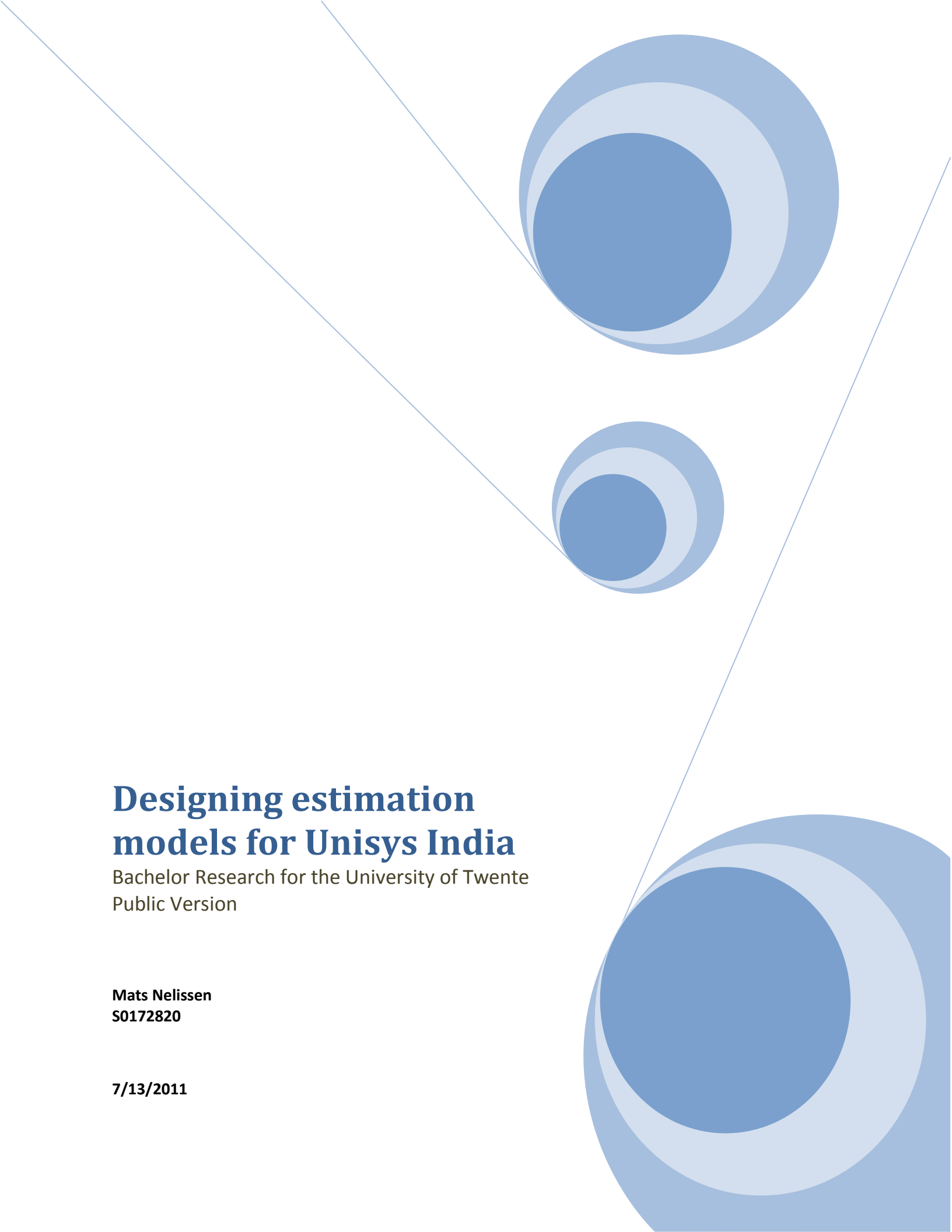

# Designing estimation models for Unisys India

Bachelor Research for the University of Twente
Public Version

**Mats Nelissen**
**S0172820**

**7/13/2011**

# Preface

This bachelor assignment has been created as part of an internship with Unisys Global Services in Bangalore, India. During this internship I worked on a project with the aim to provide improved cost estimation models for Unisys, a multinational IT company. During a period of twelve weeks I have worked in Bangalore, to collect all necessary information and knowledge.

During the work on these estimation models another opportunity came by. The scope of the research extended to integrating the new estimation models with existing HR models, enhancing Unisys forecasting abilities.

Before continuing the research I would like to thank some people for their help and support. Without them, this research would not have reached its current state. First of all, Peter Schuur, my primary university contact person. Without his support and critical questions I couldn't have successfully completed this research. I would like to thank Naik Nilesh Premanand and Badri Garla Prasad for their close support during my daily operations at Unisys. Hans Heerkens, my secondary contact at the University of Twente, thanks for taking the time to support me as well. Last but not least, I would like to thank Fred Bosch, the person without whom I would not have worked at Unisys in the first place.

Mats Nelissen

## Glossary

In this document several industry standard terms will be used. If at any time a word seems unclear, refer to this Glossary section for further explanation. They are ordered alphabetically.

| | |
|---|---|
| **Agile development** | Collective noun for many SLCs following the agile development manifesto, one common characteristic is short, frequent iterations without extensive planning. Unisys applies Scrum Methodology |
| **Automated Tests** | The opposite of manual tests. This form of testing requires the user to press a button, which will initiate the testing effort. The computer automatically executes all selected TCDs, and prints the result to the user's screen. No manual intervention is required. |
| **Complexity** | Used to indicate the complexity of both creating and executing tests on a four point scale: Not Applicable, Simple, Average and Complex. Based on Complexity TCPs will be awarded. |
| **CWF** | Abbreviation for Composite Weight Factor. The CWF is used to weigh the differences between testing projects, so the effort for a new project can be calculated based on a historic, somewhat similar project. |
| **Domain** | Vaguely indicates the software's application. Two domains used in this research are client-server applications, and Mainframe applications. Mainframe applications are also known as system applications. System domain should not be confused with the System test level. |
| **Environments** | Short for software environments. Examples of environments include: Windows XP, Windows 2000, Windows 7, Linux, Mac OS X. |
| **Estimation procedures** | Several procedures are considered during this research. They include Delphi Technique, Analogy Based estimation, Software Size Based Estimation, Test Case Enumeration Based Estimation, Task (Activity) based Estimation and Testing Size Based Estimation. Refer to section 5 for more information. |
| **Function Points** | Common standard set by the IFPUG organization to quantify the size of a software development project. |
| **Functional Tests** | Tests that have the objective to verify whether a specific functionality works. |
| **Integration tests** | Test level used to determine whether different modules (or units/components) work together when integrated. |
| **Manual tests** | This form of testing requires the user to perform all steps in the TCD by itself. Generally this is more time consuming than automated testing. |
| **Module tests** | Also known as Unit and Component tests. This is the lowest test level, where standalone functionality is tested. Usually performed during development. |
| **Non Functional Tests** | Tests that have the objective to verify the quality of the software, for example the software is targeted by an increasing number of connections, until it crashes. Several testing possibilities are connections per minute, total connections, data entries, et cetera. |

| | |
|---|---|
| **PMP** | Abbreviation for Project Management Plan. This plan is created when a project is first started. It contains all available information regarding that particular project. During the projects lifetime the PMP will grow. |
| **Regression Tests** | Common name for repeating software tests over time, resulting in less effort required. |
| **Scrum** | An Agile software developing technique. |
| **SDF** | Abbreviation for Solutions and delivery framework. It includes all Unisys' guidelines regarding their core businesses. SDF Testing refers to the testing section in the SDF. |
| **SLC** | Abbreviation for Software Life Cycle. A SLC describes the phases and stages any software development and testing project can be in, much like product life cycles. Refer to section 3.2 for more information. |
| **Static tests** | Static tests are performed without executing the code, i.e. by reading it. The opposite is dynamic testing, where the software is (partially) executed. |
| **System tests** | Test level to determine whether an entire system will pass the required criteria mentioned by the customer. This usually means combining chunks of software that have been verified by integration testing. |
| **TCD** | Abbreviation for Test Case Description. A TCD is a digital document which contains all relevant information about how to run a test, and when it is passed. Because of the detail required in a TCD, it can only apply to one project. |
| **TCE** | Abbreviation for Test Case Execution. TCEs are digital documents which describe the results of a particular test execution. It includes a reference to which TCD was being executed, and the results. Results can be PASS and FAIL. In case of failure more information regarding the failure is included. |
| **TCP** | Abbreviation for Test Case Points, a term used to quantify the size of a testing project. |
| **TCIS** | Abbreviation for Technology, Consulting and Integration Solutions. One of three major branches in Unisys worldwide. This research is performed for the TCIS branch in the Purva location in Bangalore. |
| **Test box** | Used to indicate how much access there is to the software code that needs to be tested. Testing boxes include white box (full access), black box (no access), and grey box (partial access). |
| **Test level** | Used to indicate the size of the software test. Test levels include Module tests, Integration Tests and System Tests. |
| **Test Scenario** | Client requirements are broken down into test scenarios. If all scenarios are passed, the client requirements are met. Scenarios again exist of potentially many TCDs. All TCDs need to be passed to pass a scenario. |
| **UGSI** | Abbreviation for Unisys Global Services India. All Unisys locations in India belong to the UGSI grouping. |

| | |
|---|---|
| **(U) RUP** | Abbreviation for (Unisys) rational Unified Process. Unisys' own implementation of the RUP SLC. This SLC combines iterative processes with detailed planning. |
| **User Acceptance Tests** | The final phase of software testing. In this phase the software is tested by its end users. Common examples are Alpha, and the more known Beta tests. |
| **VBA script** | Programming language which is used in this research where excel's functionality did not suffice. |
| **Waterfall model** | SLC without any iterative processes. |

## Summary

Software testing is one of Unisys' core services. For both internal and external clients, Unisys often resorts to one of its locations in India: the Purva location in Bangalore. The expertise, combined with the low cost of labor in India together result in a very competitive service offered.

One of the issues that is important when considering outsourcing to India is the total cost. Currently, cost estimations are done manually, by people who have been leading testing teams for multiple years. They resort to their years of knowledge to give a first indication of the required resources: both time and money. Should more accurate estimates be necessary, they break down the project in smaller pieces, and again use their expertise to make an estimation.

Although these estimates can be very accurate (less than 5% deviation), depending on the time available they do require up to two weeks to be made. That means that people in the sales team do not have quick access to such estimates. Also, small changes in requirements may require reviews by the team leaders, resulting in another few days lost.

This research is focused on tackling this issue, by providing estimation metrics in excel form, which anyone at Unisys can use to make a correct effort estimation. Because the model will lack personal interaction and expertise, it is likely the model will not realize the same error margin. UGSI management has set the acceptable error margin at 20%, the value they normally use for budgetary estimations. The central research question follows:

*How can a model describe the costs and time the testing function in Unisys' Purva location, Bangalore requires to complete any testing project within 20 percent of the actual costs?*

During the research we identified many different characteristics of software tests, that all have been considered when the model was built. We started by making a version of the model which allows for very specific data entry, namely per test case specific information. This level of detail is usually only available once the managers have made a detailed project management plan.

The model translates input into Test Case Points (TCPS), before translating TCPs into effort and US dollars. One TCP is defined to require 1.875 minutes of work, but this may differ slightly from person to person. Because all projects can be translated into TCPs, Unisys now has a tool which allows cross project comparison of their employees. Earlier it may have been hard to find hard evidence of someone performing far below the acceptable line. Now, with these newly introduced TCPs it will be very simple.

Once this detailed version of the model was operational, work started on establishing a baseline. This is a very important step, required to ensure the model is as accurate as possible. Although the results cannot be guaranteed, at times they are known to be within 3.5% range, which is nearly as good as the manual estimations.

The next step was to make tailored versions of the model, which require far less input. This will of course sacrifice some accuracy, but this will enable the sales team to respond to client requests very quickly. Two versions have been made: Scenario based input when some detail is available, and

Requirements based input, when there is practically no knowledge of the project at all. Depending on the situation, either one can be used by the sales team.

To ensure that the models can be used by people who may have less knowledge about software testing, such as the sales team, a detailed manual has been created to go with the model. In this manual every aspect of the model is explained, ensuring the easy usability that is vital for the successful implementation. To further increase the chances of this research being installed in Unisys future daily operations, a small training session has been given to all the team leaders that will use it. They will share this knowledge with all people who may need to resort to it in the future.

At the start of this research, the only focus was creating estimation models. However, over time more ideas were generated that could be of great value to Unisys India. One of those ideas was to integrate these newly created models with other currently existing reports. This will not only enhance Unisys ability to reply quickly to customer requests, it will also give Unisys better insight in its own available resources. We have added a secondary research question to address this idea.

*How can the newly created models, together with existing forecasting reports, be combined into a dashboard overview which provides forecasting figures related to manpower?*

This additional model is now known as the Supply Dashboard, and is dependant of mulitple weekly and montly reports and forecasts. It combines these files into a clear dashboard overview in which management can quickly see the size of Unisys' task force for the next six months. Moreover, it also displays the estimated people required per month, a bench percentage, and gap figures between availability and requirements. Based on this Supply Dashboard Unisys may decide to approve or reject projects, and whether the current taskforce size at any point in time needs to be influenced.

# Table of Contents

# 1 Introduction

During this research I have worked on a project with the aim to provide improved cost estimation models for Unisys' software testing function. Before continuing to the problem definition and research approach, I will first introduce Unisys. We will have a look at Unisys' global business structure first, and then at the structure in India, where this research is performed.

## 1.1 Unisys around the world

Unisys is a large multinational operating in the IT industry. It has been founded in 1873 as a typewriting machine fabricator. Over time transformed into a full systems integrator and outsourcer delivering a focused set of IT services and products. Today, Unisys serves clients in over 100 countries. Unisys employs around 23000 globally.

Until 1994 most revenue resulted out of IT Products and connected services, such as mainframes and open systems. From 1994, services and solutions became the company's single largest business. Starting 2001, Unisys entered into the first long term outsourcing contracts with well known companies like Lloyds TSB, Air Canada, BMW Bank and GE Capital Bank.

Nowadays Unisys presents itself as a company which designs, builds, and manages mission-critical environments without any room for error. To be able to deliver these services, Unisys employs experts on consulting, systems integration, outsourcing, infrastructure and server technology. Combined expertise leads to the four areas in which Unisys excels:

- **Security**: helping clients secure their operations — people, places, assets and data to create more reliability and less risk.
- **Data Center Transformation and Outsourcing**: increasing the efficiency and utilization of data centers.
- **End User Outsourcing and Support Services**: enhancing support to customer's end users and constituents through their devices and desktops.
- **Application Modernization and Outsourcing**: modernizing their mission-critical business applications.

**Business Structure**

On a global scale Unisys only employs three business units (BU). These are Federal Systems, GOIS and TCIS. This research is performed for the TCIS business unit in India.

**Federal Systems**

Federal systems is a very secure branch within Unisys, which delivers a broad range of services and solutions to U.S. government clients, including consulting, systems integration, managed services and outsourcing.

**Global Outsourcing and Infrastructure services (GOIS)**

The main function of this business unit is concerned with IT infrastructure, outsourcing and other service management services.

The main focus of this BU is on winning and managing outsourcing contracts. This can range from large desktop and end user support contracts, managing the security of clients' networks and providing tailored infrastructure (cloud/hosted) solutions.

This BU also contains a field operation organization that provides onsite infrastructure services.

**Technology, Consulting, and Integration Solutions (TCIS)**

This business unit is provides systems integration, consulting and technology products and services to clients worldwide.

## 1.2 Unisys in India

One of the countries Unisys operates in is India. Within India, there are four different locations: one in Mumbai (Corporate Office), two in Bangalore and one in Hyderabad. The GOIS and TCIS business units are both represented in India.

The location where this research is conducted is the "Purva" location in Bangalore, part of Unisys Global Services India, or UGSI. Currently, roughly 1000 people work at this location. It serves mostly as an outsourcing hub for services that other departments around the world require.

In this location there are two areas of interest for this research. Both are part of the TCIS business unit, for which I will work. The first one is TCIS Services. This department offers software development and testing capabilities to other Unisys locations. For example if Unisys Europe wants to (partially) outsource a software development project, TCIS services would pick it up. This can be done for several reasons, including capabilities, competencies, quality, efficiency and cost offered by UGSI.

The second area of interest is located in ESC, which is part of TCIS. All ESC's subdivisions have a testing component, since all software Unisys develops needs to be tested. This means that across all five functions in ESC testing needs to be done.

# 2 Problem definition

The redesign of the test function has led to managers rethinking ways things work at the facility. Currently the process is as follows:

- A potential client has a software program that requires testing. This client can be external, but often is an internal client. Internal clients are located around the globe, and require their programs to be tested at relatively low cost.
- The client submits all he knows about the software and to which extent the software requires testing. More thorough testing scopes and techniques generally cost more recourses: both time and money.
- The request is reviewed by a team leader at the Purva location in Bangalore. All non technical work is done by presales, but for cost estimation the request is forwarded to the test department. There, a reviewer creates an estimate of the required time and money to complete the testing. This is done based on expertise and some models/metrics where available.
- If the client approves of the estimate the process is taken to the next step where details and contracting are discussed. If not approved, the client can either cancel the request or change the submitted information. Should he resubmit, the process is repeated.
- When the project is formally approved the following processes are initiated.

Usually creating the estimates is not a very time consuming task. If it is done effectively it can be done within one or two days. However, because workloads tend to be high the requests are usually not reviewed immediately. Also, there are only few team leaders skilled adequately for this task. Due to these facts average times range from 2 to 5 working days for each request. In extreme situations it is known to take up to two weeks and more.

Estimates are by their nature not required to be 100% accurate. Besides that, a large part of all testing work is very similar to other testing projects. This means that estimating the costs is a rather standard procedure. Therefore, managers now believe that a model can be made which outputs the estimated costs. Because a model can never fully replace the expertise a tester uses when reviewing each request, the model thought of is supposed to make very rough estimates. These rough estimates are better known as budget estimates, and have an error margin of up to 20%. These budget estimates can be very useful for a client to get a first idea of the costs, and can even be sufficient for smaller projects. When more accurate estimates are required clients will still have to submit their requirements to UGSI.

A major advantage of such a model would be the possibility for clients to create estimates on the fly, allowing them to see how minor changes in their requirements would influence total costs. How the model will be used in the end is unknown at this point in time, but it could for instance be implemented online. This would enable any potential client to get a fair approximation of the required recourses for his project instantly.

Another advantage is that the total number of requests for estimation the test team has to process will be reduced. This is because some clients will stick to the budget estimates and other clients may refrain from a request because the budget estimate shows it is far above their budget. Then finally potential

clients won't have to resubmit their requests as often as currently since they already inputted several different requests into the model and know which options will be most likely to fit their needs/budget.

Finally, it can contribute by making the choice to outsource a (partial) project easier. Currently it is not easy to determine whether cost/benefit wise it is attractive to move any project from its current location to India. A model will give more (direct) insight in the costs, and therefore simplify the matter.

## 2.1 Central research questions

UGSI has asked me to build such a model/metrics. The central research question for this research will therefore be:

1. *How can a model describe the costs and time the testing function in Unisys' Purva location, Bangalore requires to complete any testing project within 20 percent of the actual costs?*

To answer this question the processes the testing function uses to go through the testing process must be known. Therefore the following sub question must be answered first:

1.1. *What are different testing techniques performed by the testing practice, and how can they be categorized based on approach and required resources?*

To answer this question I will first look into all testing techniques used industry wide, and later look into how Unisys uses these techniques in its own test function. The next step will be to categorize these techniques based on the mentioned criteria.

Then to build the model, it will be necessary to identify all relevant input parameters, and to identify their respective inputs on the cost. The resulting question is the following:

1.2. *What factors have an influence on the required effort and cost of the testing function, and what is their respective impact?*

To answer this question I will look into the current estimation processes, see whether there are other processes used across the industry, and meet with project managers to discuss their views on estimation.

To build the model first time right would be nearly impossible, since accurate data on cost parameters are mostly unknown. Therefore it is necessary that the model allows for easy manipulations of these parameters, to allow for continuous improvement of the accuracy of the model. To realize this, during the "building" of the model, we will constantly consider the following issue:

1.3. *How can easy tweaking of the model be realized, so it won't lose its value in the coming years?*

Apart from the main research question, Unisys has asked me to do one more thing: integrate these new models and existing models into an overview dashboard that will provide forecasting figures for manpower related issues. The final question that we will look into during this research is as follows:

2. *How can the newly created models, together with existing forecasting reports, be combined into a dashboard overview which provides forecasting figures related to manpower?*

### 2.1.1 Problem cluster

This research will focus mostly on the primary research question. During the work on this issue we found that the problem we were solving was actually the root of another major problem too, which led to research question number 2. The problem tangle in Figure 2.1 graphically illustrates how providing models will resolve both central research questions.

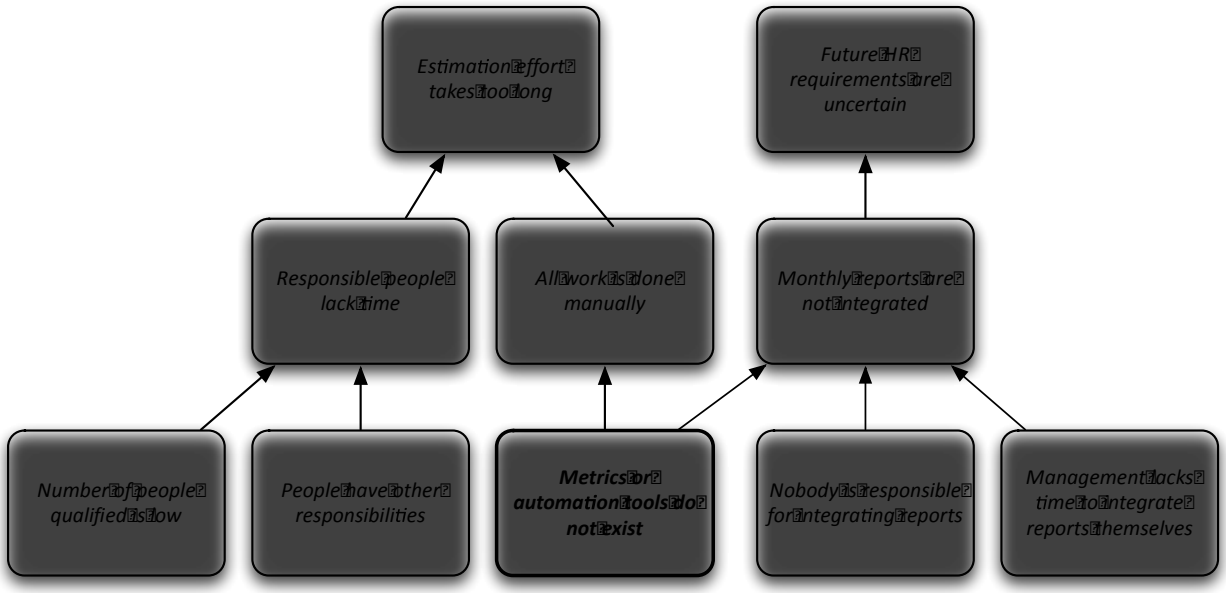


Figure 2.1 Problem cluster

## 2.2 Approach

As became clear in the introduction section, Unisys India has many different departments. Also in all these departments, different project managers run different projects. Currently, all these people use different techniques to make their estimations. That means that the result of an estimation depends on the person who makes it. The main goal of this project is to eliminate those differences, by providing UGSI with an estimation model which everyone across the facility will use.

To do so, I will follow the following steps during the three month period I am in India.

- First I will get an understanding of all different testing efforts that are performed at the Purva location. The goal of the first period is to get a good understanding of what UGSI does, so the model can be adapted to that.
  To do this I will go through Unisys' internal documentation, external information sources, and have meetings with different people at the UGSI organization.
- The next step is to consider different techniques and metrics for estimation, and to select the one that will form the fundament for the model. I will do this in close collaboration with management.
- The next step will be to identify all relevant input parameters, and to translate them into a model which will output the effort in person hours and dollars.
- The final step will be the optimization of the model, I will do this by inputting as much historic data as possible, to confirm the values for the parameters used in the model.

We have agreed that the model should be build in excel. This is for several reasons: excel allows for easy future tweaking, should the model's results no longer be accurate. Also, it is easy to understand for the users, since excel is commonly used across the UGSI facility. The third reason is that I feel comfortable building such a model in excel, whereas other programming languages are not part of my training and may prove to be too difficult.

During the research I found the basic excel functionality not to be sufficient. Excel is has good calculation options, but also has its limitations on other areas. An example is the functionality to automatically add similar rows, or to solve linear equations without user interference. To resolve these and other issues I have resorted to Visual Basic for Applications, a more complex programming language which can be interpreted by Excel. Although my personal skill for this particular programming language is limited, I was able to use it where Excel's functionality did not suffice.

This means that the final deliverable will be an excel model, which can be used for *any testing project*, by *anyone* in the UGSI facility, at *any time and phase* during the testing effort, and will give *accurate results* considering the phase.

Once this is all done, I will look into the secondary research question: combining the model with existing reports. Depending on the timeframe I will discuss with Nayak Prashant, my manager for this secondary objective, on how to implement this idea.

### 2.2.1 Public version

This document is the public version of this research. In this version there will be no information that can not be found on public websites as well. Therefore, several sections have been removed from the original file. The sections that are listed in this document conatin public available information regarding Unisys, as well as the information used as theoretic framework. Also, it contains a altered version of the conclusions and recommendations section.

# 3 Testing Theory

To continue the research the first step is to understand all different testing efforts that are performed at the Purva location. During this phase we will answer the first sub question which will give the first indication of different input parameters that influence the estimation. The first question is the following:

*What are different testing techniques performed by the testing practice, and how can they be categorized based on approach and required resources?*

Immediately upon starting to investigate this, I found there should be an extra split here. On one hand there are the different techniques that are used, but also there are different approaches to plan the testing effort. Planning differently can impact the total cost, and this parameter may therefore be relevant for the model. The different approaches are known as software lifecycle models, and will too be explained in this section.

First, this section will describe all software testing procedures as they are known today. There are many different procedures and techniques, all with different characteristics which can be used to categorize them. For estimating it is not relevant which techniques are used, instead it is more interesting to understand what *kind* of technique is used. Therefore in this section all different categories will be mentioned, and more detailed information will be moved to the appendix.

Different Lifecycle models will be described in section 3.2.

## 3.1 Testing characteristics

The goal of any testing practice is not to guarantee that the software works, it is to find conditions in which the software does not work. This can never fully guarantee that the software will never fail. This is an important principle in testing that always holds for more advanced software. This is because it is virtually impossible to test every possible form of input.

A good definition of software testing is the following:

*Testing is a process of planning, preparing, executing and analyzing, aimed at establishing the characteristics of an information system, and demonstrating the difference between the actual status and the required status.* (Jari Andersin, 2004)

### 3.1.1 Test objective

Different testing efforts have different causes and different objectives. The most common are the testing efforts performed during development, regular testing efforts. Two other efforts are regression testing and user acceptance testing.

#### *3.1.1.1 Regression testing*

This form of testing means using previously run test suites to confirm the program is still working. This can be very useful to test whether a code change (update) has broken previously working sections of the

program. Proper regression testing is both efficient and automated because all tests have been run before, to verify earlier versions of the program. Not running regression tests can cause dramatic failures, as happened in the US several years ago when a mobile provider lost all connectivity for several days after a very small update. Regression testing would have shown that the change broke the software and should therefore not be installed (yet).

Any test that is run for the second time is by definition a regression test. Running tests multiple times also means there are going to be time benefits: the team will learn, and the test case needs to be created only once and can be used many times. For this reason it is an important test characteristic, and the model should be able to distinguish between regression tests and regular, first time tests.

#### *3.1.1.2 User acceptance testing*

There are three forms of user acceptance testing (UAT): smoke tests, alpha tests and beta tests.

Smoke tests are small tests to verify that the program works in the first place. Before continuing to debugging and more thorough testing the program needs to be able to start in the first place. These tests are always run to verify the programs stability before starting the scheduled testing efforts.

Alpha tests are run when a program is nearing completion. It is distributed amongst multiple people in the company, to have more end user feedback on the functionality.

When alpha tests have passed the program continues to beta testing, where it is distributed to the public. The objective is the same as in alpha testing: get more end user feedback to find the last bugs before the final release. Once the program passes beta testing and all feedback has been processed it is ready for final release.

UAT is the final stage of testing and is not considered part of the core testing function. The time it requires is often derived from the total core testing time, using some factor. For different projects the factor may be different, and in the model it should be properly incorporated.

### 3.1.2 Functional and non Functional tests

Functional tests are all those tests that will check a section of code or an entire program for functionality. The outcome of a functional test is the answer to the question: does this area of code work? The more different inputs used, the more certain one can be that the answer is not false positive.

The opposite is non functional testing. Non functional tests test the programs capability to handle certain circumstances, and register the programs performance in those circumstances. A typical non functional test would be to test the program's performance under 100 connections per minute, and 1,000, 10,000. Etc.

Functional tests are typically run during development, whereas non functional tests are mostly run to test the performance of an operational system. This is usually at a later stage in development.

All software tests can be qualified as either functional or non functional.

### 3.1.3 Testing levels

According to SWEBOK, the international Software Engineering Body of Knowledge, there are four levels in which a test can be run. They are described here, in order of size and complexity.

#### *3.1.3.1 Unit testing (component testing, module testing)*

This is the lowest level of testing. Its purpose is to test a specific section of the code, usually a specific function. Often these tests are done by the programmers themselves during the development, to ensure what they have just created is working properly. It will take significantly more time for someone else than the coder himself to create appropriate Unit tests.

#### *3.1.3.2 Integration testing*

The purpose of this level of testing is to verify whether software components will work together. It is usual to start small, integrating two components. Then, should this work a third and fourth will be added. Later chunks of components that are known to work can be added as well. Connecting all components at once is known as the big bang, and is not usual unless there is reason to believe that everything will work flawlessly right from the start. If it doesn't, it is not easy to trace the error location since it is unknown which sections work and which don't.

#### *3.1.3.3 System testing*

The purpose of this test is to verify whether an entire system is working as it is supposed to. This can be both functional and non functional tests. It can be tough to test this thoroughly, since systems can be large, and allow for infinite different input combinations. Only one needs to be bugged to be able to crash the system.

#### *3.1.3.4 System integration testing*

The point of this test is to verify whether a fully working system will also work in or with another third party system or environment. This too can be a very tricky area to test, since one will assume that many things work, because they are known to work in another situation. This can easily lead to insufficient testing.

### 3.1.4 Testing boxes

There are three boxes known in software testing: white box, grey box and black box.

White box testing refers to all testing that is done while having access to the source code of the program. This allows for very thorough testing such as code review, literally checking every line of code one by one.

Black box testing on the other hand is the opposite: the code is not available to the tester. The tester will have to input lots of data to make sure all desired scenarios are tested. Still, there are very structured approaches to handle these situations.

In between these two areas is grey box testing: here the actual code is unknown, but the tester does know how modules within the software interact with each other. Moreover he can also have these modules print logs of their actions. This allows the tester to know what actually happened in between the first input and the final output (unlike in black box testing), but he doesn't know exactly why it happened (which is the case in white box testing).

### 3.1.5 Static and Dynamic testing

Static testing includes all tests in which the code is not executed. Static testing includes reviews, walkthroughs and inspections. Because of the nature of static testing, all are white box tests (it is not possible to check the code without executing it, while not being able to actually see it).

The opposite is testing the code by executing it with test input parameters. This is known as dynamic testing. Both procedures can produce valuable results, but many firms leave out static testing as they believe its efforts can be covered by dynamic tests as well.

## 3.2 Software Lifecycle Models

Software Lifecycle (SLC) models represent the timeline any software program passes. These models will go through the following phases: requirements phase, design phase, implementation, integration, testing, operations and maintenance.

It is very important to be aware of the software lifecycle before development starts and to pick the model which is most appropriate for the project at hand. Not considering the SLC or choosing the wrong model can result in higher development costs and even project failure. There are several different models, which will be described in this section. Depending on the amount of relevant information known and the volatility of that information the proper model needs to be selected.

These models can be categorized in three different groups: traditional models, Agile models, and the Rational Unified Process. Depending on the project, Unisys uses one of three different models: the waterfall model, which is a traditional model, SCRUM, which is an Agile model, and URUP, Unisys' own version of RUP. These will be described in this section. For a more clear understanding of other lifecycle models and how they differ from these three, see appendix B.

Using different approaches (models) will result in a different project planning and will therefore influence costs. It is important the model is able to distinguish between these different lifecycles.

### 3.2.1 Waterfall model

This is the oldest and least flexible model around. Using this model, there will be no iterative processes and proceeding to the next step in the development stage will only be done upon full completion of the previous step. This model should only be used if all requirements are known at project start, and it should be very unlikely for these requirements to change in the future. These tend to be projects that can easily be outsourced overseas. The development steps of the waterfall model are as follows:

- Document system concept
- Identify system requirements and analyze them
- Break the system into pieces (Architectural Design)
- Design each piece (Detailed Design)
- Code the system components and test them individually (Coding, Debugging, and Unit Testing)
- Integrate the pieces and test the system (Integration and System Testing)
- Deploy the system and operate It

The general criticism to this model follows:

- Problems are not discovered until system testing. (late in the total SLC)
- Requirements must be fixed before the system is designed - requirements evolution makes the development method unstable.
- Design and code work often turn up requirements inconsistencies, missing system components, and unexpected development needs.
- System performance cannot be tested until the system is almost coded; under capacity may be difficult to correct.

### 3.2.2 The Rational Unified Process

Traditionally, in many software development projects testing is seen as a burden that has to been done at the end of development. It adds a lot to the total cost of a project and doesn't contribute a lot. Furthermore, when development does not meet deadlines, testing is often delayed. In many cases the final delivery deadline is not delayed, so the total time for time for testing suffers. Testing is done by the end users, which is not desirable as this can increase testing costs by a factor 100-1000.

To overcome this issue a more modern approach has been development in the recent years. The main idea is to integrate the testing more throughout development, and it's called the Rational Unified Process (RUP). Like many other models (Appendix B) RUP is an iterative process. RUP has several tools to realize better integration of testing:

- Making testing a distinct discipline
- Using an iterative development approach
- Scheduling implementation based on risk
- Continuously verifying quality
- Letting use cases drive development
- Managing change strategically
- Using the right-sized process

For this research it is important to understand how using RUP influences the planning and thus the costs of a project, compared to the waterfall model and SCRUM. This information will be obtained by interviewing and meeting with people who are familiar with this approach. For a more clear description of the tools mentioned above, refer to appendix C.

### 3.2.3 Agile Software development

The final large branch in software development is Agile developing. Agile is a group of techniques all following the agile development manifesto, a document created in 2001. The main difference compared to traditional and RUP approaches, is its tendency to focus on reacting to change rather than planning ahead. A team which follows an agile approach might not be able to say what tasks are planned for next week. Instead, they might only have a vague planning of what features are planned to be developed during the next month. Also, the iterations tend to be very short, preferably 1-4 weeks, compared to at least 6 weeks in RUP. Because of the short duration they are usually referred to as sprints. One other big conceptual difference that one member of the team is actually one of the clients employees. Involving the client in the development structure ensures the team will be very responsive to the clients changing requirements and expectations.

Agile can be implemented in three different ways: only at the actual development level, only at the project management level, or both. Focus on both is better known as full coverage of the development lifecycle. The best results are booked in relatively small project where no more than 10 people work on a project. However, there are reports of successful implementation in larger (40 people) projects.

For this research it suffices to mention the key elements that characterize Agile development instead of explaining it in detail. This is because Unisys only uses one Agile programming technique: SCRUM. The key elements are the following:

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances

There are many different implementations of Agile development used across the industry, Unisys chose to go with Scrum. All Agile characteristics are also seen in Scrumming. Again for this research it is not relevant to know the exact steps during Scrum development, rather it is important to realize that it has a different impact on costs, and therefore should be an input parameter for the model. For a detailed explanation on Scrum, refer to Appendix D.

| | Waterfall | RUP | Agile |
|---|---|---|---|
| **Iterations** | None | Long | Short |
| **Planning** | Rigid/long term | Flexible/Long term | Flexible/Short term |
| **Volatility in requirements** | Not acceptable | Possible | Usual |
| **Project size** | Huge/Large | Large/Normal | Small |
| **Testing integration** | Afterwards | Fully integrated | Fully integrated |
| **Documenting** | Average | High | Very low |
| **Typically used in/by** | Complex, mission critical projects without tolerance<br><br>NASA/ Military | Government/educational institutes<br><br>Partial off shoring, packaged releases. | Single small projects, fast market response<br><br>British Telecom |
| **Downsides** | Slow, expensive, project maybe obsolete on completion | Not as quick as Agile<br><br>No competitive time to market results | High personal interaction → project must be on one place. Low documenting → Unclear forecasts. Requires experienced team. |

Table 3.1 Software lifecycle overview

## 3.3 Summary on Testing Theory

This concludes the section on testing theory. All differences that characterize the tests have been discussed, as well as different software lifecycle models. More information on specific testing techniques can be found in appendix A. They include some examples of different functional and non functional tests, many of which can be used in all three boxes, and at different testing levels. Some of these procedures are more common than others in Unisys' daily operations. Also, more information on other SLCs can be found in the mentioned appendices.

It should be clear that most software tests can be used in different ways. There are characteristics for each test in a given project but most tests will have multiple possibilities (i.e. fuzz testing can be used for integration testing as well as system testing).

The first sub question can now be answered.

1.1. *What are different testing techniques performed by the testing practice, and how can they be categorized based on approach and required resources?*

During the search for different techniques we haven't found any evidence to state that approaches can be categorized based on resources. However, it is possible to identify different approaches to software testing by considering several characteristics. The characteristics we found follow in table 4.4.2. Additionally the planning aspect is also relevant: it doesn't characterize the test itself, rather it shapes the planning of the entire planning effort. It is however an important parameter and should therefore be mentioned here.

| Planning Approach | Objective | Testing Level | Static / Dynamic | Testing Box | Functionality |
|---|---|---|---|---|---|
| Waterfall | Regular | Unit | Static | White box | Non Functional |
| URUP | Regression | Integration | Dynamic | Grey box | Functional |
| Scrum | User Acceptance test | System | | Black Box | |
| | | System Integration | | | |

**Table 3.2 Overview of test characteristics**

# 4 Estimation procedures

In this section we will look into several common estimation procedures. We will mostly discuss procedures that are specific to the testing industry, but for instance the Delphi technique is used in other industries as well. This section will provide basic knowledge about different approaches to cost estimation, which can be used when we start building the model in section 6.

Recall from section 4 that Unisys starts each testing effort by creating the Project Management Plan (PMP). In this plan the first estimation is included, and the client has to agree with it before the testing can actually start. At this point in time it is very hard to outline exactly what tests need to be done. In this PMP the entire test effort is planned, but on high level scale. The client usually only specifies what software functionality he wants to have tested.

The next planning that will be made is for the first iteration. This planning will have specific information about who will test what functionality, using what tool. Once this planning has been created the PMP is updated as well with this more in depth information. Because this planning is far more specific it allows for far more detailed cost estimation and allocation. However, that is still only information for the next 6 weeks, thus the complete estimation is still mostly rough.

Once the first iteration is over the actual results are also logged into the iteration plan. The finished first iteration plan is added to the PMP. This allows for comparing the estimated data to the actual data. As time passes more and more information is available: the initial request from the customer, planned efforts for every iteration, and historic data on previous iterations. All this data can be of significant value in effort estimation. Therefore, it should all be used in the model.

Although no detailed information is available at project start, it is still very important to have a proper estimation. Because of the differences in available input, the best option is to make the model in such a way that it allows for multiple different forms of input. This way it can be used by both the field team, and the actual test team. The field team will use an input form which will result in rough estimates from client requirements, whereas the actual test team will input far more detailed input, such as the historic data mentioned earlier.

There are several known ways for cost estimation in software development, all with different characteristics. Here 6 different techniques will be discussed and compared. Together with management and the test team we will then decide which techniques can properly describe the test function at Unisys.

- Delphi Technique
- Analogy Based estimation
- Software Size Based Estimation
- Test Case Enumeration Based Estimation
- Task (Activity) based Estimation
- Testing Size Based Estimation

One difference between these techniques is whether they are top down or bottom up approaches. Top down techniques derive the estimation from the size of the project. Measuring size can be done several ways, depending which technique is followed. Opposite to this approach is bottom up, where the project is broken down into tasks which are then all estimated and summed. Bottom up techniques generally require more entries and therefore more effort to complete. On the other hand for top down approaches the translation from size to effort is more difficult to properly model. Currently Unisys uses different bottom up approaches, mostly Test Case Enumeration Based Estimation and Task based Estimation.

## 4.1 The Delphi Technique

This is a well known forecasting technique that is used for forecasting in many different fields. The technique relies on several different experts, who all give their opinions and answers to a question, not knowing what the others think at that time. After all experts have given an answer, they will all see the answers the others gave as well. They are then allowed to revise their answers based on the new information and points of view. This is repeated until some stop criterion is met, for instance time or number of iterations. The idea is that over time their answers will converge, and finally find a consensus. That answer is then used for forecasting.

This technique can for obvious reasons not be used in a model. However, it can be used as a checking method to confirm the data a model has created. This technique is used mostly when people are uncertain about the forecasts they have at some point, and thus require more verification.

## 4.2 Analogy Based estimation

This technique uses historic data from similar projects as a basis for estimation. It requires an extensive database of historic projects, so that each new project has close similarities with at least one other project. This is a top down approach: an estimation is made without breaking the project down into smaller pieces.

The first step is to select criteria on which the projects are going to be compared. The selection step should not be automated because it can easily influence the results of the estimate. Doing so would render the entire estimation useless, and therefore some human interaction is required here. Criteria that can be used are for instance: type of the project, application domain, client organization size, modules and many more. (Chem Cons).

Once a matching project has been identified the differences that still remain must be accounted for. If a difference is considered not relevant it can be discarded. (I.e. it won't have an impact on the testing effort). All other differences need to be weighted. Weight between 0 and 1 will result in the new project having less effort; more than one will assign more effort. Once all differences are weighed the composite weight factor (CWF) is calculated. This is done by summing all weights, and dividing that sum by the number of weights. Then, the final step is to multiply the effort of the matching project by the CWF. This will result in the estimate for the new project.

This technique can be very reliable, since it uses historic data created by the organization itself. Practice proves that is very easy to learn, and that anyone who understands the process can come up with good estimates.

For many companies this technique is not very useful because they lack the historic data. That can be either because they are a startup, or because they haven't kept detailed job histories. Unisys does have a large database of previous jobs. The detail of the data is unknown at this time, but before choosing this technique to make a model it should be verified that the detail is sufficient for the model to make budget estimates.

One issue that needs to be considered is that no matter what size the database, there may always be a project that has no good matching historic record. Usually this occurs when a company is entering a new field of expertise. This can happen at Unisys too, since they are relatively new to Scrum methodology. If historic data is lacking other estimation techniques should be used, otherwise there is a risk the estimate is too far off.

## 4.3 Software Size Based Estimation

This technique too is a top down approach. Instead of comparing the project to another one, its size is determined. A common tool for quantifying software size is function point allocation. The norms for one function point are set by the IFPUG organization. For instance, the number of different screens, buttons, and inter-program connections all increase the number of function points allocated to a particular functionality. The more function points allocated, the larger the project. For this research further understanding function points is not required. Based on the number of function points, the estimate can be derived.

Unisys already is familiar with function points, since they are also used for other purposes, mostly in development. Using function points ensures some degree of standardization which is very attractive for Unisys, because it enables management to compare projects. Currently different project managers use different models and different procedures to quantify the size of their projects.

To use this technique the function points should be multiplied by a productivity figure, which represents the number of function points one employee can work through per hour. The outcome would be the effort estimate.

The simplicity of this technique is both an advantage and a disadvantage. The positive side is that it allows for very fast calculations and it is easy to learn and comprehend. The problem however lies in the fact that not all software of the same size requires the same effort. There are major differences between web programs, standalone programs, client-server programs and 3 or even 4 tier programs. To solve this, different productivity figures must be used, and these figures must be accurate as well. The result is that this technique requires a very detailed and large historic database. The data must be occasionally checked by professionals to ensure the productivity figures are correct. If this is done properly, the technique can result in very accurate results, considering its simplicity.

## 4.4 Test Case Enumeration Based Estimation

This is a more detailed estimation technique. It will require more knowledge about the actual testing program and how much time each test will probably take. It is a bottom up approach, which some project managers at Unisys are currently using.

First, all test cases that need to be run are identified and listed. This requires a lot of knowledge about the to-be-tested program, and will take some time. The next step is to set the expected test time for each specific case. Also, best case and worst case scenarios are added for each case. Then, using a beta distribution, the estimated number of person hours can be derived. Summing these expected numbers will lead to the estimated effort.

The obvious disadvantage is that this will take far more time than other techniques. However, Unisys could use default hours for test cases to save time. They could set worst, normal and best cases for a number of different test cases based on difficulty. This will reduce the required input time. Still, because of the required detail, this technique is not useful for the model the field team requires.

The major advantages are the technique's accuracy, and the fact that it includes a worst-best range in which the project is likely to be completed.

## 4.5 Task based Estimation

This technique too is a bottom up approach and is very similar to test case enumeration. It adds more detailed estimation for noncore efforts that needs to be done. Instead of using the number of tests, the number of activities is used. Because a testing project consists of far more than the actual testing it can be beneficial to include these steps in the estimation. In previous techniques this can be done using factoring, for instance allocate 70% of the time to testing, then estimate testing and calculate the remaining 30%. However, actually estimating such tasks is often more accurate (and time consuming).

Each company has a different approach to testing, and so does Unisys. Unisys approach has been described earlier, in the SDF testing framework. The phases and corresponding iterative activities are mentioned there, however tasks were let out. There are many tasks in SDF testing, and for task based estimation to be accurate all tasks would need to be estimated in the model. For each task a worst, normal and best scenario needs to be defined, and again using a beta distribution the estimates can be calculated.

This technique is generally considered the most accurate. However, executing it thoroughly is a very time consuming task. Therefore, it is more common to sacrifice some accuracy for the sake of usability. For this technique to be useful at Unisys a lot of the tasks would have to be standard values, since there are far too many tasks to manually estimate in each time a project starts. There can of course be multiple standards, for different kinds of projects. Also, the standards need to allow manual manipulation if there is reason to believe the project will require different values.

Similar to the previous technique Task based estimation cannot be used for the model that will be used by the field team. It simply requires too much detail and effort to be useful at that point in time.

## 4.6 Testing Size Based Estimation.

The final technique that is interesting for the model Unisys requires is Testing Size Based Estimation. This technique uses so called Test Points, which are derived from the size of the software. To determine the size function points are a common tool.

The point of this technique is to normalize each project, so it can be compared to any other project as well. This is a great added value compared to other estimations, as it enables performance measurement. Such measurements are often lacking in the testing practice.

To normalize a project the size is converted into Unadjusted Testing Points (UTP). This is simply done by using a factor; this factor should be the same for all projects. The next step is to normalize the UTPs. The first factor is for the type of software: standalone, clients-server, or other options. The next step is to compute a composite weight factor, similar to the analogy based estimation technique. This time it is however done slightly different:

The combined weight of integration testing, system testing and user acceptance testing is defined to be 1. Then, if more tests need to be done, their weight derived from these weights. If for instance IT, ST and UAT are all thought to take the same time, their weights would all be 0.33. Now if the projects also required unit testing, which is thought to last twice as long as system testing, its weight would be 0,66. Similarly more weight can be added if more tests need to be done. The sum of all weights is the CWF.

Using this procedure to calculate the CWF, and using a standard procedure to derive function points ensures that all projects can be compared to each other. That makes this technique very useful to determine a tester's productivity.

## 4.7 Conclusion on estimation procedures

For additional clarity the following table provides an overview of the differences between the six techniques.

| | Delphi technique | Anology bE | Softw. Size bE | Test Case Enume. bE | Task bE | Test Size bE |
|---|---|---|---|---|---|---|
| **Manual req.** | Difficult ++ | Difficult | Easy | Difficult | Difficult | Easy |
| **Automation** | No | - | + | No | No | + |
| **Approach** | Both | Top down | Top down | Bottom up | Bottom up | Top down |
| **Database req.** | No | Yes ++ | Yes | No | Yes | Yes |
| **Time consuming** | Yes | Yes | No | Yes + | Yes ++ | No |
| **Skill req.** | Very high | High | Low | High | High | Low |
| **Detail req.** | Not req. | Not req. | Some | High | High | Some |
| **Pros** | Always usable | Easy to comprehend | Standardized, allows comparing | Accuracy, confidence range | Most detailed, confidence range | Standardized, allows comparing |

| | | | | | |
|---|---|---|---|---|---|
| **Cons** | Not automatable | Risk of no match in database | Size doesn't say it all | Very time consuming, high manual requirements | Hard to comprehend |

**Table 4.1 Comparison between estimation procedures**

Together with management we have decided to create a new model, using different characteristics of the above mentioned techniques. We have agreed on the following points of interest, based on which the model will be developed. During the meeting we also discussed other points of interest, such as added functionality that would be desirable to implement.

- The possibility to compare projects because of some standardized parameter is very attractive. We will use Test Case Points as independent parameter to quantify the size of a project. To do this successfully, computing the estimate will happen in two steps.
    - First, all parameters that define the *size* of the project will be translated in the number of test points.
    - Then, depending on relevant variables the test points will be translated into person hours and costs. These are variables that depend on *who* will do the work. Expertise is an example of such a factor.
- Depending on the phase and availability of information, the model must be able to handle both bottom up and top down approaches.
    - In the initial phase, the model must be able to compute an estimate based on test scenario information.
    - Later, when test case specific information is available, this too must be a possible form of input.
    - One extra level of detail would be test specific information.
- Task based enumeration will not be necessary since SDF testing is too large for that and it will never be accurate enough, due to ongoing changes and differences between project approaches.
- Using different factors for performance and wages the model may derive the cost savings for:
    - Outsourcing to India, compared to testing in any other country.
    - Automating a testing effort, compared to manual testing.

# 5 Conclusions and recommendations

## 5.1 Conclusions

The aim of this research was to provide Unisys India with a model that can accurately describe the costs related to their testing function. The main research question was:

1. *How can a model describe the costs and time the testing function in Unisys' Purva location, Bangalore requires to complete any testing project within 20 percent of the actual costs?*

After looking into the existing testing practices, we found that the model needed to consider different characteristics to describe these costs. Later, we found that some factors were redundant, while others needed to be added. In the end, we used the following characteristics to categorize software tests:

- Complexity for creating and executing (based on sub characteristics)
- New or existing
- Manual or automated
- Number of iterations
- Type; either functional, non functional, GUI, or UAT.
- Stage; module, integration or system testing.
- Domain; client server or mainframe.
- Number of environments.

Based on these inputs the model now calculates the number of test case points, and translates those into person hours and US dollars. It uses four different productivity rates, depending on the team skill and adds 35% for rework requirements. Also, the model includes a more detailed cost calculation function, which calculates required cost and effort based on a team with multiple skill levels, using linear equations.

Unisys makes cost estimations at different stages in the testing process, some very early while others are made later. Because of this, the level of detail in the available information differs significantly. To make sure the model is able to provide estimates in all situations we came up with three different input forms: test case based input, scenario based input and requirements based input. We have seen that that the results can be within 3.5% range using TCD based input, but more tests need to be run before we can guarantee that accuracy. It does however seem very unlikely that it will be above the 20% range mentioned in the first research question. Scenario and requirements based input have less accuracy and have not been excessively tested, so we cannot indicate a range for these inputs yet.

Later, a secondary objective was added, with the aim to create a new supply dashboard Unisys can use to make HR related forecasts. The related research question is:

2. *How can the newly created models, together with existing forecasting reports, be combined into a dashboard overview which provides forecasting figures related to manpower?*

We used existing headcount report, hire report, attrition report, demand report and bench report files to realize this. Currently the new estimation models are not integrated; they could in the future be connected to the demand report for increased atomization. At this time there is still some manual interaction required, but the dashboard does provide a very clear overview of the required and actual available human resources for a 6 month forecast.

## 5.2 Recommendations

In this section some points will be discussed by which Unisys can improve the model we created during this research.

**TCD based input**

This version of the model has been the biggest focus of the research. It is fully finalized and operational. However, if Unisys wants to be able to rely solely on this model, its results need to be verified more. Two options exist:

- Have someone input historic data in the model, and verify the models outputs.
- Use the model parallel to existing procedures until team leaders are confident of the results.

The first option will lead to faster confirmation of the models output, but does require a lot of effort. During this research this has not been done because historic data is not listed using the model's input characteristics. Therefore, if this is to be done, someone with enough expertise should be assigned. This is the only way to ensure that historic data is translated into the proper input forms, which is required for validation. The difficulty here is that this research was started because the team leaders that make these estimations have very little time. Adding such a big task to someone's responsibilities may not be a good move.

The second option requires far less extra work, but it will take longer before old methods can be fully abandoned. I do however recommend this approach, since only then managers will see the added value of the model first hand, and thus feel confident enough to make the switch.

During this process team leaders should also document the difference (percentage) between the model's estimates, the current estimates, and the actual data. Hopefully the results will be as good as the results we have seen when using Basha's data, 3.5%.

There may still be errors in the model that we haven't found during this research, using it in parallel may identify such problems. Should they be found, the model can easily be tweaked using the "table" sections. If someone does this he should verify the new results with other team leaders as well.

**Scenario based input**

This version of the model is a generalization of the TCD based input. It leaves out very little data, and should therefore get nearly the same results as TCD based input. Of course only if information is available. One important recommendation regarding the use of this input form is to fill in as many

scenarios as possible, so that each bit of known information is used. This is preferred to grouping of TCDs in one scenario.

**Requirements based input**

This input form has been created, but the scenarios we identified are in a very early stage. Before this form can be used Unisys should have some more brainstorming sessions where people give the requirements we identified more accurate scenarios. Once that has been done this should be used in parallel to current estimation procedures to see whether the requirements result in correct scenarios. This should be done before this form is used by the field team to make estimations. At this time the results are very uncertain, so the field team should not yet rely on it.

**Training**

I have given a small presentation using a PowerPoint document and the model's manual that have been created. If in the future more people will use this model they should have similar presentations, to make sure that everyone interprets the characteristics used in the same, correct way.

**Supply Dashboard**

The supply dashboard is fully working, and correct. It uses existing reports and simple mathematic calculations, so as long as the reports are correct, the figures in the dashboard will be correct too. In the near future it may be interesting to make the dashboard fully automated instead of partially. Currently, the user needs to fetch the reports himself, put them in a folder, rename them and then press a button. Although this is very little effort, it is valuable if it can be fully automated. This requires several things:

- The report files must always use exactly the same interface:
    - There can be no extra lines or columns added if full automation is desired.
    - Filenames must be standard, instead of the current _June and _revised parameters.
- Someone with enough VBA programming skills must write macros that will automatically download the files from the Unet server.
    - This requires the files to always be stored in the same location.

After using the dashboard for a few months Unisys should consider whether these requirements are worth the extra added value of a fully automated dashboard. During this research it has not been done because of time restraints and because we chose not to interfere with the existing reporting structure at this time.

# 6 References

## 6.1 Literature

Patel, Nirav and others, for Cognizant Technology Solutions (2001). *Test Case Point Analysis.* Verion 1.0

Andirsin, Jari (2004). *TPI – a model for Test Process Improvement.* Received on 24-5-2011.

Boehm, Barry (1988). *A Spiral Model of Software Development and Enhancement*.

Schwaber, Ken (2004). *Agile Project Management with SCRUM.*

*UNet - Unisys' Intranet. Employee accesible databases with information regarding a.o. Unisys' operations.*

## 6.2 Interviews

Many interviews with different people at UGSI have taken place. The names of these people will remain confidential.

# 1 Appendix A: Testing techniques

Appendix A contains more information regarding testing techniques. Several different techniques will be discussed. These techniques can all be categorized based on the characteristics discussed in the main research.

## 1.1 Static Tests

### 1.1.1 Reviewing and inspecting code.

Reviewing code can be done either formal or informal, and both public as private. It means having other people check your code for errors. Usually the aim of this test is to find and remove common vulnerabilities such as format string exploits, race conditions, memory leaks and buffer overflows. The latent error discovery rate is 60-65% for formal reviews and 50% for informal testing. Dynamic testing practices usually have a 30% rate.

The most common form of formal testing is the Fagan code review. Here a person or a group of people review code line by line, often using print outs. This procedure is very effective, but time consuming.

The steps in the Fagan approach are as follows:

- **Planning:** The inspection is planned by a moderator.
- **Overview meeting:** The author describes the background of the work product.
- **Preparation:** Each inspector examines the work product to identify possible defects.
- **Inspection meeting:** During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.
- **Rework:** The author makes changes to the work product according to the action plans from the inspection meeting.
- **Follow-up:** The changes by the author are checked to make sure everything is correct.

Common informal testing practices are the following: (can have equal results to Fagan review if performed properly)

- Over-the-shoulder – One developer looks over the author's shoulder as the latter walks through the code.
- Email pass-around – Source code management system emails code to reviewers automatically after checking is made.
- Pair Programming – Two authors develop code together at the same workstation; this is common in Extreme Programming.
- Tool-assisted code review – Authors and reviewers use specialized tools designed for peer code review.

The common criticism to the code review approach is that it costs a lot of time/ money, while the detected errors could also be discovered by the dynamic testing procedures (if executed properly).

### 1.1.2 Walkthroughs

Another common static test is a walkthrough. The author of the code reads through the code with a group of experts listening and asking questions on the software and choices he has made.

Compared to the Fagan approach the checking is done *during* the author's explanation; whereas in the Fagan approach the reviewer sits down to check for errors after he has had an explanation. The Fagan approach is more thorough, and more time consuming.

## 1.2 Dynamic Tests

Opposite to static testing, dynamic testing implies executing the code (or a section if it) with some test input. The procedure is to test the software with some input, and see whether the output is equal to the expected output. There are some dynamic tests that can only be performed white box style, whereas others can be operated in any box condition. The white box only tests will be described first.

### 1.2.1 White box dynamic tests

These include (amongst others) Code Coverage, Fault Injection and Mutation Testing.

#### *1.2.1.1 Code Coverage*

Code coverage is a test where a specified percentage of all code will be executed. If the program tester doesn't find any bugs and the required percentage has been covered the test is passed. Full code coverage (100%) is hard to realize in larger programs, but would prove the software to be fully functional.

A tester can create a test that will run each line of code at least once. There are several levels in code coverage:

- Functional coverage: each function in the code has been executed at least once.
- Statement coverage: each decision point in the program has been covered at least once.
- Decision coverage: each decision point in the program has been true and false at least once.
- Condition coverage: each Boolean sub expression has been both true and false at least once.
- Condition/ Decision coverage: both at the same time are covered. Very thorough test.

Examples of these coverage levels:

Function X outputs true if A && B, and false otherwise.

- Functional coverage: function X is called at least once.
- Statement coverage: Condition A && B has been called at least once.
- Decision coverage: Condition A && B has been true and false, for instance 1,1 and 0,1 have been used as input.
- Condition coverage: A has been 1 and 0 and B has been 1 and 0.

Condition/ Decision coverage is the most complete coverage, and can take a lot of time to realize 100%. Note that condition coverage alone doesn't guarantee decision coverage: here it can be achieved by testing 1,0 and 0,1. Both A and B have been 0 and 1, but the statement has never reached true state.

#### *1.2.1.2 Fault Injection*

Fault injection is a technique that uses on purpose error making to reach sections of the code that would otherwise rarely be triggered (for example error message sections). The code is reviewed, and a faulty input is carefully put together by the tester to make the program enter a specific code area. This technique is used to increase the level of code coverage.

#### *1.2.1.3 Mutation Testing*

Mutation testing is a tool to help the tester develop a proper test suite. The source code of the program is altered slightly but significantly, to where it is no longer supposed to pass the test. If the test suite identifies the error it passes the mutation test. If it does not, the test suite is apparently not able to pick identify all errors and should therefore be altered.

A simple example of mutation testing is as follows:

Function X adds input parameters A and B. The result will be outputted. One way to do a mutation test would be to change the + symbol to a – symbol. The function will no longer produce the desired output, and should therefore not pass the test. However, if the mutated function is tested and it still passes, it is clear that the test is not able to properly test the program (false positives). It should therefore be altered.

### 1.2.2 Other dynamic tests

The following dynamic tests can be performed in any box: equivalence partitioning, Boundary value analysis, All-pairs testing, Fuzz testing, Model-based testing, Exploratory testing and Specification based testing.

#### *1.2.2.1 Equivalence partitioning*

This is a technique used to reduce to total number of tests cases that need to be developed. If a particular section of software is supposed to accept a range of input to be valid, and all other ranges to be invalid, it may suffice to check if from each range one case gives the desired result. These ranges are called the partition, and this theory says that only one case from each partition needs to be tested to evaluate the entire partition.

Example: inputs 1-5 are accepted all others should throw errors.

The partitions are as follows: < 1; > 5; and 1,2,3,4,5

Now if the test suite uses -3, 4 and 7 as test inputs. If the program returns errors for -3 and 7, and the desired return for 4, the test is passed.

This technique is not a full guarantee that the software will work, since not all input is tested. But at least input from every partition is tested, which gives some information on the programs functionality. 100% testing is often impossible.

#### *1.2.2.2 Boundary value analysis*

This technique is similar to equivalence partitioning, but more thorough. Instead of testing only one value in each partition, all boundary values are checked for validity. Boundaries are common locations for bugs, and therefore require thorough testing. Usually the boundary itself and both sides near it are tested. In the previous example boundaries are 1 and 5, so 0,1,2 and 4,5,6 would be used as inputs.

#### *1.2.2.3 All-pairs testing*

This technique tests all possible combination for each set of input parameters. This can easily be automated, and is very useful because most bugs are caused by a simple faulty parameter or a combination of two specific values of two parameters. All pairs testing catches both these faults. All-pairs testing is considered a reasonable cost benefit compromise to very exhaustive, expensive methods, and on the other hand less thorough methods.

#### *1.2.2.4 Fuzz testing*

Fuzzing throws randomized, invalid inputs at the system, to see how the system reacts to these inputs. Should the system crash, the input parameters that caused it will be saved so the programmer can fix the error. Fuzzing can easily be automated, and because it generates many different situations that can cause an error it has a high cost benefit ratio. Fuzzing is not a thorough way of testing, because it can't guarantee to reach each line of code. Compare this to fault injection, where the tester can generate faulty inputs that *do* reach each line of code. For this reason fuzzing should not be used as a bug finding tool. Rather, it should be used as a tool for overall quality assurance.

#### *1.2.2.5 Model-based testing*

Using this technique the final test suite will be derived from several other elements. One of these elements is a model which describes all elements of the testing data, mainly the test cases and the text execution environment. This model is usually derived from another model which is a representation of functional aspects of the software that needs to be tested.

Once the test model is completed, it needs to be transformed into an executable form. It is possible that the test model itself already offers enough information to be transformed to an executable form, but often it does not. In case it does not it is up to the tester to decide what procedures he will use to transform the model. Because testing is mostly experimental and based on heuristics there is no best practice to do this.

Once this is done there are three different ways to proceed:

- Online testing: the modelled test suite creates input to test the system, and automatically feeds this into the system. (dynamic testing)
- Offline automated testing: the modelled test suite creates input to test the system, saves it, and feeds it into the system at a later point in time
- Offline manual testing: the modelled test suite creates input to test the system, which can be interpreted by humans, and then humans feed it into the system that's being tested

#### *1.2.2.6 Exploratory testing*

This is more an approach to testing than an actual procedure. The opposite of exploratory testing is scripted testing, where the inputs and corresponding desired outputs are known at start. The tester then sees if the software generates all desired output to each input. In exploratory testing the desired output is not necessarily known. The tester tries to understand the software, and inputs both difficult and easy cases, and sees how they are processed. He will then critically investigate the result.

An advantage over scripted testing is that less preparation is needed and mayor bugs are easily found. Also deductive reasoning can be applied, which scripts cannot do. A disadvantage is that it is often unclear which tests have exactly been run and which sections have been tested thoroughly and which haven't.

#### *1.2.2.7 Specification based testing.*

Here a test suite is built based on a specification. Usually the test suite is built based on a criterion that must be satisfied, for instance a specific percentage must be met using code coverage.

This is similar to model based testing, although it can be more specific. This is because models are usually built as a basis for test suites, whereas specification based testing can use specific criterions which only apply to the software at hand. Model based testing is therefore more general than specification based testing.

# 2 Appendix B: More lifecycle models

In this appendix you will find more common lifecycle models. Unisys does not use these models, they are only mentioned here for a more complete understanding of the lifecycle principle.

## 2.1 The V shaped Model

This model is similar to the waterfall model, except testing is considered a separate process, and testing development will run parallel to software development.

While the system concept is being documented, a testing plan is setup too. During the identification of system requirements, the requirements for system testing are setup as well. When the project moves to Architectural design, integration tests will be built in parallel. And once the detailed design phase starts, work will also begin on building Unit Tests. This procedure allows for testing to be done parallel to the design process instead of doing it afterwards.

Like the waterfall model this model too is pretty straight forward and will move to implementation phase fairly quick. The downside is that like in the waterfall model here too there is no room for volatility in requirements.

## 2.2 Prototyping

When prototyping, the developing team first makes a working prototype of what they believe suits the customer's needs. This is then shown to the customer, who will have the opportunity to give feedback. This allows for easier change, because the software has not reached its final state when the feedback is received.

Another advantage is that not all project resources are needed at the beginning of the project, because it takes longer to get to the implementation phase than waterfall/V models.

Finally, this model also allows for more volatility in requirements, because of the feedback moment where users can clarify their needs more precise.

One downside of this approach is that total development will take longer, as prototypes may be thrown away or drastically changed. If the requirements are too volatile, there may be a lot of useless programming which can increase the costs of the project significantly.

This approach requires the programmer to have a good understanding of the area the software is built for, as many aspects will not have been defined by the user at project start. To build a working prototype the programmer may have to fill in some of these aspects himself.

## 2.3 Incremental model

In incremental development, the system is developed in different small stages, with each stage consisting of requirements, design, development, and test phases. In each stage, new functionality is added. This type of development allows the user to see a functional product very quickly and allows the user to impact what changes are included in subsequent releases.

Here too, not all resources are needed at project start because of the longer timeframe. This approach should be used in complex projects where there is little understanding of the requirements at project start.

The mayor downside is that if requirements change, parts of the software that are confirmed working (i.e. fully programmed, and thoroughly tested) have to be changed. This can be very expensive. Because of this, the incremental model should be used when requirements are not volatile, but on the other hand not easy to fully grasp at project start.

## 2.4 The Spiral Model

The spiral model too is an iterative approach to software development. Here iterations are referred to as spirals, but have no different meaning. The model has many similarities when compared to the incremental model, however there is one major difference: the steps in the development process are based on risk.

Risk in software development refers to the degree of difficulty to develop an area of code or a function. Every following iteration addresses the next functions that seem to be the most challenging and hard to code properly. Each spiral consists of: determining objectives, alternatives, and constraints, identifying and resolving risks, evaluating alternatives, developing deliverables, planning the next iteration and committing to an approach to the next iteration. Barry Boehm (1988).

The pros of this model are that it allows for a very complex project with incomplete understanding of requirements, because development is done in small phases. Also it allows for high volatility in requirements.

# 3 Appendix C: RUP Development lifecycle

In the main document the following tools were mentioned. RUP uses these tools to realize better integration of testing throughout the software development lifecycle. The tools will be explained briefly in this appendix.

- Making testing a distinct discipline
- Using an iterative development approach
- Scheduling implementation based on risk
- Continuously verifying quality
- Letting use cases drive development
- Managing change strategically
- Using the right-sized process

The first step is making testing a distinct discipline. The RUP creates small groups in which activities are sorted. One of these groups will be testing. For each group their contribution to the project is clearly defined, as well as its interaction with other groups. This ensures that at project start its clear what each group uses as input, and what output each group is supposed to deliver.

Every group will have a workflow schedule that it will follow during each iteration. The workflow schedule for the testing group is shown in figure 3.1.

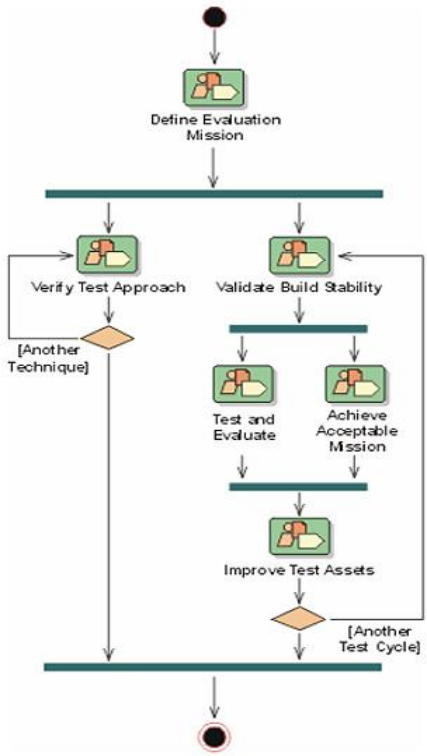


Figure 3.1 Test group workflow schedule

Furthermore, within each group clear roles are defined which all need to be filled. One person can take on multiple roles if the project is small. The testing group has four roles, shown in the figure 3.2.

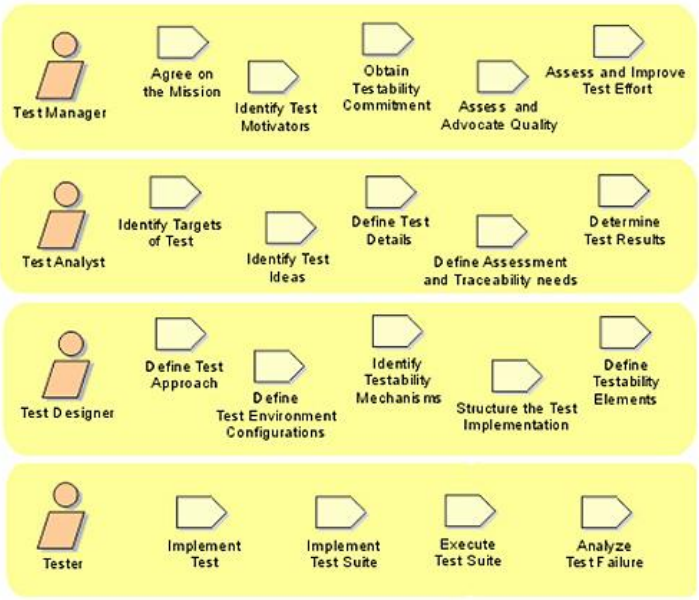


Figure 3.2 test roles in RUP

The second and third tools are using an iterative development approach and scheduling implementation based on risk.

At project start a planning horizon is created. It includes the number of cycles and the length of each cycle. The length can differ, based on resources available, size and difficulty of the project. Typically it ranges from 4-6 weeks. The most risky and hard to develop parts of the projects are done at project start, while more easy areas are done later in process. This might seem obvious, but using other models easy development is mostly done at start because this allows for quick and large steps, which make it easy to show progress to investors. Implementing risky parts later may however become impossible and require costly changes.

If figure 3.3 the RUP development cycle is compared to that of the waterfall model.

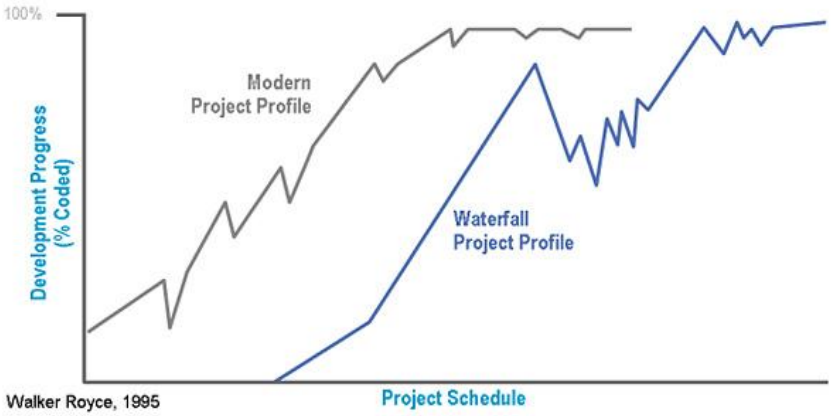


Figure 3.3 RUP and Waterfall compared

The next tool is continuously verifying quality. The RUP has created the small groups with clear task descriptions for each role a person can have. The next step is creating quality check points for all artifacts, activities and other deliverables throughout the project. This ensures that everyone on the project has a feeling for the desired quality and excludes people who are merely creating documents because it is their task. It is very irritating for project members to see someone delivering poor work and getting away with it because there are no quality targets.

Letting use cases drive development is a tool through which RUP clearly differs from other models. At project start a use case diagram is developed, which focuses on end user perceived value. Use cases don't show how internal systems work, as this is not relevant to the end user. At the end of each cycle the work is compared to the use case. This ensures that unlike in traditional models, work is checked for end user value, instead of working internal components.

Managing change strategically is a tool created to prevent big code changes near the end of development. Using traditional models there is often a gap between the testing and development teams. This gap results in development believing software has been approved by testing up till a certain level, whereas testing may believe there are still areas of code being worked on because they haven't passed the tests yet. Often this results in mayor setbacks close to the final deadline, because only then it is signaled that some vital areas are not fully working. Figure 3 illustrates how these setbacks can occur using the waterfall model. These setbacks can however be far more destructive to the code, delaying the project.

RUP solves this issue by adapting a change management strategy which allows testers to communicate with the development team directly, and change the priorities in the development process to ensure important bugs are filtered out when they are found.

The final tool is understanding how to implement RUP. RUP allows for very clear guidance throughout the project, by creating many artifacts, groups, targets and such. However, for smaller projects this may seem like a lot of extra work without any added value. RUP is scalable to any project: smaller projects may not need a formal structure and a lot of controlling processes, whereas larger projects may need more tools and all the processes available to allow for open communication between testing and development.

# 4 Appendix D: Scrum planning and methodology

In SCRUM methodology three different roles are defined:

- Scrum Master, also known as project manager
- Product owner, who represents the client
- Team member, member of a cross functional group of usually 7 people who do the actual work (developing, testing etc).

The team works through 'sprints' which are periods of 2-4 weeks. Each sprint starts with a sprint planning meeting, in which the team agrees on the highest priority tasks. The product owner also states what he wants to have done after the next sprint. Each sprint should end with a working version of the product, which can be presented to the client. If targets are not met during the sprint they are returned to requirements for the next sprint.

A key principle in Scrum (and any other agile method) is that it realizes that during a project the client can change its mind about what he wants and needs and that unpredicted challenges cannot be easily addressed using traditional predicting or planning techniques. Therefore, Scrum uses an empirical approach, and accepts that the problem cannot be fully understood and defined at project start. Instead it focuses on maximizing the ability to deliver quickly and comply with changing requirements.

During a project the following schedule is used, created by Ken Schwaber (2004):

**Daily Scrum**

Each day during the sprint, a project status meeting occurs. This is called a daily scrum, or the daily standup. This meeting has specific guidelines:

- The meeting starts precisely on time.
- All are welcome, but normally only the core roles speak
- The meeting is timeboxed to 15 minutes
- The meeting should happen at the same location and same time every day
- During the meeting, each team member answers three questions:[9]
- What have you done since yesterday?
- What are you planning to do today?
- Do you have any problems that would prevent you from accomplishing your goal? (It is the role of the Scrum Master to facilitate resolution of these impediments, although the resolution should occur outside the Daily Scrum itself to keep it under 15 minutes.)

**Sprint Planning Meeting**

At the beginning of the sprint cycle (every 7–30 days), a Sprint Planning Meeting is held. Work for the following sprint is selected.

- Prepare the Sprint Backlog that details the time it will take to do that work, with the entire team

- Identify and communicate how much of the work is likely to be done during the current sprint
- Eight hour time limit
    - (1st four hours) Product Owner + Team: dialog for prioritizing the Product Backlog
    - (2nd four hours) Team only: hashing out a plan for the Sprint, resulting in the Sprint Backlog

At the end of a sprint cycle, two meetings are held: the Sprint Review Meeting and the Sprint Retrospective.

**Sprint Review Meeting**

- Review the work that was completed and not completed
- Present the completed work to the stakeholders (a.k.a. "the demo")
- Incomplete work cannot be demonstrated
- Four hour time limit

**Sprint Retrospective**

- All team members reflect on the past sprint
- Make continuous process improvements
- Two main questions are asked in the sprint retrospective: What went well during the sprint? What could be improved in the next sprint?
- Three hour time limit

First, the reviewer identifies the main tasks that need to be achieved during the process, and multiplies them by a difficulty factor. The output is the number of person hours required for the project. To that a risk factor is added, resulting in the initially estimated hours that will be billed to the client.

The difficulty factor is a factor on the scale C1 to C4. For each project the reviewer can determine the number of person hours associated to each factor. This means that a significant portion of estimating is done on gut feeling and expertise: the number of tasks need to be identified, the difficulty of these tasks, as well as the number of hours each difficulty requires.

There are however more points further down the line where more planning is done. At these points in time there is far more insight in the actual required efforts and types of tests that need to be run. At the beginning of each testing iteration the evaluation mission is generated. Part of that task is to determine what tests need to be run the coming period. During the initialization of the iteration new estimates can be made, to check the earlier estimate for correctness, and to have more insight in the allocation of the costs.

Even further down the line there is also historic data available on the previous iterations. That means that the difficulty factors (C1-C4) can be reviewed and if necessary revised.

The difference in input means the model should be able to cope with more detailed input as the project continues. For the initial estimation it is desirable to break the work down in large chunks, multiplied

with the difficulty factor. Later the actual number of tests and type of tests can be input to allow for more detail. Of course, it should also be possible to input detailed information at project start, but this is most unusual.

Another way of estimating the work to be done is by standard fractions. This procedure is mostly used when the estimation also includes the costs for development, not only testing. The procedure is as follows:

The reviewer only considers core tasks, known as Coding and Unit Testing tasks, or CUT. From the request the reviewer identifies all the core tasks, and assigns required hours to them. Then, to compute the total required effort the reviewer multiplies the CUT by standard fractions. As a rule of thumb the following fractions are used:

- Requirements Specification 15%
- Design 20%
- Coding and Unit Testing 30%
- Integration testing 20%
- User acceptance testing 15%

One other technique that could be used is function points based estimation. Unisys has a standard procedure to award function points to a program. The points are based on several different parameters, such as lines of code, number of functions, operating system(s), etc. It should be possible to translate these points to a cost/effort estimation that can be used by the testing team.