HAAGA-HELIA
University of Applied Sciences

Optimizing software testing life cycle: reducing effort use in integration- and system testing

Teemu Vänskä

Thesis
Business Information Technology
2008

HAAGA-HELIA
University of Applied Sciences

Abstract

24.11.2008

Business Information Technology

Authors Teemu Vänskä	**Group** BITE
The title of your thesis Optimizing software testing life cycle: reducing effort use in integration- and system testing	**Number of pages and appendices** 26 + 2
Supervisors Raine Kauppinen	

For a software company, it is very important to have a functional process that guarantees smooth work flow, and that is comprehensive for all roles of the company. But not all processes are suitable for real life; they only sound good in theory. I decided to make a research on different processes to see if it was possible to create a custom process to a company that is in a need for one.

To narrow the subject to be more specific, it was necessary to specify the target areas of the research, system testing and integration testing. For the research it was also important to consider the possible company roles this investigation would probably affect the most. Those roles were a software tester, a software developer, and a software specification writer.

Before the actual "brain storming" for the new solution could even take place, it was clear that some of the well known and already made software development processes would have to be reviewed and studied in order to have a better picture of the composition of a development process. Just studying processes and models wouldn't have been enough; also the question "What kind of tests are there, and how are they done?" should be answered. That's why I had to review different testing methods for this project also.

After studying processes and models, testing methods, it was time to see how all the theories differed from reality, and what parts were the most likely reasons for those differences. Only after those analyzes it was possible to pursue the ultimate goal of this research.

But the result wasn't what I at first expected. It wasn't a magnificent scientific breakthrough, but vice versa; it was a moment of clarification. Only a theory of an improvement was achieved during this research, a collaboration of different process models mixed as a hybrid version, but the lesson itself was far greater than I would've ever imagined. It also made me see that I clearly wasn't the only one thinking about these things.

There was no time for real life experiments. Continuing this project in the future would be interesting indeed, to see if my theory would actually work in practice. But that would require a lot more time, and I definitely would like to hear a second opinion of the theory.

Key words
Software testing, software testing method, software development process

Table of contents

1 Introduction

The value of software testing is growing day by day in small software companies where budgets are marginally small, and have no space for redundant work or time loss. The amount of software testers in such a company is very small so the efficiency of the actual testing is a priority. The downside has always been an unclear workflow of testing, since there are no standards for it, at least not for real life. After working about a year in a medium-sized software company, I've realized that none of the already written testing processes or methods applies perfectly on what is happening in reality. The amount of redundant testing is outrageous and easily noticeable even for a novice. This kind of an action leaves a stain to the picture professionals' skills, and lowers the company's creditability in the software market.

So I came up with an idea of developing a new or improved method/process for software testing lifecycle to enhance the outcomes of testing, and to lower the cost of it. By creating a prototype model of a new standard for software testing it could make the whole procedure swifter and smoother with lesser errors or repeats. Majority of this kind of working method is based on solid rules and regulations, rather than on completely new workflow of testing lifecycle with no variables.

The goal of this project is to find those time spending risk factors and reduce them, even eliminate them if humanly possible increasing the accuracy and speed of software testing life cycle in integration- and system testing mainly for life insurance software.

The new/improved software testing lifecycle method would be desired amongst the software testing community. The benefit would not only help the actual testers, but also the developers and the specification writers who are dealing with the testers every day. And since business rules are relevant part of software testers' lives, not all of the redundant testing is made by them but also the client's. Finding a mutually beneficial way to test one's program would deepen the bonds of co-operation between the software company and the client, and ensure a long partnership.

2 Integration- and system testing

The main reason why I chose the integration testing and system testing phases of software development process is that in those phases most of the usual problem/challenges occur. The second reason was that I've been doing both of them for almost ten months now and have noticed those problems/challenges at some point. For clarification I've chosen the "V-model" to show where exactly this study focuses on.

Integration- and system testing are both located in the middle of the V-model of software development process (see Figure 1). At the bottom of the V-model is unit testing which has its own part on this study, but is not concentrated on as much as integration- and system testing. The same applies to acceptance testing, on top of the V-model.

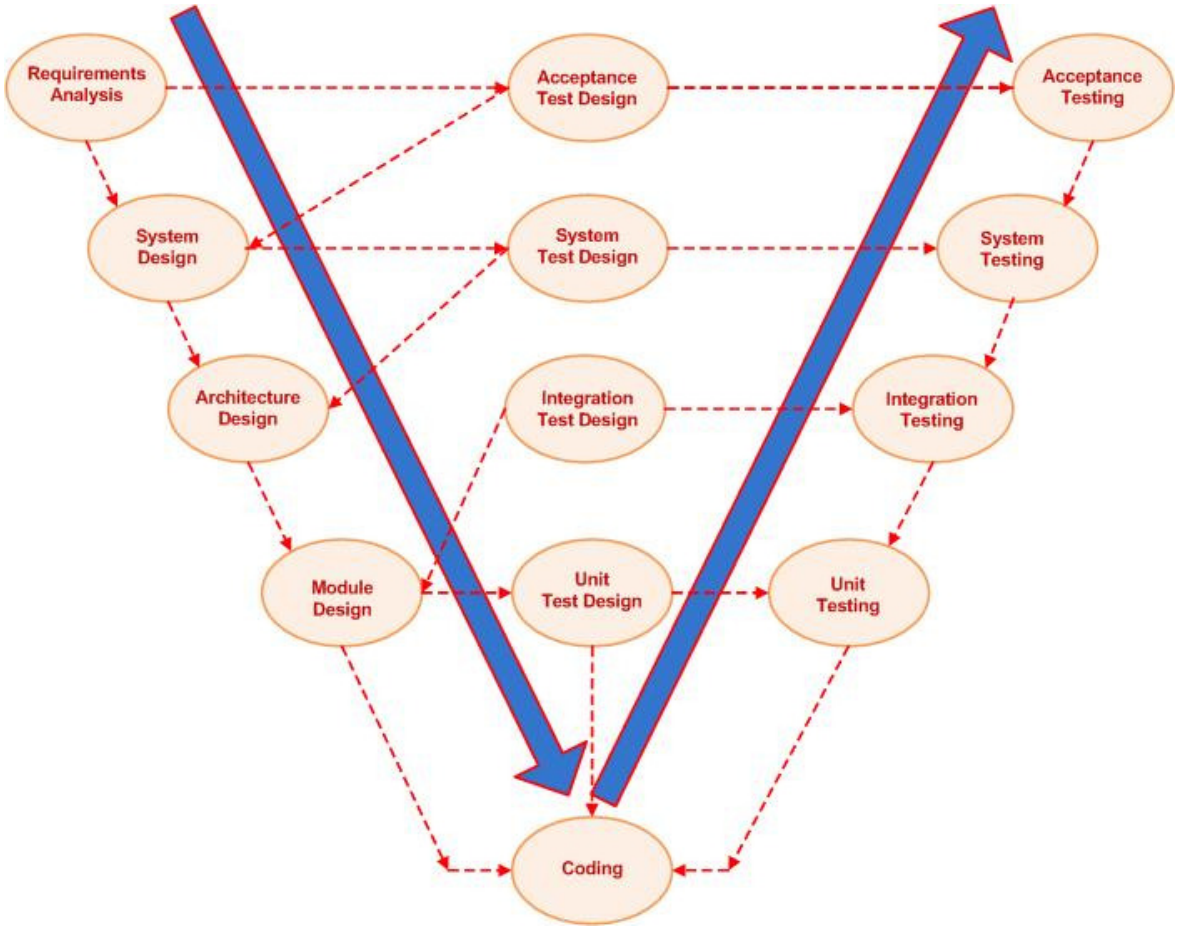

Figure 1. The software development process called the V-model (Wikipedia 2007.)

2.1 Integration testing

The purpose of integration test is to make sure that interaction in a system between multiple components and data transfer work properly without flaws. There is more than just one level where integration testing takes place. The developers usually are responsible for the integration testing at the lowest level; they are the ones who check if all units work well together. The developers can also do the higher level integration tests, but usually software testers are the ones who take the lead at this stage, since the biggest focus is on the interfaces which are usually software testers' speciality; to look if all the units of the system work properly together. (Craig & Jaskiel 2002, p.130.)

As the integration test is about many different parts of a system working together, it is done usually near beta testing when it's possible to combine the newly created software with e.g. old system parts which are meant to interact with the new software. In some cases these can be done with a customer, since they have more knowledge of how the new system should work. Integration tests require quite a lot from the testing team, as it may continue very long, even after the deployment of the software, and because of that, the testing team has to try to check every aspect of the system in order to have the upper hand over the customer. (Loveland, Miller, Prewitt & Shannon 2005, p.38.)

But it must be taken into a notice, that usually even the most advanced or qualified test team hardly ever succeed to create a test environment which would be identical to the customer's own system. It can be similar from the outside, but the units can differ. This is the point where good relationships are required with the customer, and a mutual trust must be achieved in order to have the customer's knowledge of how the their own system is created, how the units work, and how they're configured. This knowledge helps the test team create an environment as similar as possible to the customers own environment, helping them to understand how it functions, and make better tests. (Loveland et al. 2005, p.39.)

2.2 System testing

Where integration testing focused more on how different units in the system work together, system testing has a lot vaster area of tests involved. In a medium sized software company there are hardly ever several test teams, so at this point they have a lot on their hands. System test includes usually functional tests, load tests, performance tests and reliability tests, not to mention that system testing can last very long, depending on resources and needs. So these are probably the reasons why software testing is often generalized as system testing. And since the system test is considerably large part of the whole testing of the software, most test cases and sets are stored and reused when regression testing takes its turn. (Craig & Jaskiel 2002, p.121.)

For system testing there has to be a system test plan. This plan is usually created by the manager of the testing team, or a senior software tester who is the one responsible on coordinating users' and developers' focus on areas that matter. Sometimes the test team collaborates with the development team and the customers to have a shared view of the plan. There are some prerequisites for the system test plan: the requirements from the customer which have been modified into specifications of the system, and then the software design documentation. The writing of the system test plan can be started earlier, but it is not recommended. (Craig & Jaskiel 2002, p.121-122.)

No developer or tester is perfect, and that is a sad fact. After long periods of tests, there comes a time when the customer gets his hands on the software, and founds a defect. This can happen during a life cycle test in approval tests with the customer, or later when the deployment has already been made. Once a defect is found, it is reported and studied until it is identified and provided with a fix. The same kind of tests similar to the system tests have to be performed when the fix is applied, and a new release of the software has been published. And these things are not to make haste with, since now that the customer has seen a flaw in the system, they will only get more accurate and agitated with the test, and testers. (Loveland et al. 2005, p.40.)

3 Company roles and testing challenges from their point of view

Usually the company roles are narrowed down into three different personnel who have something to do with software development. These three roles are testers, developers, and specification writers. All of these roles are tightly together when it comes to software testing, for all of them are dependant of each other. But as an individual, they all see defects and challenges in different perspective as the next one, so this is the point where team work steps in.

Companies also have different teams which include more personnel, but who are only dealing with e.g. marketing and business issues. Sometimes they can be consulted about some specific business rules, but only rarely since usually there's always someone who can answer questions without spending "outsiders'" time.

3.1 Tester

For this part I can speak for myself. Testers are specialized on the software product in hand, and are skilled in finding defects from different features. They are familiar with the user interface (later referred as UI) and with normal basic functions of the software product. In this case, they are also very close to the customer, and interact with them all the time. Normally testers' work consists mainly of using the software product, finding defects, and reporting them into a defect database (later referred as DB). Their job is to identify anomalies in the product, investigate if they really are defects, create workflows for the developers so they can re-create the situations, and finally test the fixed defect. Not all of the defects are internally found; sometimes defects come directly from the customer and are not in the same format as our own, and they need constant consultation from the customer. Since testers are considered experts, professionals in spotting right from wrong, they also often get caught in cross fire of the customers and their own team. Those situations require lots of time, because no one wants to make a bad call about a defect which may affect the whole product, now or in the future.

3.2 Developer

Developers, or as some people call them just programmers, are the backbone of every software company. They are the people behind most of the software product, or at least its functionalities. They hardly ever search or find defects by themselves, but they are the ones who are called the second a defect has been discovered. A software product, e.g. life insurance software, contains numerous different areas of development work, such as UI, mathematics, possible printouts and so on, so each of the developer teams have their own specific area they're specialized in. Their knowledge about the product's functionality is vaster than normal tester's but slightly constricted compared to specification writers. Developers have to deal with the defects face to face unlike the testers from whom they receive the tasks.

Their responsibility is also identifying the location of the problem in hand and making an accurate fix that won't jeopardise the product's other areas later on. Usually even though the developers have done a good job, the get a message a while after a fix that some other part of the product has been damaged by the newly added fix. This is usually due to their lack of knowledge on the product itself, or the flow of using the UI. Developers walk hand in hand with testers and solve problems together, filling out the blanks of each other's abilities. But unlike testers, developers hardly are in contact with customers since their way of dealing with the program differs so much from normal user's way that it would be more or less unnecessary to involve them to the customers. Testers are for that, with the consultation of developers of course.

3.3 Specification writer

Easily the most wanted person on software development process, and due to this also the busiest role of them all. They have the knowledge of the product beyond anybody else and are the last straw when identifying defects. These people write specifications to the functions of the software product which make them experts in all areas of the development process, except for using the UI. The life cycle of the testing process begins from specifications, and if there are no specifications there can be no testing either (Hutcheson 2003, p.12.).

Since software products are evolving all the time, specification writers do not have much time to help testers or developers with direct guidance, but they do show the correct specifications where to look for answers which is usually enough. Although that might sound a bit helpless for a tester or a developer to ask other people to show the correct documents that contain the correct information, but remember, the amount of functionalities of a software program can go easily up to several tens of thousands which all have their own specification of how they should work, and when they should work. Finding the correct document would take a normal tester or a developer a day or two, when founding the same document takes only five to ten minutes for a specification writer. But as skilled as they are in the field of pure knowledge of the product, they have very little information about the product UI, and are usually on the background when testing the product.

4 Studying software development processes and models

"Cheap, fast, good: pick any two." Many times, testers face ridiculous deadlines with shortage on resources which is the gist of the before mentioned quote. It is quite impossible to achieve all three attributes with limited time or resources, and the most often chosen attribute by the testers is quality above all. Then it's up to management team whether the software is made quickly but expensively, or slowly but cheaply. Defects are the ones that can turn the boat upside down in a development cycle. That's why it is very desirable to spot the defects as soon as possible to avoid any extra costs. The later a defect is discovered, the more it will cost to the company, as time goes by the "price of a defect" grows exponentially. Although there are occasions where even a defect discovered at late hour hasn't altered a thing, and that is because the defect was discovered by an individual who can fix it by himself/herself without involving anybody else along. But this isn't the case in most defects. Usually the defect is found by a tester or a customer, which makes fixing a lot more complicated and more expensive. People from the testing team, the development team, and maybe even from the specification team have to work on the same bug to get it fixed, and after the fix it has to be integrated to the next version of the software by a builder. Just imagine the resources and working hours spent on one defect that for some reason has managed to get by the developer's eye. (Loveland et al. 2005, p.27-28.)

That is also why software companies need professional testers instead of only relying on the developers; it is only realistic to divide workloads into several different roles that have their own field of work they're specialized in. Some idealistic could say that there is no need for testers if the developers find the defects all by themselves, but that is not possible; it would require remarkable talent from a developer not to make one single mistake into his/her code, or to discover the flaw before forwarding the code any further. This has already been noticed when different development processes and models were designed. Their function is to make the flow of software development fluent and smooth without losing any of the three desired attributes, as time and cost are always major factors to the management team, and of course to the customers. (Loveland et al. 2005, p.28.)

4.1 The waterfall model

Probably the most known software development models by its straightforward and consecutive style would be definitely the waterfall model. What makes it so straightforward or consecutive? Simply said it means that the process is consisted of steps that are meant to execute in certain order where the next step cannot take place until the step before that has been completed (see Figure 2). The steps are extremely logical; first you get requirements from the customer, then you create specifications from them, you design the software, and then you start the actual development. (Loveland et al. 2005, p.43-44.)

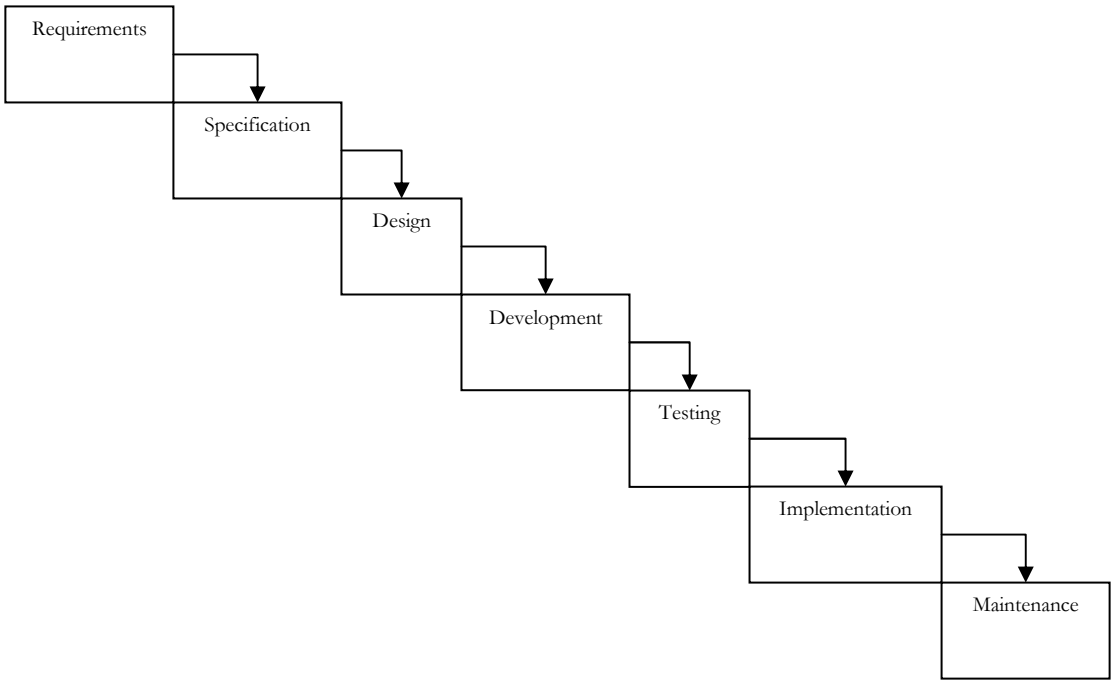

Figure 2. Waterfall software development model (Loveland et al. 2005, p.44.)

Now, the waterfall model is based on following the steps strictly so that the software is developed carefully, and checked inside out before releasing it. This model still has one big weakness in it. As the steps are dependant on each other, the whole process becomes extremely slow which isn't the most desirable attribute if a software company tries to stay as no.1 on the market. So this tends to make the developers and testers act in haste so the pace wouldn't die too fast, and that will cost them latest at the implementation phase when beta testing has just ended and maintenance is about to start. (Loveland et al. 2005, p.44.)

4.2 The spiral model

Another iterative software development model was presented by a software engineer Barry Boehm: a spiral development model. It was designed to avoid the same weakness as the waterfall model has, and to give the viewers a new angle in approaching software development. Why the name spiral model is quite self-explanatory (see Figure 3), but the development process itself separates this model from the others. (Loveland et al. 2005, p.52.)

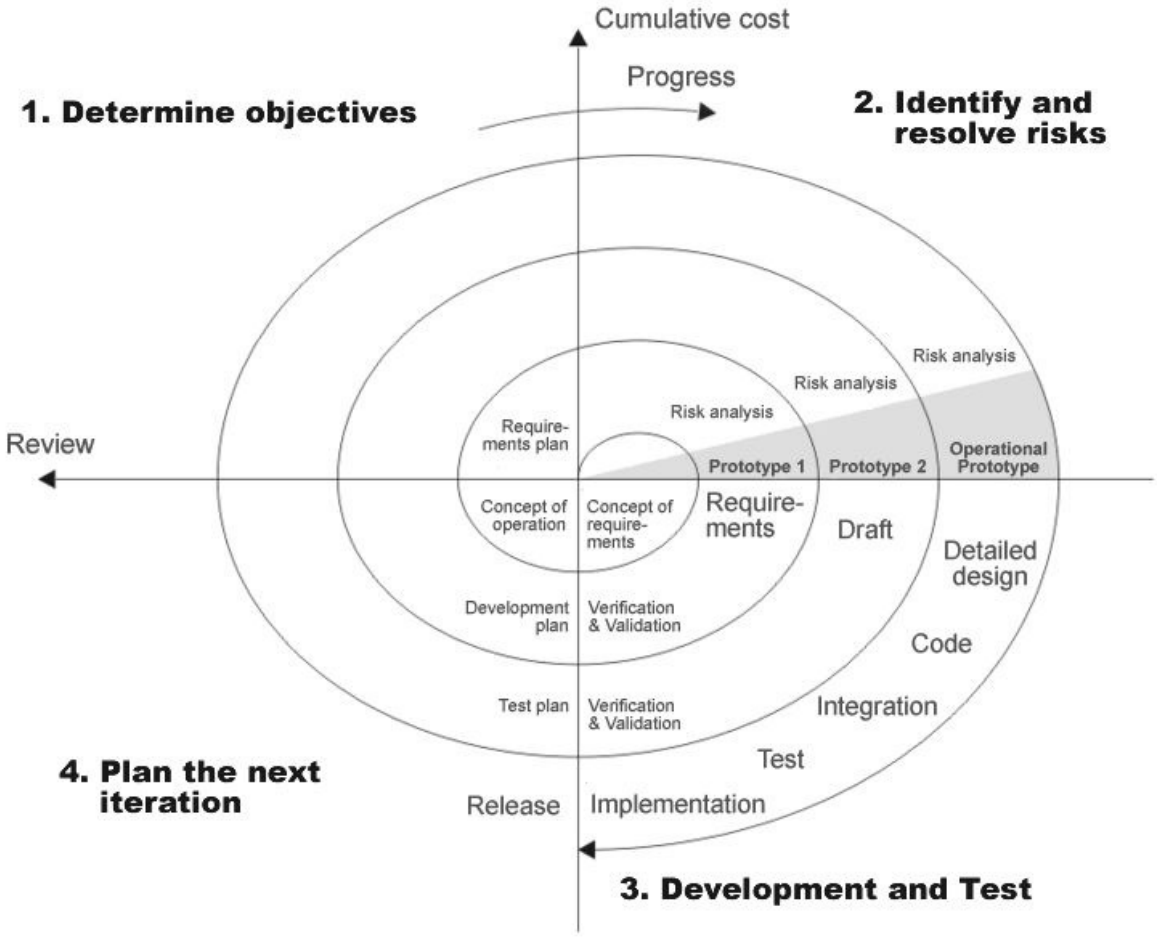

Figure 3. Boehm's spiral development model (Wikipedia 2008.)

The progress of this particular model is very straight forward, starting from the inner rims and circling clockwise to the outer rims, going through all processes presented in this model: object definition, risk analysis, development and engineering, and finally the planning of next phase. Every phase will experience these processes in each rotation of the spiral, which are the base for this model in order to be effective. (Loveland et al. 2005, p.53.)

4.3 Waterwheel model

The waterwheel model differs from the classic waterfall model mostly at between the development and test phases. The slowness of the waterfall model was that the finished code was delivered as one big package which would easily fall apart as the code is implemented; more defects pop out and the implementation gets delayed as the development has to concentrate on the problems in hand. But instead of delivering the code as one big package, the waterwheel model maintains its balance by delivering small portions of the actual code in well defined packages which allows the development team to continue their own work and prepare the next package while getting live feedback from the test team. By applying the software as smaller implementations it reaches the markets faster, and at the same time gathering of requirements, turning them into specifications, and designing the software product itself may continue at the background. (Loveland et al. 2005, p.45.)

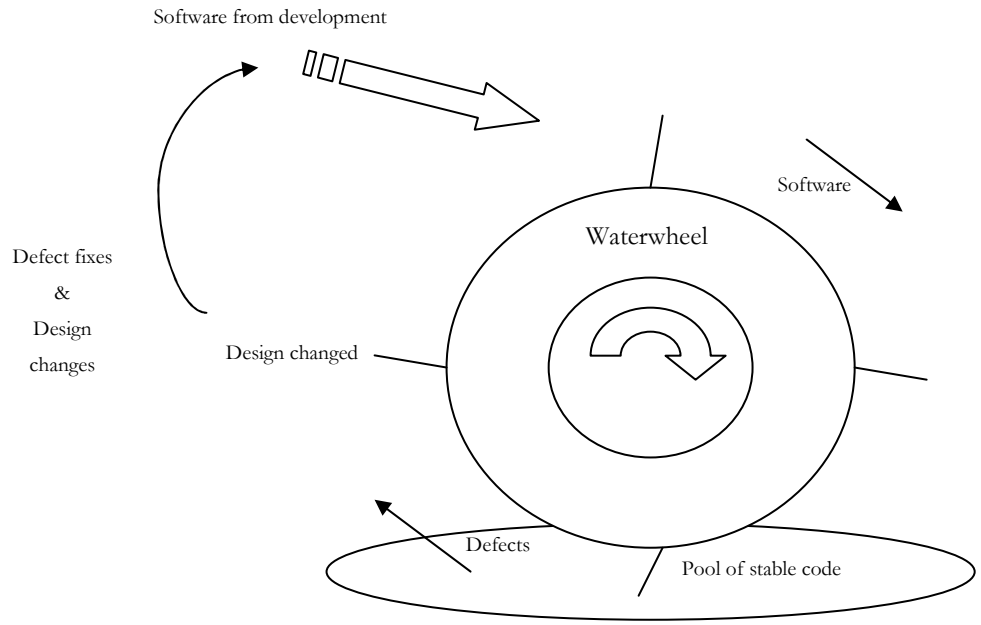

Figure 4. The waterwheel model (Loveland et al. 2005, p.47.)

4.4 Evolutionary model

The evolutionary development model could be called as a combination of the waterfall model and the waterwheel model. It proceeds from one phase to another while having feedback loops which inform the development team on any changes regarding the software design or the process itself. (Loveland et al. 2005, p.55.)

> May and Zimmer outline their use of the evolutionary development model (EVO) as a set of smaller iterative development cycles. They point out that breaking the implementation phase of the software development process into smaller pieces allows for better risk analysis and mitigation. (Loveland et al. 2005, in May 1996, p.55.)

To point out the main difference between the evolutionary model and the rest of the models, I'll refer once more to the weakness of the waterfall model: the feedback. Instead having the only feedback at the end of the development phase, the evolutionary model has inlayed feedback loops in small waterfall cycles which have a shorter lifespan than a normal waterfall model, normally from two to four weeks. These "small waterfall cycles" have all the same phases as the bigger version, from design to initial testing. The smaller cycles overlap the next ones as the feedback is studied from the earlier cycle and the next one is executed. (Loveland et al. 2005, p.55.)

This sounds a lot like the waterwheel model, but there is a difference between the waterwheel model and the evolutionary model. The main difference is in the type of feedback loops. The waterwheel model is mostly done as an inside job, and so is the testing of the software. The feedback is delivered only by the internal test team, made by professional testers using white-box and black-box methods, and report their founding directly back to the developers. This is what I'd call an internal feedback loop. The evolutionary model is more outside oriented than the waterwheel model. In the evolutionary model the temporal version of the software product is handed directly to the customer to be tested, and the feedback is provided by the customer itself. This makes the evolutionary model even faster on delivery than the waterwheel model, but it also can be more unstable than the waterwheel model since its brief testing and early delivery. Depending on the client and its testers, there can be numerous change requests to the development team, and the number of redundant defects can multiply, unlike in the waterwheel model where all testing is made by professional testers. (Loveland et al. 2005, p.55.)

5 Studying testing methods

There are no software companies who wouldn't dream of non-bugged software products. Most certainly there is software products that have no defects – only change requests made by customers – but there's not one software product that has had all its defects discovered at the earliest test phase, I guarantee you that. There are different methods for different phases and testers, but not one of them is suitable enough at discovering them all before the process continues to next phases. One reason for that is that not all of the defects are discoverable before later phases or before different testers who look at the software product from a different angle than the already oriented professional testers. (Loveland et al. 2005, p.28.)

5.1 Black-box testing

Also known as data-driven or input/output-driven testing. So where does it get the name black-box testing? I imagine this method would be suitable for persons who have no experience in testing at all, or inside knowledge of the product, since the point is that you imagine the software product as a black box that you can only see from outside. Inside this black box is the products mechanics and logic, but you are not aware of those things – you can't see inside of the box. You have only the knowledge granted by the specifications, and the goal is to search anomalies from the product based on the rules that are defined in the specifications. Now you understand why I thought this could be easier for people without any experience in testing, or who'd have inside knowledge of the product; because for a professional tester it can be amazingly hard to see the product only from the outside. (Glenford 2004, p.9.)

5.2 White-box testing

Unlike black-box testing, white-box testing (also known as clear-box, or logic-driven testing) has also the inside view of "the box". You have the knowledge of product's logic which you use to test the product. The goal is to have all possible functions tested, but usually the goal is never achieved due to a simple fact that the number of functions (and their paths) can be incredibly large e.g. like in a life pension software. Though it is of course possible to test them all, but nevertheless there still can remain defects that are discovered only by executing functions with illogical paths that are not described anywhere. Usually these kinds of defects pop up when a user unintentionally makes an error (like a mouse miss click) during a process. (Glenford 2004, p.11 and 13.)

5.3 Grey-box testing

The grey-box testing is a mixture of black-box testing and white-box testing. You can have the knowledge of the internal data structures and algorithms but the actual testing is done using the same method as in the black-box testing. Since the actual test is a black-box test, the tester is not allowed to manipulate input data nor format the output; the internal data structures and algorithms are allowed to be used only when designing test cases. The ideal use of a grey-box test would be in an integration test, where there are different modules coded by different developers, and for the test only the interface would be visible. (Wikipedia 2008.)

Grey-box testing is mainly meant for the professional software testers or developers who would more likely have the inside information of the software product as well. For the people who do not know about the inside structure of the software product is the black-box method. (Loveland et al. 2005, p.185.)

5.4 Regression testing

Not so much of a method, but more like a testing phase. The one form of testing we'll never get rid of, is the regression testing. The regression testing comes into picture when you have a perfectly functioning feature in your product, but which gets torpedoed by an implementation of a new function that somehow affected the functionality of the system. The error is made either by the new function's bad implementation, or the data created by the implementation which creates an error in an already working function. The soul purpose of a regression testing is to retest old functions after new implementations, just to see if they still work properly. Sounds like a treat to test old, already familiar functions of a product again, but believe when I say this; almost nothing is more frustrating than retesting old functions over and over again. (Perry 2000, p.115.)

But it is the testers who should be blaming themselves for having regression testing at the first place. The quality of software testing determines how often a possible, new defects generating implementation is created; if a software product is so thoroughly tested that the developers and testers have already considered the future changes (excluding customer's change requests) there wouldn't be any need for regression testing. Of course once again, this is not possible. And even if it was, it is a part of a development process to include regression testing as a part of quality assurance. "The determination as to whether to conduct regression testing should be based upon the significance of the loss that could occur due to improperly tested applications." (Perry 2000, p.116.)

6 Theories vs. Reality

There are hardly actual tools for testing, but only methods mentioned to be used as tools. Opposite to this are the new and highly evolved tools for the developers; new languages and codes, new programs. When you add those and continuously increasing standards of the market, software testing becomes more and more demanding and skill requiring. (Loveland et al. 2005, p.5.)

For a layman, software testing might sound like a walk in the park; you just play with a program and see if it works. I confess I had those thoughts too at first. But after working almost for a year as a professional software tester, I can say there is a lot more to it than meets the eye. Some companies spend from 20% to 50% of the software development budget on testing, so it can't be just anyone's job. Because it is not only doing tests for some program, it is also communicating with the customer, mastering different programs used on testing, reporting, and it requires a certain "eye" to discover what's right and wrong. Most of all, testing requires nerves and stamina, because difficulties never end. (Craig & Jaskiel 2002, p.9.)

The same question, "Why is software testing so difficult?" was asked from a software tester called Clare Matthews (Craig & Jaskiel 2002, p.9.). The answer she gave was like from my everyday life:

> Sure! Not only can I give you a fresh perspective, but by being in the trenches every day, I can offer a reality check quite well. I am sadly well-versed in doing whatever it takes to test, which includes working with ridiculous time frames, bad to no requirements, testers who need to be trained in testing, politics in test responsibility, providing data in as neutral a way as possible when notifying development and marketing state of the product. You name it… I think I've experienced it all. (Craig & Jaskiel 2002, p.9.)

6.1 Schedule

To keep on track of the project's schedule, it is very recommendable to have certain points where to focus on when doing a project plan. These points are called milestones. Milestones are usually dates, where the project reaches a certain goal from where it proceeds to next one. These goals can be the e.g. finishing date of writing specifications, workshop meeting date and so on. Then there are also testing milestones all together, which can function on different levels depending on the details of the test plan. On a higher level those milestones could be different reviews, implementation of the software etc. But on a lower level the milestones are concentrated on smaller goals, such as completion of different units of the software. (Craig & Jaskiel 2002, p.90.)

These milestones are usually agreed with the client, but rarely all of those milestones are met in time. Failing to meet them in time, means extra costs for both sides; to the client for software company's efforts, and to the software company for its employees' work. Although moving the milestones and rescheduling could lead into a conflict between a software company and the client, usually neither of them wants to end up fighting. Instead of that, they agree new milestones, but for the software company it can be a really hard place, since usually at that point the client has an upper hand when negotiating, and schedules are always next to impossible. This leads to rescheduling the rescheduled milestones and to even more unrealistic time frames.

The schedule also consists of time spent on testing the product itself. As the testing process has numerous different tests, it is extremely important to have some sort of estimation of how long it'll take to have those tests completed for agreed milestones. Normal testing, such as regression testing is easier to predict, since the earlier experience of the use of those modules has left some kind of documentation behind, which most certainly includes time spent on tests. The time required for life cycle tests can be harder to estimate; the first estimation would always be quite optimistic assuming all test cases pass without any show-stopping defects. Of course this is not realistic. So some of the test cases will have various defects that require time in order to get fixed, and that will definitely affect the schedule. For these types of tests you also have to consider the planning of the test and analyzing the outcomes of the test which are both added to the time used on the test. (Hutcheson 2003, p.110-111.)

Usually there is no time for accurate estimations when we're selling software, and try to stay no.1 in the business. Estimations are important, but can be discarded as they will never be accurate enough. Almost six out of ten defects have a change of course during their lives which make the estimations inaccurate. Be also aware that humans are the ones who make mistakes, not machines. These mistakes can be from simple user error whilst using the product to coding error in the development team. Either way, mistakes are unpredictable and cannot be taken into calculations when making estimations about time consumption.

6.2 Resources

From one point of view, resources include the costs of the software product and the reasons for them. When it comes to costs, defects can have a big part in increasing them. The reason to the increase of the costs related to the moment when a defect is discovered and fixed is very simple, and a bit frightening. Partially it is based on the amount of personnel you have to involve to get the defect fixed; if you find a defect at requirement or design phase, there is not even a bit of code written, and you can fix it just by writing it correctly. But when the project proceeds to the system test phase, the cost of a defect will be ten times higher compared to the earlier amount, and it'll be hundred times higher after the system has gone into production (see Figure 5). These kinds of things happen and are considered in the budget, or else the management team would have the hardest time of their lives after founding a defect at late hour. (Perry 2000, p.63.)

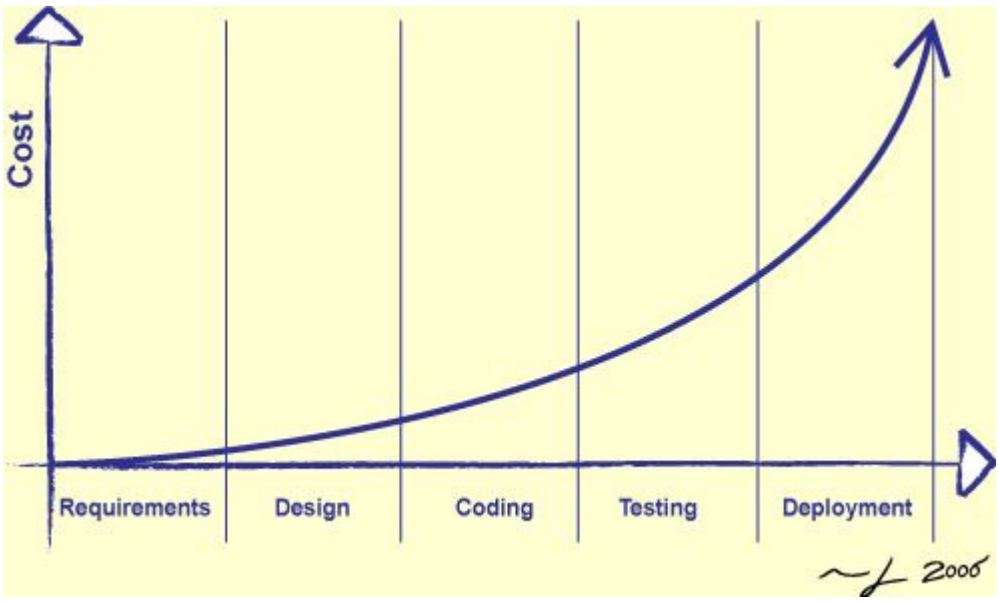

Figure 5. A crude model of relative cost vs. the project phase (Software Carpentry 2006.)

Resources consider also the employees that are involved in software testing. In a medium sized company there're simply not enough personnel to test programs as thoroughly as the client would want to. As the projects progress, new projects emerge. And when the number of testers is limited, the same testers have to be assigned to numerous of projects that all consume time. After the deployment when maintaining begins, defects start flying from every direction, mostly from the customer's. The next section will talk about that more clearly.

6.3 The client

The usual problem about working closely with the client is that they tend to see only their own perspectives of things and their own preferences above anything else. When specifications are written, they are consulted with the client and agreed mutually. Often after writing the specifications, they tend to be forgotten from both parties, and that's when the conflicts begin. If the client has its own testers for the product, the defects that they discover are sent to the software company for investigation and possible fixing. Although it sounds like a very wise decision considering that the customer ought to know the system better than anyone, it still amazes me how ignorant they are about the system and its functions. This causes the software company's e-mail getting filled with error reports from the customer (unless the software company and the client have the same bug DB), from which usually half of them are already in the bug DB which makes them duplicate bugs, or then they are just customers' opinions of how things should be, and that makes them change requests that are chargeable, and not even close to a bug. All those things, including debates and extra information requests from the customer add significantly to the workload the software company employees already have, not to mention all the other projects that are up and running and in need of maintaining.

7 Moulding a process for real life testing

The reality differs a lot from theoretical scenes, and not always are procedures done like in a book. Many development processes and testing methods have shown that if executed correctly and flawlessly, they can be powerful tools. Yet still there are some things that can be more accurately defined – some parts of a process that can be left out as obsolete steps, or then they can be replaced by a different kind of a step which is more accurate to a real life situation. Testing methods also have possibilities for fine tuning, though even after those tunings things can turn upside down which is caused by the only factor that makes mistakes in the first place: a human. This means that each company with testers require their own customised development process via testing methods with strict rules before even starting any projects. The number of these development processes and testing methods is quite big, but after studying these pre-mentioned ways, there is a change to modify them and mould them to serve each company's needs the best.

7.1 Comparison between a theory-based process and "the real" process

One of the most common and well known development processes, the waterfall process is probably the most effective one. Another reason why I brought this process up was that from the processes I've studied, this is the closest one of them all I've worked with and thus easier to approach. It proceeds very strictly through steps which are always required before proceeding to the new step. Beginning from customer requirements (see Figure 2, page 9) and ending at maintenance testing it sounds almost perfect in theory. If we break down the waterfall model's testing phase we can see a "miniature version" of a waterfall model which rotates the testing cycle (see Figure 6). (Loveland et al. 2005, p.43.)

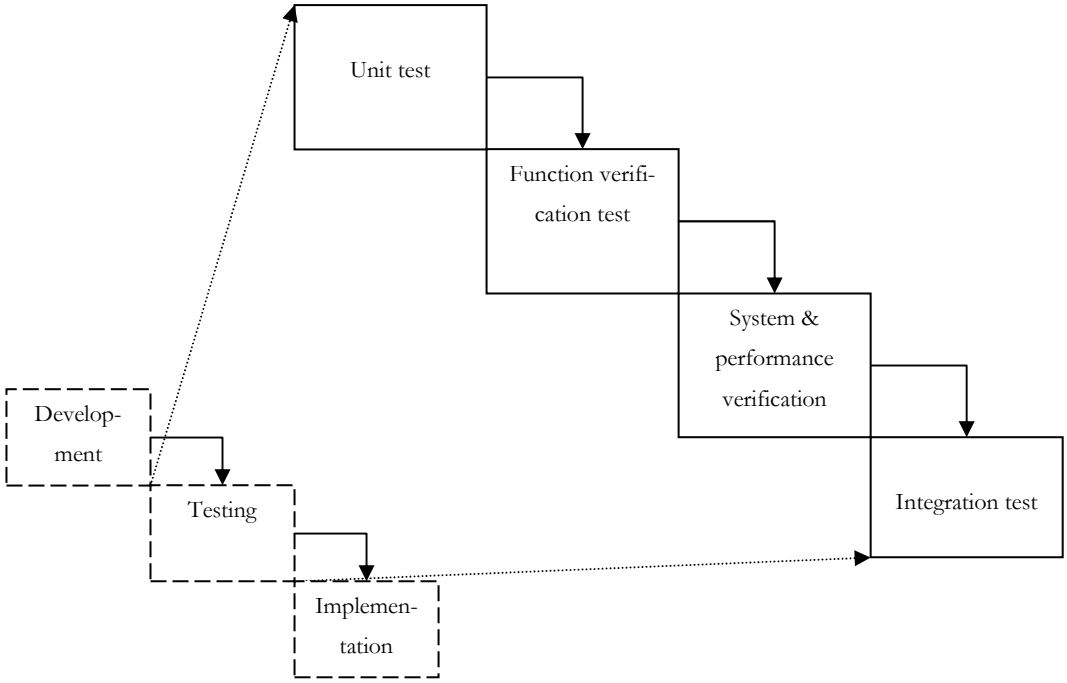

Figure 6. Breaking down testing phase of the waterfall model (Loveland et al. 2005, p.44.)

The testing phase is divided into four different parts, unit testing, function verification testing, system and performance verification, and integration testing. From these four, one is not even performed at our company, and that is system and performance verification. It might be that some developers who are executing the three other test phases as individuals but the more I test software, the more I'm convinced that all of these are, metaphorically speaking, mutilated by a blender, mixed in a big bowl and randomly picked out for action.

7.2 Removing unnecessary phases

Usually some sort of testing process is clearly defined in a software company which is strictly followed without questioning the phases that are in them. The software development and test teams can get contemptuous and pass some of the phases in the way, still believing that all the serious defects won't appear during those steps, or that they've already been discovered. Each phase has its own method of testing which reveal the weaknesses of the software product, and if a phase gets skipped, those weaknesses will stay in the system and pop out at the worst moment since none of the later phases' testing methods have the logic to discover them. These kinds of defects can be very hazardous especially if the defect that was not discovered is from a very early phase, fixing it may affect several other areas of the software and cause multiple errors in the system. (Loveland et al. 2005, p.59.)

Probably the most common discarded test phase would be probably the system verification test phase which is very similar to the integration test phase. The discarding tends to happen if the software development and test teams have a solid, stable test environment which can handle all sorts of tests that are able to reveal even the most complex defects to be found. The main difference between an integration test phase and a system verification test phase is the number and type of different tests that are run at the same time. While integration test concentrates on few different modules' interaction and collaboration, system verification test runs multiple tests with parallel functions which can have varying workloads. This will able a continuous testing progress as one test can fail due to an error or a defect, but as that defect gets reported and the fix is made, at the same time those other tests are able to continue if the defect didn't affect their area of system. The effectiveness and accuracy contributed by a system verification test shouldn't be underestimated in any circumstances. (Loveland et al. 2005, p.59.)

Skipping the system verification test might have totally opposite affect to the testing schedule the development and test teams were trying to shorten. This kind of action would have disastrous consequences during the final integration test where the collaboration between the constructed software is tested with the customer's own system's parts. The defects that were not discovered can have a show stopping affect to the whole test as most of the parts of the system are reliable on each other; one serious defect that blocks any future actions on one area might block any other actions made on any other test case. Even if the fix to this kind of a problem would be available just in a matter of hours – which usually isn't the case – the test cases would have a chance of progress for a few steps and then stumbling on a new "showstopper" which might take more than just few hours. This might give us a conclusion of not removing the system verification test from the test phases, since it clearly has its own purpose in the process. (Loveland et al. 2005, p.59.)

7.2.1 Providing theoretical aid to the phases

From what we learned from the evolutionary model, waterfall process model is lacking one very useful thing – a natural feedback loop. Feedback from the prior cycle is evaluated during the execution of the next cycle, and can be critical to the ultimate success of the project. The short cycles include all aspects of design, code, and initial testing of a new version of software. The concept of having miniature "development cycles" within the overall development process is comparable to the waterwheel model discussed earlier because the waterwheel is iteratively delivering functions in code drops, each of which goes through design, development, and test prior to delivery beta customers, along with additional, post-delivery testing. (Loveland et al. 2005, p.55.)

This gives a picture of a wild hybrid of processes; a chunk of waterfall model, a bit of waterwheel model, and a hint of evolutionary model, all working together as one massive tool. The waterfall model would be the "skeleton" of the actual development process, and create a preliminary model and phases for it. The contribution made by the waterwheel model is those tiny portions of code delivered by the development team which are tested by test team, except this is the section where the evolutionary model kicks in; those tiny portions of code would be tested in both ends, by the professional testers and the customers, and the feedback of those tests would be delivered back to the development team. This procedure would enhance the quality and the preciseness of the testing feedback due to different testing methods used at the same time by different level testers, and it would also decrease the birth of false defects, the amount of wrongly executed bug fixes, and the time spent on endless regression testing.

The problem is that when a solution like this pops up, I begin to wonder, how in the world I could get it in use at our corporation. The time required for adapting into a new kind of software development process model would be enormous, not to mention the shift from the old one would have to be taken gradually, and not without a proper build and installation process which tend to take some time. No software development model will ever be successful without an efficient build and installation process. The technical professionals performing this are critical to the project's success. Without the build and install team, there would be nothing to test. (Loveland et al. 2005, p.48-49.)

8 Conclusions

A theoretical solution was found during this research; to merge waterfall, waterwheel and evolutionary model into one big hybrid process. This would probably correct the issues of quality management which is a noticeable part of testing during system and integration test phases. Though it would sound like an easy solution, it most certainly is not. To build a specific model requires lots of time to build, and a lot more research, plus, to get it work it would require professional help to help personnel to adapt to the new model and to a company that's on a roll, is not an easy or cheap task.

It was enlightening to found myself from a book, described with so accurate details but it was also annoying when I realized that the goal of one of my objectives was completely the opposite I was aiming at. When trying to develop/enhance software development processes and models, the last thing to do is to cut out phases which are already proven to be essential. Instead of removing, why not try replacing or adding phases? Although it sounds like workload increases and time frames getting narrower, but on the other hand it could offer us more accurate test results in earlier steps which would pay back later as a satisfied customer and a fewer hours used in total.

Before even considering a new development process or a testing method for a company, I'd suggest more workshops and discussion between testing and development teams, as well as between the software company and the client. By creating strict rules that bind everyone should – and will – make the working environment pleasing and the flow of work itself fluent without changing the process. This would still require a great amount of time to master the process already in use, but it would still be the fastest and cheapest choice of them all. Changing a process in a middle of a run can be hazardous, especially if the new process model hasn't been properly tested and proven to be more effective then the earlier one; you wouldn't jump out of a moving car without slowing down just to prove walking is healthier, now would you?

9 Future research

Now that a new idea out there and ready to be implemented, a new challenge arrives to my path; the company's approval of the solution, final internal research of the process, and finally getting the model in use. Not a day goes by when someone wouldn't complain about the current process, about its lack of this and that, but when new solution is presented to a group of professionals who have already adapted the current process so well, they have hard time approving the new process and call it as their own style. Since this new model would mostly affect testers' everyday lives, it wouldn't be too popular from the other roles' point of view. This doesn't surprise me at all; humans are after all very self-centered and eager to find the best way for themselves to manage their business. If this new model wants to get used, it has to evolve to serve other company roles as much as it would serve testers, and only then it would be easier for people to adapt to it. (Craig & Jaskiel 2002, p.388.)

If this solution would ever see daylight it would have to be run through some test environments, e.g. a small project which has just started in a software company. The feedback of the new process in progress would determine if it should or should not be taken into use to the whole company. As not being the easiest one of missions, it would most certainly require months and months of preparations and planning. The execution itself would take few months more just to see the stable results of the process. So planning and executing a plan of this scale requires lots of resources and time – the research of the plan itself requires resources and time. I wouldn't be amazed if some companies would give this plan a go, but in my opinion, if anyone would ever consider even trying this, I'd suggest more research and not to do it alone; the more professionals working on the subject the better.

Bibliography

Loveland, S., Miller, G., Prewitt, R., Shannon, M. 2005. Software testing techniques: Finding the Defects that Matter. Charles River Media, Inc. Hingham, Massachusetts.

Beizer, B. 1995. Black-box testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc. Toronto

Hutcheson, Marnie L. 2003. Software Testing Fundamentals: Methods and Metrics. Wiley Publishing Inc. Indianapolis, Indiana.

Craig, Rick D., Jaskiel, Stefan P. 2002. Systematic Software Testing. Artech House Publishers. London.

Perry, William E. 2000. Effective Methods for Software Testing. 2. ed. John Wiley & Sons, Inc. Toronto.

Myers, Glenford J. 2004. The Art of Software Testing. 2. ed. John Wiley & Sons, Inc. Hoboken, New Jersey.

Wikipedia, the free encyclopedia. URL:
http://en.wikipedia.org/wiki/Image:V-model.JPG. Quoted: 27.10.2008.

Wikipedia, the free encyclopedia. URL:
http://en.wikipedia.org/wiki/Image:Spiral_model_(Boehm,_1988).png. Quoted: 27.10.2008.

Wikipedia, the free encyclopedia. URL:
http://en.wikipedia.org/wiki/Software_testing#Grey_Box_Testing. Quoted: 29.10.2008.

Software Carpentry. URL:
http://swc.scipy.org/lec/img/dev01/boehm_curve.png. Quoted: 1.11.2008.

FINAL REPORT

CONTENT

1. Finalizing the project

The thesis project was launched on 8.9.2008 with a goal to discover a new/improved method for software testing life cycle on integration and system testing. The first steering group meeting was held on 17.10.2008 and minor changes were made on the project plan. This second steering group meeting was supposed to be held on 14.11.2008, but was rescheduled to 19.11.2008 due to Raine Kauppinen's illness. The closing meeting of the project was held on 24.11.2008 at Haaga-Helia, room 6005 where the final report and the thesis were turned in.

2. Plan & schedule changes

The project plan was not modified after the fourth (corrected) version which was modified after the first steering group meeting that was held on 17.10.2008. The time of the closing meeting was rescheduled by Raine Kauppinen for the same day, Monday 24.11.2008, from 8.00 to 12.00. The maturity test date was decided at the second steering group meeting that was held on Wednesday 19.11.2008. The test day is Wednesday 26.11.2008.

3. Accomplishments

All parts of the thesis work are complete and ready to be turned in. The final parts, "fine tuning" of the chapters and correction of references were made after the second steering group meeting.

4. Use of resources

It was decided by Teemu Vänskä that less internet resources would be used for the thesis in order to secure the authenticity of the material used in this project. Only a few internet resources were used for diagrams or possible extra information. A large amount of useful books were/are found at the Haaga-Helia library which will be the main resource for the project.

5. Hours spent on the project

WEEK	37	38	39	40	41	42	43	44	45	46	47	48	49	TOTAL
Hours according to project plan	30	35	15	20	20	50	40	30	25	45	35	40	15	400
Actual hours spent on the project	18	32	10	17	15	41	36	43	42	55	48	38	12	407

From the table it can be seen that there are large differences between what was originally planned and how it was executed. For this effect I can name two reasons:

1. I was working at the same time I was doing the thesis. I planned that I could get some days off from my job for the project, and so I did. Nevertheless, after few weeks I was told that my presence was required every day at the office and that mixed up my pace.
2. I woke up too late for the project, and was too narrow-sited. Only after when I had started the actual work I realised how big it was going to be. This caused quite a hassle at the end.