and think that the practice can be useful for ISM. Only some developers were afraid to apply the practice in an already ongoing project.

7.3 External Validty

The results of this case study are not fully generalizable, because of the context in which the case study was conducted and the lack of relevant data. ISM makes customized web applications integrated with an evolving software product, in relative small projects (the

project has an average duration of less than hundred working days and the project team exist from, in most cases, not more then four developers). The introduction of TDD may go smoother in other contexts. Nevertheless, the way we tried to determine if TDD is cost-effective can also be used in different contexts.

Chapter 8

Discussion

8.1 Time and Risk is not taken into account

Following Rico [39], some people think that the only way to calculate the economic value of a software improvement is by a Net Present Value (NPV) analysis. NPV take the time value of money into account. If you invest now an amount of money and over a few years this yields an amount of money, that future money does not have the same value as the money in the present. For example the inflation has to be taken into account. We did not take the Net Present Value into account, because the effects of TDD should directly be visible in the same project. The projects of the case study had only a duration of a few months.

Another factor we did not account for is Risk Discount [17]. If someone takes the Risk Discount into account, then that can mean that you decide to receive less of a ROI in exchange of less risk. Maybe the ROI of an investment is lower, but there is a higher change on a pay-off. Such a Risk Analysis can come into play when there are multiple options to choose from. We have only one option (the usage of TDD). The only risk that ISM has is that using TDD in their projects is not cost-effective. That is something we were investigating with this case study.

8.2 Other Factors which influence the cost-effectiveness

We measured the cost effectiveness by monitoring the time developers spent on rework and bugs in a project. Besides working on rework and bugs, there are maybe also other factors which could be affected by the introduction of TDD and could affect the cost-effectiveness of the practice.

An example of this is the External Failure Costs [42]. These are the costs that arise when a fault of the product occurs at the customer side. TDD should lower the defect injection, so it would be logical that the External Failure costs would be lower when using TDD. This makes the practice more cost-effective. TDD can also affect how RFC's are performed at ISM. When a web application is delivered, the customer can request for small changes. To implement these changes take in most cases not more than one or two days. A benefit of TDD is that you build a regression test suite. During the implementation of the RFC the

developer can verify whether his changes do not break something else in the system. A disadvantage is that the developer might have to rewrite or add some unit tests. This can cause extra overhead in implementing the RFC.

We did not investigate these factors because it did not fit into the scope of our case study. These factors could make the usage of TDD at ISM more or less cost-effective. Some developers already mentioned during the interviews that the regression test suite would be a big help when the project is finished. For example if another team with less knowledge about the project would perform a RFC, they can find out much quicker if they break something.

8.3 Effort versus Monetary Value

The cost-effectiveness was not calculated in monetary values, but how in terms of time/effort saved when using TDD. We could have calculated in monetary values what would be saved or extra costs to use a TDD approach, however every project has different costs. It depends on how many and which developers are assigned to the project and on the agreements made with the customer. In addition, we should have taken in account how many hours are spent by each individual developer. The cost of a developer depends on the level of the developer (junior/medior/senior).

That is why we believed it would be more practical to determine the cost-effectiveness by looking how many hours are saved or generated by using TDD. This number is easier to use to compare with other projects at ISM.

Chapter 9

Summary, Conclusions and Future Work

9.1 Summary

We presented in this thesis the introduction and evaluation of TDD into the development process of the project development teams at ISM. We wanted to research whether costs and effects of TDD are cost-effective in the context at ISM. We researched the development process of ISM and whether TDD could be implemented. Then TDD was introduced into two different projects, but with similar characteristics, of ISM. By the gathering of metrics of the monitored projects and by comparing this data with project of ISM, where no TDD approach was used, we determined the cost-effectiveness of TDD by determining the ROI. Then a survey was conducted among the developers, who participated in the case study, to get more insight in the quantitative data gathered.

9.2 Conclusions

During this research we proved that is technical feasible to write unit test in a customization project, by mocking different objects of the Sana Software. These findings were bundled in different documents and made available for the all the developers. By proving that unit tests can be written for the customization projects, it also proves that TDD can be implemented into the development process of ISM. The unit tests function as the test cases which drives the development forward. The developer can use the already defined acceptance criteria to write the unit tests before implementation. However, these are very high level and thus the developer has to, in most cases, determine himself how the unit test must look like.

The cost of using TDD in context at ISM is the overhead it creates for the developer during implementation. Writing unit tests did not happen in the old development process of ISM and does not replace any other activity. The developer has to design the test cases himself with the unit test framework. In addition, writing unit tests in this context is complicated due to the dependencies with the Sana Software, a lot of GUI's are present in the

code and a lot of functionality is in the code behind pages. Due to these complexities the developer has to write a lot of mock objects to make his unit tests fast and independent.

All the work items which are implemented with a TDD approach during the case study, the developer indicated that he had overhead because he used a TDD approach. In project 1 we monitored that the average overhead was 31%. This means that developers took 31% more time developing items when using a TDD approach than when they do not use a TDD approach to implement work items.

The effects we predicted were that the developers would spend less time on rework and bugs. In project 1 the results were that less time was spend on rework and bugs in comparison with a project where no TDD approach was used. However, we can doubt how reliable this data is. The number of work items implement following a TDD approach was very low (17% of the work items). Consequently it is hard to conclude that due to the usage of TDD in the project less time was spent on rework and bugs. In addition, we do not know if the percentage of time spent on rework and bugs we found in the project without TDD (22.27% of the time developers are busy with rework or bugs) is a representative percentage for all the projects at ISM.

Due to this unreliable data, it is furthermore hard to determine if the practice is actually cost-effective. However, we think if more data is collected of the projects at ISM, it should be possible to determine if TDD is cost-effective by using the approach proposed in this thesis. In our opinion the practice was in the case study not cost-effective, because it took the developers too much time to write the unit tests and because of the low test coverage of the unit tests it does not have a large effect on reducing bugs and rework.

From this case study we also can conclude that more time should be used to introduce the practice TDD at ISM. Most developers pointed out in the interviews that they were positive about writing unit tests and about using TDD. They think due to their inexperience in writing unit tests and their lack of knowledge about TDD, it was not possible to get fast results. The majority of the developers think that when they have more experience with TDD, it would be a useful practice in the projects of ISM. In our opinion this is correct. When the developers get more experience, the overhead of writing unit tests will be lower and this will stimulate to use a TDD approach on more work items. This will positively influence the test coverage of the unit tests and it will result in less bugs and rework. In the future projects most of the interviewed developers are going to use TDD.

We did not prove indisputable that TDD is cost-effective at the context of ISM, but lesson learned from this study can still be used by others. Furthermore, the approach we used to determine the ROI of TDD can be used by ISM to determine whether TDD is cost-effective in their future projects.

9.3 Future Work

The data collected during this case study contains many uncertainties. A lot of the uncertainty is created by the amount of data. During the case study it was only possible to get data from a few sprints and only data of two projects were collected. To get a more representative image of the time what is spent on rework and bugs, data of more projects at ISM

can be collected. To see if TDD is really cost-effective, TDD should be introduced in more projects. In addition, the duration of the new case study should be longer to determine more accurately what the effects of a TDD approach are.

Furthermore, to determine whether TDD is cost-effective, other factors could also be included, like the ones discussed in section 8.2. By studying the effects of TDD on these factors, it could be possible to get a better notion whether TDD is cost-effective.

To remove more uncertainty of the data to be collected, a more accurate approach can be used for recording the hours that developers needs for implementing the work items. Now the developers record themselves how many hours they work on an item and they estimate the overhead created by TDD. For example a recording tool could be used to record how many hours it the developer takes to finish a work item and how long he is busy in the test project creating/running/updating the unit tests for that particular work item.

Other approaches of TDD could be investigated whether they are more cost-effective. One of these is Acceptance Test-Driven Development (ATDD) [3] [35]. Acceptance tests are test written (preferably by the customer) to test whether the User Story meets the acceptance criteria. The Product Owners at ISM already define the acceptance criteria of the User Stories. Therefore acceptance tests can easily be extracted from the acceptance criteria by the Product Owner or by a tester. Then the acceptance tests can be automated by the developers or testers, before the developers start with the development of the work items.

Benefits of using an ATDD approach are that the developer does not have to define the test scenarios himself. In addition, the developer does not have to worry about mock objects and interfaces, because the tests do not have to be independent from external sources. This could remove a lot of overhead created by the more traditional approach of TDD, which was introduced at ISM. A downside of this approach is that it is maybe harder to determine where the bug occurs, because the tests are not independent and it could not stimulate the developer to write better code as a normal TDD approach would.

To make sure that the developers still write unit tests after this case study has ended, the teams should include writing unit tests to their definition of done (DoD) [30]. This is a checklist of criteria, which a work item must fulfill to be considered ”done”. A work item could never be ”done” when no unit test is written for it. When someone’s code is reviewed, the reviewer should also pay attention to quality of the unit tests (are they really independent, are they fast enough etc). Furthermore, there should be a mentor in the team, to which team members could ask guidance about the practices: mocking, unit testing and TDD. It would be logical that this role should be fulfilled by the lead developers.

Another recommendation would be the practice Continuous Integration [13]. With Continuous Integration the code of developers is integrated with each other on a repository. On the repository the solution automatically builds to see wheter it compiles. When the developer checks in his code, the unit tests should automatically be run. This way the team can quickly find out when a test fails and the just checked-in code broke something. Various teams at ISM already use source control in combination with automatic build definitions. The teams whom use TDD should add the automatic running of the unit tests to these build definitions.

Bibliography

[1] P. Abrahamsson, A. Hanhineva, and J. Jlinoja. Improving business agility through technical solutions: A case study on test-driven development in mobile software development. In *Business Agility and Information Technology Diffusion*, pages 1–17. Springer, 2005.

[2] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods review and analysis. Technical Report 478, VTT PUBLICATIONS, 2002.

[3] Johan Andersson, Geoff Bache, and Peter Sutton. Xp with acceptance-test driven development: A rewrite project for a resource optimization system. In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003 Proceedings*, volume 2675, pages 180–188. Springer, 2003.

[4] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[5] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[6] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363, New York, NY, USA, 2006. ACM.

[7] D. Chaplin. Test first programming. *TechZone*, 2001.

[8] Alistair Cockburn. *Agile Software Development: The Cooperative Game (2nd Edition) (Agile Software Development Series)*. Addison-Wesley Professional, 2006.

[9] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[10] DSDM Consortium. DSDM. `http://www.dsdm.org/atern/`.

[11] Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[12] Lars-Ola Damm and Lars Lundberg. Results from introducing component-level test automation and test-driven development. *J. Syst. Softw.*, 79(7):1001–1014, 2006.

[13] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[14] ISM eCompany. Sana software. `http://www.sana-software.com`.

[15] S. H. Edwards. Using test-driven development in the classroom: Providing students with automatic. In *Proc. Int"l Conf. Education and Information Systems: Technologies and Applications (EISTA 03)*, 2003.

[16] H. Erdogmus. On the effectiveness of test-first approach to programming. *IEEE Transactions on Software Engineering*, 31:1–12, January 2005.

[17] Hakan Erdogmus, John Favaro, and Wolfgang Strigel. Guest editors' introduction: Return on investment. *IEEE Software*, 21(3):18–22, 2004.

[18] Martin Fowler. Xunit. `http://www.martinfowler.com/bliki/Xunit.html`.

[19] Boby George. Analysis and quantification of test driven development approach, September 2002. Ph D thesis.

[20] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, New York, NY, USA, 2003. ACM.

[21] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

[22] J. Highsmith and A. Cockburn. Agile software development: the business of innovation. *Computer*, 34(9):120–127, 2001.

[23] Watts S. Humphrey. *Managing the software process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[24] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers, 2004.

[25] L. Williams J. C. Sanchez and E. M. Maximilien. A longitudinal study of the use of test-driven development practice in industry. In *Proceedings of Agile 2007 Conference*, pages 5–14, Washington DC, USA, 2007. IEEE Computer Society.

[26] David S. Janzen. An empirical examination of test-driven development. *SRC Grand Finals Third Place Winner, ACM Digital Library*.

[27] Ron Jeffries and Grigori Melnik. Guest editors' introduction, tdd: The art of fearless programming. *IEEE Softw.*, 24(3):24–30, 2007.

[28] Reid Kaufmann and David Janzen. Implications of test-driven development: a pilot study. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299, New York, NY, USA, 2003. ACM.

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

[30] Henrik Kniberg. *Scrum and XP from the Trenches*. Lulu.com, 2007.

[31] Johannes Link and Peter Frolich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[32] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. In *Extreme programming examined*, pages 287–301, Boston, MA, USA, 2001. Addison-Wesley Longman Publishing Co., Inc.

[33] Lech Madeyski. Preliminary analysis of the effects of pair programming. In *and Test-Driven Development on the External Code Quality, Software Engineering: Evolution and Emerging Technologies*, pages 113–123. Press, 2005.

[34] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Grigori Melnik and Frank Maurer. Multiple perspectives on executable acceptance test-driven development. In *Agile Processes in Software Engineering and Extreme Programming, 8th International Conference, XP 2007, Como, Italy, June 18-22, 2007, Proceedings*, volume 4536 of *Lecture Notes in Computer Science*, pages 245–249. Springer, 2007.

[36] Microsoft. .Net. `http://www.microsoft.com/NET/`.

[37] Microsoft. Source Control for Visual Studio. `http://msdn.microsoft.com/en-us/library/zxd4dfad(VS.80).aspx`.

[38] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[39] David F. Rico. *ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*. J. Ross Publishing, Inc., 2004.

[40] Julio Cesar Sanchez, Laurie Williams, and E. Michael Maximilien. On the sustained use of a test-driven development practice at ibm. In *AGILE '07: Proceedings of the AGILE 2007*, pages 5–14, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[42] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41(8):67–73, 1998.

[43] Stephanie D. Teasley, Lisa A. Covi, M. S. Krishnan, and Judith S. Olson. Rapid software development through team collocation. *IEEE Trans. Softw. Eng.*, 28(7):671683, 2002.

[44] Arie van Deursen. Customer involvement in extreme programming: Xp2001 workshop report. *SIGSOFT Softw. Eng. Notes*, 26(6):70–73, 2001.

[45] Arie van Deursen. Program comprehension risks and opportunities in extreme programming. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 176, Washington, DC, USA, 2001. IEEE Computer Society.

[46] Rini van Solingen. Measuring the roi of software process improvement. *IEEE Softw.*, 21(3):32–38, 2004.

[47] Shane Warden and Jim Shore. *The Art of Agile Development: With Extreme Programming*. OReilly Media, Inc., 2007.

[48] Don Wells. Extreme Programming. `http://www.extremeprogramming.org/`.

[49] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.

[50] O. Hazzan Y. Dubinsky. Measured test-driven development: Using measures to monitor and control the unit development. *Journal of Computer Science*, 3:335–344, 2007.

[51] S. Yenduri and L. Perkins. Impact of using test-driven development: A case study. *Software Engineering Research and Practice*, pages 126–129, 2006.

[52] R. Ynchausti. Integrating unit testing into a software development team's process. *In Proceedings of XP2001 Conference on Extreme Programming*, pages 79–83, May 2001.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ATDD: Acceptance Test Driven Development

Bug: Error which is not related to a user story.

DSDM: Dynamic System Development

Functional Error: Error which is related to a User Story

GUI: Graphical User Interface

ISM: Innovative Solutions in Media

LOC: Lines of Code

RFC: Request for Change

ROI: Return on Investment

SBI: Sprint Backlog Item

SUT: System Under Test

TDD: Test-Driven Development

TFS: Team Foundation Server

VSTS: Visual Studio Team System

WYSIWYG: What You See Is What You Get

XP: eXtreme Programming

Appendix B

Unit Testing Sana Commerce Live

B.1 Introduction

”Unit Testing” tests the smallest testable part of the source code. Usually the unit test exercises a particular method in a particular context. It is important that the unit tests are independent from each other or from other external data sources.

With unit testing each small functionality in the source code is validated. Unit testing provides a strict, written contract that the code must satisfy. It forces the developer to think about his code and keeps his methods testable and highly decoupled from each other. Unit tests can run directly after a small functionality is finished. This way bugs can be found quicker and are then easier to fix. Therefore when using unit testing, fewer bugs are introduced.

An extra benefit of unit testing is that it positively influences the program understanding (comprehension). 40% to 60% percent of the time spent on software maintenance is used on program understanding. Unit tests can explain the behavior of the code which is tested and therefore developers can use the unit tests to see how the code should behave.

When using unit tests developers have more confidence to add changes to their program. This because when the program is correctly unit tested, the tests have a high code coverage and the methods are highly decoupled. When a change is implemented, it is easy to check if everything still works properly by running the unit tests.

Another practice, which is closed related to unit testing and also highly recommend by the agile community, is Test-Driven Development. This practice believes that the best way to write unit tests is before implementation. By using this practice the programmer have to think about what really is expected from his code. He first writes a unit test and then writes the code to make the unit test pass. When the unit test passes, additional unit tests can be added and the cycle starts again. This repeats until no more additional unit tests are necessary.

This document will give some general information and links about unit testing and some examples on how to setup unit testing for a Sana Commerce Live solution and how to mock different objects in the code. This document is not a complete tutorial about unit testing/TDD.

2 HandsOnLabs are written for that purpose:

I:Development.Team4NL.Documentatie.HandsOnLab - Test Driven Development

B.2 Unit Testing

B.2.1 Used Tools

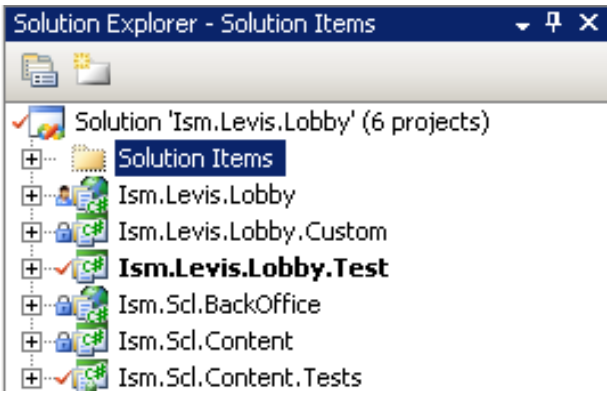
For Unit Testing the Sana Site Solution described in this document we used these programs:

- VSTS Unit Testing Framework
- Rhino Mocks (http://ayende.com/projects/rhino-mocks.aspx)

There a lot of other programs available like: NUnit, MbUnit or NMock, but it is up to the team to decide which tools they are going to use. We choose these tools because the VSTS Unit Testing Framework and Rhino Mock are also used for unit testing SANA Commerce Live.

B.2.2 Setting up Unit Tests

The best thing to setup the unit tests is to create a separate project for it or even a new solution. In the new project you can use the same directory structure as the source code so you can quickly find our test classes. But how to setup the unit tests is a decision entirely up to the team.



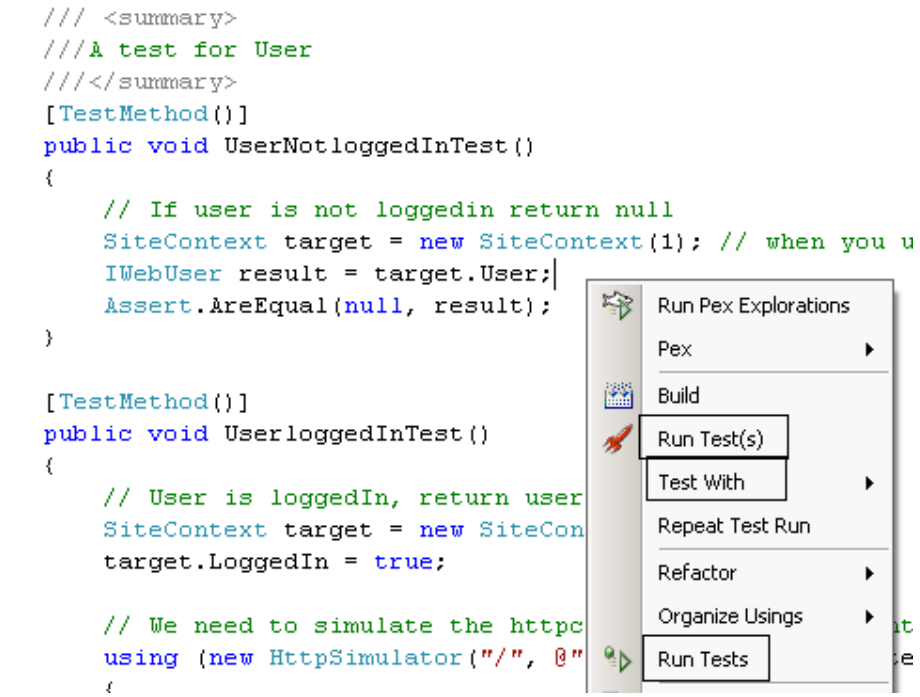
Add a reference to Rhino Mock. Add references to the different projects of the source code. If you want to test a functionally from ”class A” then create a test class with an appropriate name like (TestA.cs or AFixture.cs) and put it in the corresponding folder of the test project. Now you are ready for writing unit tests. For how to setup unit tests with VSTS see:

http://msdn.microsoft.com/en-us/library/ms379625.aspx

Here a quick example of a unit test:

```
[TestMethod]
public void Test_Add()
{
      int result = calculator.Add(2, 2);
      Assert.AreEqual(4, result);
}
```

With the object Assert you can test if the method returns the desired result, if not the unit test will fail. To start the unit test, right click on the code and select ”Run Tests”.

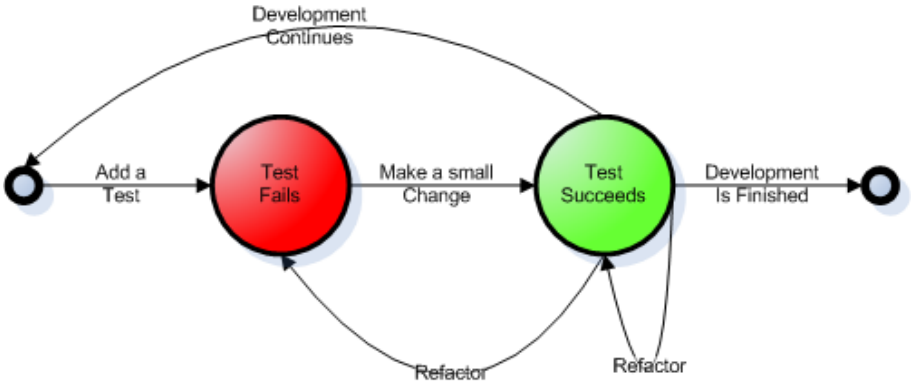


(First 2 options are from the testdriven.net plugin).

B.2.3 Test Driven Development

Test Driven Development is test-first approach. This software design practice is closely related with unit testing, because the developer writes first a unit test before he starts with implementation. This unit test behaves as a written specification for the code what has to be implemented. TDD has a cyclic approach where in small steps additional functionality is added. This functionality is completed when no additional unit tests have to be added:

The TDD cycle consist out 3 phases:



1. Red Phase.

In the Red Phase the developer writes a test for the new feature he wants to implement. An important rule in TDD is: ”If you can not write a test for what you are about to code, then you should not even be thinking about coding”. All old tests should succeed, except the new one.

2. Green Phase.

In the Green Phase the developer makes the smallest possible piece of code to make the test succeed. This way the developer makes small incremental steps of development. If he makes bigger steps, there is a higher change that faults slip in the code.

3. Refactoring.

When the new test succeeds, we can clean up the code in the refactoring phase. By running all the tests we can ensure that our refactoring activities do not break anything. Now if additional tests can be written, the cycle starts again. If this is not the case, then development stops.

The benefits of writing unit tests before implementation:

- You will get high test code coverage.
- Lower fault injection.
- You have more confidence that the code act following your specifications.
- Bugs are quickly discovered.

This is practice is explained in the book: ”Beck, K. Test-Driven Development by Example, Addison Wesley, 2003”. Some useful links:

http://www.agiledata.org/essays/tdd.html
http://www.codeproject.com/KB/dotnet/tdd_in_dotnet.aspx

B.2.4 Unit Testing

In this section, we will explain how to unit test methods, which are hard to test. Please try avoiding these situations as much as possible.

B.2.4.1 Protected Methods

Sometimes it is maybe necessary to test a protected method. Before you do this, try to figure out if the method can not be made public or it is really necessary to unit test the method. If this is not the case you now have the problem that the method can not be called by other unrelated classes. This can be solved by making a derived class which has public methods (but this solution will increase the maintenance of the code) or by using the System.Runtime.CompilerServices.InternalsVisibleTo attribute.

Add the following line to AssemblyInfo.cs of the target project which contains the protected method:

$[assembly : InternalsVisibleTo("TestProjectAssemblyName")]$

Add to the protected method the type ”internal”:

protected internal Boolean method()

When the target assembly is signed with a public key, you can get the following error:

Error 2 Friend assembly reference 'TestClasses' is invalid. Strong-name signed assemblies must specify a public key in their InternalsVisibleTo declarations.

Then you have to sign your test assembly (right click on the test solution and goto signing). After that extract the public key with sn.exe:

sn -p Foo.Bar.Test.snk Foo.Bar.Test.PublicKeyOnly.snk sn -tp
Foo.Bar.Test.PublicKeyOnly.snk

Then you have to add the public key to the InternalsVisibleTo attribute in AssemblyInfo.cs of the target project:

```
[assembly: InternalsVisibleTo("TestClasses,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100
3f15831f528b047ba4d30a98c3d92db616866a8922f68b31e5f1e95b270947f49255e5
76d8a594eff9e1ec066a2a309f712acb196e218f12e93a6ec89d51524453d5bcc4e735e5
dbec93937efd4e2381763c88fc261ff73790da980e63297970746780c8977794d815c3e9
d359bce12455e1343bc1e3bbdb09b82180f8a566d5")]
```

After building you now can call the protected methods in your test project. For more information see:

http://blog.tylerholmes.com/2008/04/unit-tests-and-internalsvisibleto.html

B.2.4.2 Private Methods

Just like testing protected methods, before you try to unit test them, consider if it is really necessary to test them or if it is possible to transfer the functionality to other public methods or to make the method public. Unit testing private methods is on going debate, but is possible with reflection. This example show how to unit tests a private method of a PressReleaseRecordManger:

private static string GetRecordNameById(Guid id)

You can test a private method by using reflection. You have to add "using System.Reflection" to your TestClass. Here an example how to use reflection for unit testing the private method:

private static string GetRecordNameById(Guid id)

You can test a private method by using reflection. You have to add "using System.Reflection" to your TestClass. Here an example how to use reflection for unit testing the private method:

```
// Using reflection to call private static method
Type type = typeof(PressReleaseRecordManager);
// if method is not static: BindingFlags.NonPublic |
      BindingFlags.Instance
MethodInfo method = type.GetMethod("GetRecordNameById",
      BindingFlags.NonPublic | BindingFlags.Static);
// Pass parameters in an object array
// Warning: If the class isn't static, first parameter must be instance
      of the class where the private method is called
String result = (String) method.Invoke(null, new Object[] {id});
Assert.AreEqual("test1", result);
```

If the method was not static the example would look like:

```
// Using reflection toen call private method
PressReleaseRecordManager instance = new PressReleaseRecordManager();
Type type = typeof(PressReleaseRecordManager);
MethodInfo method = type.GetMethod("GetRecordNameById",
      BindingFlags.NonPublic | BindingFlags.Instance);
String result = (String) method.Invoke(instance, new Object[] {id});
Assert.AreEqual("test1", result);
```

B.2.4.3 Code Behind

Unit Testing ”Code Behind” of webpages can be tricky, but can be done. You have to make the attributes ”protected internal” and make also the functions public or internal. Functionality in the Page_Load function can be defined in a separate method.

But there are better frameworks for testing the webpages like Selenium, Watir and Sahi. They can simulate the interaction between the GUI and the end user. Even Selenium and Watir can be integrated with your Unit tests.

B.2.5 Unit Testing Sana Commerce Live

B.2.5.1 Setup

Before you can start writing unit test, the mappings have to be performed before the unit tests are runned. This can be done with the attribute: AssemblyInitialize. This method is then called before all unit tests are started in the same assembly.

Put all the mappings in a method. Nicest way to do this would be if the mappings are placed in a static method in a custom project. This way the mapping method can be called from Global.asax.cs and from the unit tests. If something has to be updated it can be done at one place. If the method is called InitObjects(), then your AssemblyInitialize method should look like this:

```
[TestClass]
public class InitializeTestAssembly
{
    [AssemblyInitialize()]
    public static void AssemblyInit(TestContext context)
    {
        SomeObject.InitObjects();
    }
}
```

These mapping are necessary for example for initializing the providers.

B.2.5.2 Mocking Providers

The providers in the Sana Commerce Live framework handle the calls to the databases. You can decide to make actual calls to the database and therefore you can test the integration between the program and the database. This way you also test the business logic of Navision. A downside of this is that it will make your tests slowly and you make your test cases dependent of the data in the database.

This is also a decision the team has to make if they are going to mock the calls to the database. It is also possible to make 2 groups of tests (unit tests and integration tests).

In both cases you have to add an app.config to your unit tests project, which should look very similar to your webconfig. In our example we are going to make our own fake customerProvider. This how normally a CustomerProvider is set upped in the config:

```
<customer defaultProvider="StarterSite1NavisionCustomerProvider">
        <providers>
          <clear/>
          <add name="StarterSite1NavisionCustomerProvider"

type="Ism.Scl.Navision.Provider.Customer.NavisionCustomerProvider,
Ism.Scl.Navision, Version=1.0.0.0, Culture=neutral,PublicKeyToken=null"
              description="CustomerProvider which retrieves customer data
from Navision using NAS"
              serviceConnection="UnitTestNavisionService" />
        </providers>
      </customer>
```

Now copy the NavisionCustomerProvider from the Sana Commerce Live Framework to your Unit Test project and rename it (for example MockNavisionCustomerProvider). Make all the methods empty except for the Initialize(string name, NameValueCollection config) method. Now change the config so it points to your new provider:

```
<customer defaultProvider="StarterSite1NavisionCustomerProvider">
   <providers>
     <clear/>
      <add name="StarterSite1NavisionCustomerProvider"

            type="Ism.Levis.Lobby.Test.MockNavisionCustomerProvider,
            Ism.Levis.Lobby.Test, Culture=neutral,PublicKeyToken=null"
            description="CustomerProvider which retrieves customerdata
            from Navision using NAS"
            serviceConnection="UnitTestNavisionService"/>
    </providers>
 </customer>
```

Now if the CustomerManager calls a method of the provider you can supply it with test data. For example let we test the method:

public override bool ValidateUser(string username, string password)

This method has to check if the password given by the user corresponds with the password in the database. The CustomerManager would be asked to get the account with the corresponding username. The CustomerManager calls the method GetShopAccountByEmail (users use their email as username). Now let we write the following Unit test:

```
// Tests if false is returned when somebody enters wrong password
[TestMethod]
public void TestValidateFaultyPassword()
{
      SCLErpMembershipProvider prov = new SCLErpMembershipProvider();
      bool result = prov.ValidateUser("test@ism.nl", "faulty password");
      Assert.IsFalse(result);
}
```

In MockNavisionCustomerProdiver edit the method GetShopAccountByEmail:

```
public override IShopAccount GetShopAccountByEmail(string email, bool
validOnly)
{
    IShopAccount account = null;
    switch (email)
    {
        case("test@ism.nl"):
              account = new
Ism.Levis.Lobby.Custom.Business.Business.Customers.Entities.ShopAccount(
);
              account.Password = "testpassword";
              account.Email = email;
              break;
        default:
              break;
    }
    return account;
}
```

We can implement ValidateUser:

```
public override bool ValidateUser(string username, string password)
{

Ism.Levis.Lobby.Custom.Business.Business.Customers.Entities.ShopAccount
account =
(Ism.Levis.Lobby.Custom.Business.Business.Customers.Entities.ShopAccount
)CustomerManager.GetShopAccountByEmail(username, true);

       if (account != null && ComparePassword(account.Password,
password))
       {
              return true;
       }
      else return false;
}
```

If you now run the Unit Test it will succeed.

B.2.5.3 HttpContext Simulator

Some user information is stored in the HttpContext and mocking the HttpContext can be difficult. These problems can be solved with a HttpContext simulator from Subtext. For more information see:

http://haacked.com/archive/2007/06/19/unit-tests-web-code-without-a-web-server-using -httpsimulator.aspx

We will show now an example of how user information in the httpcontext can be mocked. When a User get method is called from the SiteContext, you must check if the user is logged in. This information is stored in the HttpContext. First add a reference of the Simulator DLL

to the Unit Test project. Then add ”using Subtext.TestLibrary” to your code. To check if a user is logged in, the method IsUserLoggedIn from Ism.Scl.Web.Business.SiteContext is called:

```
/// <summary>
/// This function can be used to check if a user is currently logged in
on the site.
/// </summary>
public bool IsUserLoggedIn
{
    get
    {
        IIdentity userIdentity = GetCurrentUserIdentity();
        return (userIdentity != null && userIdentity.IsAuthenticated);
    }
}
```

The userIdentity is retrieved from the HttpContext. You can mock this in a unit test by the following way:

```
[TestMethod()]
public void UserloggedInTest()
{
  // User is loggedIn, return user
  SiteContext target = new SiteContext();
  // We need to simulate the httpcontext to set HttpContext.Current.User
  using (new HttpSimulator("/", @"c:\inetpub\").SimulateRequest())
  {
      IIdentity testIden = repository.DynamicMock<IIdentity>();
      IPrincipal testUser = repository.DynamicMock<IPrincipal>();
      HttpContext.Current.User = testUser;

      using (repository.Record())
      {
Expect.Call(testUser.Identity).Return(testIden).Repeat.Any();
Expect.Call(testIden.IsAuthenticated).Return(true).Repeat.Any();
      }
      using (repository.Playback())
      {
          target.User;
      }
    }
```

By simulating a request we can use HttpContext.Current. We assign the mock object testUser to User. The testUser then returns a mockObject testIden when the identity is asked. When in the method IsUserLoggedIn is asked if the user is authenticated, the mockobject returns true.

B.2.5.4 Mock SessionState

It is quite easy to mock the SessionState in Sana Commerce Live. For example if you want to mock the following code:

```
if (!SessionState.IsEmpty(identityName, sessionKey))
{
     return SessionState.GetValue<IWebUser>(identityName, sessionKey);
}
```

First you make ISessionState mockobject. Then you register the mockobject in the objectManager. Now when something is asked to the SessionState, it is redirected to the mock object, so we can decide what should be returned.

We want that IsEmpty returns false and that GetValue returns a WebUser. You use this code in your unit test:

```
ISessionState testSes = repository.DynamicMock<ISessionState>();
IWebUser user = repository.DynamicMock<IWebUser>();
// Register the mock object testSes
ObjectManager.RegisterInstance<ISessionState>(testSes);


using (repository.Record())
{
    Expect.Call(testSes.IsValueEmpty(null,
null)).IgnoreArguments().Return(false).Repeat.Any();
    Expect.Call(testSes.GetSessionValue<IWebUser>(null,
null)).IgnoreArguments().Return(user).Repeat.Any();
}
```

In the getmethod SessionState.IsEmpty IsValueEmpty is called on the attribute Current. Current is the mockobject testSes and returns false. In GetValue the mockobject testSes is asked to return an IWebUser and we recorded that the mockobject user is returned by testSes.

There is only one problem now, the mockobject testSes is now registrated and is in verification state. Because you can not unregistrate something (I do not know this for sure) you have to solve this somewhat unorthodox:

```
// Register the mock object testSes
ISessionState oldOne = ObjectManager.GetInstance<ISessionState>();
ObjectManager.RegisterInstance<ISessionState>(testSes);
```

We put the old sessionstate in a variable. When we are finished with the unit test we put the old one back:

$ObjectManager.RegisterInstance<ISessionState>(oldOne);$

You can put this in the setup and teardown, but not all tests mock the sessionstate.

B.2.6 Conclusion

Unit testing of a Sana Commerce Live Customization is possible. It will only take some time to get used to the unit tests and mocking the different objects. Also the team has to consider if some things have to be unit tested or an integration test is more appropriate.