

---

MEE05:10



---

# Automatiserad Unit Testning

**Daniel Sandberg**

Examensarbete  
Magisterexamen i elektroteknik - inriktning telekommunikation

Blekinge Tekniska Högskola

Mars 2005

---

Blekinge Institute of Technology  
Department of Telecommunication Systems  
University advisor: Gunnar Råhlén  
External advisors: Christopher Carlander, UIQ Technology AB, Ronneby  
David Rosén, UIQ Technology AB, Ronneby

---

---

This thesis is submitted to the Department of Telecommunication Systems at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

**Contact Information:**

Author:

Daniel Sandberg

E-mail: [daniel.sandberg@gmail.com](mailto:daniel.sandberg@gmail.com)

Phone: +46 70 210 26 95

External advisors:

Christopher Carlander

UIQ Technology AB

Soft Center 8

SE - 372 25 Ronneby

Phone: +46 457 46 47 38

David Rosén

UIQ Technology AB

Soft Center 8

SE - 372 25 Ronneby

Phone: +46 457 46 48 32

University advisor:

Gunnar Råhlén

Department of Telecommunication Systems

Phone: +46 455 38 56 68

Blekinge Institute of Technology

Department of

Telecommunication Systems

Soft Center

SE – 372 25 Ronneby

Internet : [www.bth.se/tek](http://www.bth.se/tek)

---

# Abstrakt

(Svenska)

Dagens utveckling av mjukvara går snabbare och snabbare samtidigt som mjukvaran blir allt mer komplex. Att under dessa omständigheter kunna bibehålla en lika om inte högre kodkvalité är en utmaning. På UIQ bestämde de sig för att undersöka om ett automatiserat *unit* test kunde hjälpa dem.

Metoderna för att komma fram till resultaten i denna rapport har varit intervjuer, en enkät, intern information och litteratur så som forskningsartiklar m.m.

Denna rapport är en utredning av automatiserade test, vad man bör tänka på, vilka fördelarna är, vilka testfall som är möjliga att automatisera med mera. Rapporten kommer även att presentera grunderna i testning av mjukvara, en introduktion till eXtreme Programming och Test-Driven Development samt hur testningen utförs på UIQ idag.

Då jag kom fram till att ett automatiserat *unit* test skulle passa sig bra på UIQ kommer jag i slutet av denna rapport presentera ett implementerings förslag. Man bör även införa en test driven utvecklings metodik för att säkerhetsställa att det kommer att utvecklas automatiserade testfall.

## **Nyckelord:**

Automatiserad *unit* testning, eXtreme Programming, Test-Driven Development

---

# Abstract

(English)

The pace at which the software industry has to deliver today to keep up to market is being faster and faster, even though the complexity of the product increases. To improve or at least keep the same code quality as before is a never-ending challenge. At UIQ Technology they decided to investigate if an automated unit test process could help them ease these issues.

The methods used to obtain the results presented in this report have been interviews, a survey, internal information and litterateur like research articles.

This report comprises an investigation about test automation, the pros and cons together with examples of test cases that may easily be automated. The report will also present the basic principles of software testing, an introduction to Test-Driven Development and eXtreme Programming and how testing is performed at UIQ today.

The general conclusion of this thesis is that an automated test process would help UIQ to increase their quality in their products. An example implementation suggestion and other issues worth considering before introducing an automated test process will therefore be presented. One of the suggestions was to introduce Test-Driven Development as a new developing method.

**Keywords:**

Automated *unit* testing, eXtreme Programming, Test-Driven Development

---

## Förord

Först skulle jag vilja tacka UIQ Technology AB för att de gav mig chansen att få utföra mitt examensarbete hos dem. Sen så vill jag även utföra ett extra stort tack till följande personer för all hjälp jag fått med examensrapporten:

*Christopher Carlander*

Handledare på UIQ som gav mig chansen att få göra mitt examensarbete på UIQ.

*David Rosén*

Handledare på UIQ som var ett bra bollplan att diskutera idéer och funderingar med. Han kom även med bra kommentarer om rapporten.

*Gunnar Råhlen*

Handledare på BTH som kommit med många värdefulla tips.

*Anton Larsson*

F.d. anställd på UIQ som introducerade mig i Symbian programmering och TEFUnit.

*Kennet Henningsson*

Doktorand på BTH/UIQ som var till stor hjälp i kostnad/vinst kapitlet.

*Alla enkättagare*

För att de tog sig tid att fylla i min enkät.

*Alla intervjupersoner*

För att de tog sig tid att svara på mina frågor.

*Övriga*

Alla andra som på något vis har hjälpt till med detta examensarbete.

---

# Innehållsförteckning

<b>ABSTRAKT (SVENSKA)</b> .....	<b>II</b>
<b>ABSTRACT (ENGLISH)</b> .....	<b>III</b>
<b>FÖRORD</b> .....	<b>IV</b>
<b>INNEHÅLLSFÖRTECKNING</b> .....	<b>V</b>
<b>1 INTRODUCTION</b> .....	<b>1</b>
1.1 INLEDNING .....	1
1.2 AUTOMATISERAD TESTNING.....	1
1.3 UIQ TECHNOLOGY AB .....	1
1.4 ENGELSKA UTTRYCK I RAPPORTEN.....	2
1.5 SYFTE OCH OMFATTNING .....	2
1.5.1 Syfte .....	2
1.5.2 Hypoteser.....	2
1.5.3 Avgränsningar .....	3
1.6 DISPOSITION AV RAPPORTEN .....	3
<b>2 METOD</b> .....	<b>5</b>
2.1 INLEDNING .....	5
2.2 INTERVJUER .....	5
2.3 ENKÄT.....	5
<b>3 TESTNING AV MJUKVARA</b> .....	<b>7</b>
3.1 INLEDNING .....	7
3.2 TEST FASER .....	7
3.3 TESTFALL.....	8
3.4 POSITIV OCH NEGATIV TESTNING .....	9
3.5 BLACK-BOX OCH WHITE-BOX TESTNING.....	9
3.6 REGRESSIONSTESTNING .....	9
3.7 TESTDESIGN DOKUMENTATION .....	9
3.8 AUTOMATISERAD TESTNING.....	9
3.8.1 Fördelar.....	10
3.8.2 Realistiska förväntningar.....	10
3.8.3 Vilka test skall man automatisera.....	11
3.8.4 När man inte skall automatisera.....	12
3.8.5 Underhåll av automatiserade tester.....	13
3.8.6 Record/playback metoden.....	14
3.8.7 Forskning.....	14
<b>4 TESTNING PÅ UIQ</b> .....	<b>15</b>
4.1 INLEDNING .....	15
4.2 UNIT TESTNING .....	15
4.3 SYSTEMTESTNING .....	15
4.4 TESTPROGRAMVARA .....	15
4.4.1 TEFUnit.....	15
4.4.2 Blueberry .....	16
4.4.3 CodeTest .....	16
4.4.4 Lint scan .....	16
4.4.5 Leave scan .....	16
4.4.6 Emulator .....	16
4.4.7 Doxygen.....	16

---

<b>5</b>	<b>PRODUKTUTVECKLINGSPROCESSEN PÅ UIQ</b>	<b>17</b>
5.1	D-MODELLEN	17
5.2	MJUKVARUUTVECKLINGSPROCESSEN	17
5.3	UTVECKLINGSMETODIKEN	18
5.4	RELEASE/INTEGRATION	18
<b>6</b>	<b>TDD, AGILE OCH XP</b>	<b>19</b>
6.1	INLEDNING	19
6.2	TEST-DRIVEN DEVELOPMENT	19
6.2.1	<i>Forskning</i>	20
6.2.2	<i>Nackdelar</i>	20
6.2.3	<i>Fördelar</i>	21
6.3	EXTREME PROGRAMMING	21
6.3.1	<i>Forskning</i>	22
6.3.2	<i>Nackdelar</i>	22
6.3.3	<i>Fördelare</i>	22
6.4	AGILE MODEL	23
<b>7</b>	<b>UNDERSÖKNINGAR PÅ UIQ</b>	<b>24</b>
7.1	INLEDNING	24
7.2	INTERVJU FAS ETT	24
7.2.1	<i>Minnesläckor</i>	24
7.2.2	<i>Heap failure</i>	24
7.3	INTERVJU FAS TVÅ	24
7.4	INTERVJU FAS TRE	25
7.5	ENKÄT	25
7.5.1	<i>Enkät resultat och diskussion</i>	26
7.5.2	<i>Samband</i>	28
7.5.3	<i>Kommentarer från enkäten</i>	30
7.5.4	<i>Felaktigheter i enkäten</i>	31
7.6	DOKUMENT ANALYS	31
<b>8</b>	<b>ANALYS AV TESTFALL</b>	<b>32</b>
8.1	INLEDNING	32
8.2	MINNESLÄCKOR	32
8.3	HEAP FAILURE	32
8.4	FUNKTIONALITETSTESTNING	33
8.5	BOUNDARY	33
8.6	OUT OF DISK	33
8.7	FILE ACCESS	33
8.8	PRESTANDA	34
8.9	SERVER CONNECTION HANDLING	34
8.10	RESPONSE TIME	34
8.11	LOCALIZATION	34
8.12	VISUAL TESTING OF GRAPHICAL COMPONENTS	34
8.13	REDUNDANT OR UNUSED CODE TESTING	35
8.14	IN SOURCE DOCUMENTATION OF SOURCE CODE	35
8.15	COMPILE TESTING OF SOURCE CODE	35
<b>9</b>	<b>KOSTNAD/VINST BERÄKNINGAR PÅ ETT AUT</b>	<b>36</b>
9.1	KOSTNADER	36
9.2	VINSTER	37
9.3	REGRESSIONSTESTNING	38
9.4	RÄKNEEXEMPEL	38
<b>10</b>	<b>IMPLEMENTERINGS FÖRSLAG</b>	<b>39</b>

---

---

10.1	INLEDNING .....	39
10.2	TESTETS STRUKTUR.....	39
10.3	NÄR SKA TESTET KÖRAS?.....	39
10.4	VAD SKA TESTET INNEHÅLLA? .....	39
10.5	HUR SKA RESULTATET FRÅN TESTET LAGRAS?.....	40
10.6	PRESENTATION AV RESULTATET .....	40
10.7	UTBILDNING OCH INFORMATION .....	41
10.8	FÖRVÄNTNINGAR .....	41
<b>11</b>	<b>DISKUSSION .....</b>	<b>42</b>
11.1	VALIDERING AV HYPOTESERNA.....	46
<b>12</b>	<b>SAMMANFATTNING .....</b>	<b>49</b>
<b>13</b>	<b>FRAMTIDA EXAMENSARBETEN .....</b>	<b>51</b>
<b>14</b>	<b>DEFINITIONER OCH FÖRKORTNINGAR.....</b>	<b>52</b>
14.1	DEFINITIONER .....	52
14.2	FÖRKORTNINGAR .....	53
<b>15</b>	<b>REFERENSER.....</b>	<b>54</b>
<b>INNEHÅLLSFÖRTECKNING – APPENDIX .....</b>		<b>56</b>
<b>APPENDIX I – ENKÄTEN .....</b>		<b>57</b>
<b>APPENDIX II – ENKÄT RESULTAT .....</b>		<b>62</b>
<b>APPENDIX III – FIGURER .....</b>		<b>84</b>
<b>APPENDIX IV – INTERVJUFRÅGOR FAS ETT .....</b>		<b>86</b>
<b>APPENDIX V – INTERVJUFRÅGOR FAS TVÅ .....</b>		<b>87</b>
<b>APPENDIX VI – INTERVJUFRÅGOR FAS TRE.....</b>		<b>88</b>
<b>APPENDIX VII – RÄKNEEXEMPEL .....</b>		<b>90</b>

---

## Kapitel

# 1 INTRODUKTION

---

### 1.1 Inledning

Detta examensarbete utfördes på UIQ Technology AB i Ronneby av magisterstudenten Daniel Sandberg höstterminen 2004. Innehållet i rapporten kommer att diskutera frågor rörande automatiserad *unit* testning.

Eftersom UIQ är ett snabbt växande företag så behöver också produktutvecklings- och testprocesserna ses över regelbundet. Det är där detta examensarbete kommer in i bilden. Företaget ville höja kodkvalitén ytterligare och vill därför undersöka möjligheten att införa ett automatiskt *unit* test samt veta hur det skulle påverka utvecklingstiden och kodkvalitén.

*Unit* testningen skall utföras av utvecklaren själv innan han gör en *release* av sin kod. Men ett av de stora problemen i dagens mjukvaruutveckling är den ökade tidspressen vilket gör att testningen som vanligtvis ligger sist i processen blir lidande. Om detta går att undvika med ett automatiserat *unit* test kommer denna rapport undersöka.

### 1.2 Automatiserad testning

Det första en utvecklare tänker på när man hör de tre magiska orden automatisk *unit* testning är att han äntligen slipper sitta med den tråkiga testningen. Medan chefen och ekonomiansvarige börjar räkna på hur mycket pengar de kan spara efter de sagt upp halva testavdelningen. Tyvärr får båda tänka om då oftast den automatiska testningen har lika lång utvecklingstid men förhoppningvis om den blir lyckad så förbättrar den kodkvalitén. Vilket kan vara minst lika mycket värt som att man sparar några kronor i minskade personalkostnader.

### 1.3 UIQ Technology AB

UIQ Technology är ett ungt snabbt växande företag inom IT-sektorn. Deras huvudinkomst kommer från mobiltelefonoperativsystemet UIQ men de har även konsulter som jobbar mot kunderna. För tillfället så håller företaget på att utveckla UIQ 3.0 som till skillnad från föregångaren UIQ 2.1 kan hantera både inmatning från penna och enhandsfattning.

Symbian som är ett samägt bolag mellan Ericsson, Sony Ericsson, Panasonic, Samsung, Siemens och Nokia startade år 1999 dotterbolaget UIQ Technology AB som innan var en avdelning på Ericsson. Deras uppgift var att tillhandahålla en plattform för *smartphones* som skulle vara oberoende från någon av de stora mobiltillverkarna. Exempel på mobiltelefoner som använder sig av UIQ plattformen är Sony Ericsson P900 och

---

Motorola A1000. När denna rapport skrevs fanns det ytterligare sju modeller på marknaden som använder sig av UIQ.

UIQ har idag cirka 140 anställda varav cirka 70 personer är utvecklare. Företaget är uppdelat i sju avdelningar bestående av *software development*, *business development*, *marketing*, *IS & facilities*, *product management*, *human resources & administration and finance*. *Software development* är den avdelning som utvecklarna ligger under och den är uppdelad i ytterligare sju större delar bestående av *application*, *communications*, *interaction design*, *plattform*, *system management & engineering*, *product management* och *product verification*.

## 1.4 Engelska uttryck i rapporten

Eftersom programmering och testning tillhör den snabbt växande tekniksektorn så används det mycket engelska uttryck. Dessa uttryck kan vara svåra att översätta, därför kommer de jag valt att inte översätta skrivas i kursivstil och i slutet av rapporten finns sedan en definition av uttrycket.

Vi diskuterade mycket och väl om vi skulle använda namnen *unit testing*, *unit testning* eller *enhetstestning*. Men vi kom fram till att det mest passande namnet var *unit testning* då det är vedertaget att säga *unit testning* när man menar *enhetstestning*. Att sedan använda det svenska ordet för testning anser vi också är bäst då det är lättare att böja grammatiskt.

## 1.5 Syfte och omfattning

### 1.5.1 Syfte

Utreda hur man kan öka kodkvaliteten med hjälp av ett automatiserat *unit test*.

För att uppnå målet ovan kommer rapporten innehålla följande:

- Undersökning av den nuvarande utvecklingsprocessen.
- Förslag på hur ett automatiskt *unit test* skulle kunna implementeras på UIQ.
- Enkät och intervjuer med personal på olika befattningar.
- Identifikation av vilka testfall som går att använda i det automatiserade *unit testet*.
- Utredning av för- och nackdelar med automatiserade test.
- Utvärdering av *Test-Driven Development* och *eXtreme Programming*.

### 1.5.2 Hypoteser

Dagens produktutvecklingsprocess släpper i genom onödiga defekter p.g.a.:

... dagens *unit test* process är otydlig och kan förbättras.

... saknad av ett automatiserade test.

... utvecklaren inte använder alla tillgängliga testverktyg.

---

Möjligheten finns att minska antalet defekter:  
... genom att införa en ny utvecklingsmetodik.  
... genom att upptäcka defekterna tidigare.  
... med ett automatiserat test samtidigt som kostnaderna är detsamma eller lägre.

### 1.5.3 Avgränsningar

Att undersöka hela produktutvecklingsprocessen skulle ta alldeles för lång tid samt till viss del vara onödig då alla delar inte påverkar antalet defekter lika mycket. Den delen jag kommer att inrikta mig på kommer att vara från utvecklaren fått sin uppgift till att han lämnar den vidare till systemtestning. Jag har även valt att enbart undersöka utvecklingsmetodikerna TDD, XP och Agile Modelling.

## 1.6 Disposition av rapporten

Denna rapport vänder sig till beslutsfattare på UIQ och andra som är intresserade av automatiserad *unit* testning. Nedan följer en kort presentation av varje kapitel.

Kapitel 2, Metod

Detta kapitel beskriver de metoder jag använde mig av för att få fram informationen.

Kapitel 3, Testning av mjukvara

Detta kapitel innehåller allmänt om testning med fördjupning mot automatiserad testning.

Kapitel 4, Testning på UIQ

Kapitel fyra presenterar kort hur testningen utförs på UIQ och en kort presentation av de vanligaste och mest använda programvarorna inom testning.

Kapitel 5, Produktutvecklingsprocessen på UIQ

Beskriver de olika processerna som används på UIQ vid produktutvecklingen.

Kapitel 6, TDD och XP

Presentation av de två utvecklingsmetodikerna Test-Driven Development och eXtreme Programming. Även kort om Agile Modelling

Kapitel 7, Undersökningar på UIQ

I detta kapitel presenteras resultaten från alla intervjuer samt enkäten.

Kapitel 8, Analys av testfall

Presentation av olika testfall på UIQ som bör ingå i *unit* testet.

Kapitel 9, Kostnad/vinst beräkningar på ett automatiskt *unit* test

Detta kapitel utreder om det är lönsamt att införa ett automatiserat test.

Kapitel 10, Implementerings förslag

Här presenteras ett förslag på hur ett automatiskt test skulle kunna se ut på UIQ

---

Kapitel 11, Diskussion

Här diskuterar jag resultat och annat från rapporten

Kapitel 12, Sammanfattning

Här sammanfattar jag de viktigaste resultaten från rapporten.

Kapitel 13, Framtida examensarbeten

Förslag på framtida examensarbeten.

---

## Kapitel

# 2 Metod

---

### 2.1 Inledning

Syftet med detta kapitel är att presentera de olika metoderna som jag använde för att inhämta information till rapporten. Informationen om företagets nuvarande process kommer att inhämtas via intervjuer, enkät och interna dokument. Övrig information kommer att inhämtas från forskningsartiklar, gamla examensarbeten, Internet och annan litteratur.

### 2.2 Intervjuer

I den första intervjufasen utfördes ett mindre antal intervjuer med några utvecklare för att få mera information om hur de testat efter minnesläckor och *heap failure*. Dessa utfördes enligt *standard open-ended interview* tekniken vilket menas att man har gjort upp ett frågeformulär innan med frågor som man sedan diskuterar en efter en.

I intervju fas två när testprocessen skulle undersökas så använde jag mig av en teknik kallad *general interview guide approach* där man har några punkter som skall diskuteras och den är mycket friare. Urvalet av intervjupersoner gjordes slumpmässigt, men jag försökte att välja en utvecklare från varje sektion.

I intervju fas tre så använde jag mig av *general interview guide approach* tekniken igen för att diskutera igenom automatiserade *unit* testet i allmänhet med en person på en chefs position.

### 2.3 Enkät

Enkäten ska gå ut till alla utvecklare på UIQ under deras sektionmöten. Frågorna är uppdelade i fyra delar, den första delen innehåller blandade frågor, den andra delen frågor om *Test-Driven Development*, den tredje delen frågor om par programmering och den sista delen är bakgrunds frågor. Jag lade in alla resultat i ett datablad i SPSS<sup>1</sup> för att kunna utföra statistiska beräkningar på dem.

I enkäten så använde jag mig av kvalitativa svarsalternativ som jag sedan räknade om till kvantitativa svar. Till exempel att svarsalternativet ”håller ej med” motsvarar siffran ett medan ”instämmer helt” motsvarade siffran fem. På detta sätt kunde jag få ett medelvärde på varje fråga som jag sedan kunde jämföra med andra frågeställningar. [1]

---

<sup>1</sup> SPSS – Statistik program som används för att sammanställa bland annat enkäter. Programmet innehåller flertalet funktioner så som att se samband, plocka ut frekvenser m.m.

---

Enkäten kommer att ha en hög grad av standardisering, då alla kommer att få exakt samma frågor. Förhoppningen är att utvecklarna skall kunna sitta i så lik miljö som möjligt samt att eventuell information som de kommer att få från mig skall vara så likvärdig som möjlig för att undvika att de påverkas att svara på något speciellt sätt. [1]

Trovärdigheten i enkäten strävar jag efter att få så hög som möjligt genom att inte använda negationer, krångliga ord eller ordvändningar. Validiteten är också viktig, ett exempel på vad validation är om man frågar efter hur ofta man läser en dagstidning så har man svarsalternativen dagligen, veckovis osv. istället för ibland, ofta osv. Detta ger en hög validitet. [1]

Gruppundersökning är att föredra framför en postundersökning (enkäten sänds med post). Nackdelen med gruppundersökningen är att medarbetaren kan se vad man svarar men då denna enkät ej innehåller några känsliga uppgifter så valde jag gruppundersöknings alternativet. [1]

Jag försöker även undvika öppna frågor dvs. frågor som inte har fasta svarsalternativ då de är svårare att sammanställa. Även att en del människor kan undvika att svara då de känner sig dåliga på att formulera sig skriftligt samt är dåliga på att stava. Tre frågor i enkäten är jag tvungen att ha som öppna då de har oändligt med svarsalternativ. Frågorna som det gäller är följdfrågorna till utvecklingsmetodikerna samt den avslutande frågan där enkätdeltagaren kunde lämna synpunkter på enkäten. [1]

Jag väljer även att använda mig av ”du” och ”ditt” i hela enkäten för att vara konsekvent. Placeringen av rutorna valde jag till höger om svarsalternativet då det passar högerhänta mycket bra medan vänsterhänta får det lite svårare men då de är i minoritet så får de anpassa sig. [1]

Man bör också väga in när man tolkar resultaten från enkäten att när man går ut med den så finns det de som brukar svara så som de tror att frågeställaren vill ha svaret istället för att svara ärligt. [1]

Jag har även valt att vara lite extra positiv i inledningarna till TDD och par-programmering då denna enkät kan vara det första de får läsa om teknikerna. Att deras första intryck måste bli positivt anser jag viktigt för hur de kommer att ställa sig vid ett eventuellt införande av de ovan nämnda.

---

## Kapitel

# 3 Testning av mjukvara

---

### 3.1 Inledning

Syftet med detta kapitel är att ge läsaren en överblick av vad testning är, hur den går till och varför den görs. Jag kommer även att beskriva *unit* testning och automatiserad testning närmare.

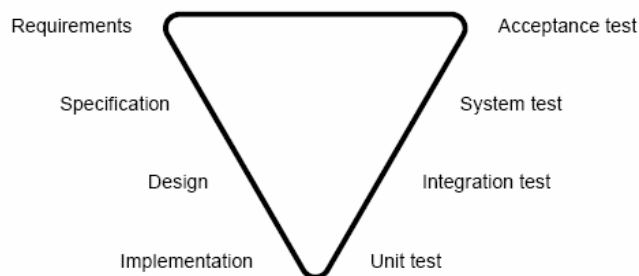
Man kan beskriva vad testning av mjukvara är med frågan, betar sig mjukvaran som specificerat? Sen kan man ytterligare dela upp det i de två termerna verifikation och validering. Verifikation är när man testar att man har gjort rätt och validering är när man undersöker att man verkligen utvecklat det som kunden efterfrågade. [2]

- Validering: Utför vi rätt uppgift?
- Verifikation: Utför vi uppgiften rätt?

Andra aktiviteter som förknippas med testning av mjukvara är *static analysis* och *dynamic analysis*. Den förstnämnda undersöker mjukvarans källkod efter problem samt gör mätningar utan att exekvera koden. *Dynamic analysis* undersöker hur systemet betar sig under exekveringen, för att sedan tillhandahålla exekveringsspår, tids profiler och testfallstäckning. [2]

### 3.2 Test faser

Det finns fyra olika faser som testning vanligtvis består av. Första fasen är *unit* testfasen, där varje komponent testas för att verifiera att den är korrekt implementerad. Därefter kommer integrationstestningen där man har ett ej färdigt system som testas för att se att komponenterna fungerar ihop. Sen kommer systemtestningen där den nästan klara mjukvaran integreras och man testar att systemet uppfyller de ställda kraven. Den slutgiltiga testningen där kunden är med och godkänner slutprodukten kallas *acceptance testing*. [2]



Figur 1: V - modellen

---

Figuren på sidan innan kallas V-modellen och beskriver relationerna mellan testnivåerna och mjukvaruutvecklingens livscykel. [3]

En viktig fråga för detta examensarbete är den om vad definitionen för en *unit* är. På UIQ finns det idag flertalet olika bud på vad som är *unit* testning och vad som inte är det. Den exakta definitionen för vad en *unit* är beror på vilken implementerings teknologi som används vid utvecklingen av mjukvaran. Lindhe och Ahmed [4] gav i sin magister uppsats några definitioner på vad en *unit* är:

- A unit in an application developed using a procedural programming language could be represented by a function procedure
- A unit in an application developed using an object-oriented programming language could be represented by a class or an instance of a class, or a method
- A unit in a visual programming environment or a GUI context could be a window or a collection of related elements of a window, such as a group box

Som man kan utläsa av punkterna ovan så beror det alltså på vad och hur man utvecklar. Rapporten kommer att diskutera detta i diskussions delen i kapitel 11.

Enligt Lindhe och Ahmed så är syftet med integrationstestning att se till att mjukvarans moduler kan interaktivera tillsammans på ett korrekt, stabilt och sammanhängande sätt innan systemtestningen. De ger även definitioner på vad en modul är som bygger på den implementerings teknologi som används: [4]

- En modul i ett objektorienterat språk kan vara representerad av en samling av objekt vilka skapar en väldefinierad tjänst som kommunicerar med en annan komponents modul via ett strikt gränssnitt.
- En modul i en visuell programmeringsmiljö kan vara en samling av små fönster vilka skapar en väldefinierad tjänst som kommunicerar med ett strikt definierat gränssnitt.
- En modul i en komponentbaserad utvecklingsmiljö skulle kunna vara en återanvändbar komponent som utgör en väldefinierad tjänst som kommunicerar med ett strikt definierat gränssnitt.

Integrationstestningen skall täcka följande enligt Lindhe och Ahmed [4]:

- Invocation of one module from another inter operating module
- Correct transmission of data between inter operating modules
- Compability (that is, checking that the introduction of one module does not have an undesirable impact on the functioning or performance of another module)
- Non-functional issues (such as the reliability of interfaces between modules)

### **3.3 Testfall**

Enligt Fewster [5] så finns det fyra attribut som påverkar kvalitén på ett testfall mer än andra och de är:

- Hur bra/effektiv testfallet är i avseende av defekt hittning.
- Ett bra test skall testa mer än en sak, där igenom reduceras det totala antalet testfall.
- Hur ekonomiskt ett testfall är att utföra, analysera och rätta fel.

- 
- Hur utvecklingsbart det är i avseende av hur mycket tid man måste lägga på underhåll på testfallet varje gång man ändrar i mjukvaran.

Man får väga de fyra attributen mot varandra för att hitta en balans mellan dem. Så testning handlar inte enbart om att hitta defekter utan även att undvika ytterligare kostnader. [5]

Man skall också försöka att få testfallen så snabba som möjligt, då forskning visat att ju längre tid det tar att köra testet desto större är risken att testet inte körs. [15]

### **3.4 Positiv och negativ testning**

Skillnaden mellan positiv testning och negativ testning är att när man använder sig av positiv testning så testar man enbart sådana testfall som man vet systemet skall klara av. Medan vid negativ testning så testar man extremfallen så som t.ex. otillåtna värden. [2]

Man skall alltid testa positivt så att man vet att mjukvaran gör vad den skall göra. Men man ska även testa negativt så att den inte gör vad den inte skall göra. [2]

### **3.5 Black-box och white-box testning**

*Black-box* testning är när man vet vad man ska få för utdata till en viss indata men inte vet vad som händer där i mellan. *White-box* testning vet man vad som händer inne i komponenten och kan därmed köra mer omfattande test så man vet att man täcker upp flera möjliga kombinationer.

### **3.6 Regressionstestning**

Regressionstestning är när man repeterar gamla tester efter man lagt till ny funktionalitet för att vara säker på att den nya koden inte skapat några nya defekter på den kod som fungerade innan.

### **3.7 Testdesign dokumentation**

Designen består av ett antal steg från en initial hög nivå strategi till en detaljerad testprocedur. Stegen är teststrategi, testplanering, testfallsdesign och testprocedurdesign. Designen av testfallen styrs av mjukvarans specifikation.

### **3.8 Automatiserad testning**

Vad som menas med automatiserad testning av mjukvaran finns det olika uppfattningar om. Men grunderna för alla automatiserade test är att man har testfall som är sparade så att de går att köra automatiskt. Sen om det automatiska testet körs automatiskt vid ett visst klockslag eller om kör igenom alla testfall manuellt så utnyttjar man i båda fallen fördelen med att testen kan köras om och om igen utan någon extra kostnad. Detta gör att

---

man kommer att testa oftare vilket förhoppningsvis gör att man kommer att upptäcka defekterna tidigare. Färre fel i slutet av projekten betyder också att man får mer exakta leverans datum [15].

### 3.8.1 Fördelar

Vilka är nu fördelarna med automatiserad testning? Hayes [7] beskriver i sin bok om de tre huvudfördelarna som man får från automatiska tester, *repeatability*, *leverage*, och *accumulation*.

- *Repeatability*, att man kan exekvera testen fler än en gång.
- *Leverage*, att vissa test aldrig kördes på grund av att de inte gick att köra manuellt.
- *Accumulation*, vikten av att samla alla test i ett test bibliotek design som stöder underhåll av testen under applikationens livstid.

Enligt Boehmer och Patterson är en av fördelarna att man hittar regressionsdefekter mycket tidigare. Enligt enkäten som utfördes så visade det sig att idag så körs nästan ingen regressionstestning alls. Den enda regressionstestning som utvecklaren utför är att testa det som gick fel, än att rättningen han utförde kan ha påverkat koden på andra ställen. Men med ett automatiskt test så skulle man snabbt kunna köra igenom hela eller närliggande delar av systemet för att se om några regressionsdefekter uppstått. [9]

En annan fördel är *refactoring*, med ett automatiskt *unit* test i ryggen vågar utvecklaren ändra/snygga till i koden utan att oroa sig för att det kommer att bli några regressionsdefekter. Fördelen med *refactoring* är att koden blir mer lättförståelig vilket underlättar för framtida ändringar. Men man bör beakta att vissa testfall kan vara så pass avancerade så de kommer att påverkas av att man snyggar till i koden vilket kan göra att utvecklaren drar sig för att göra det för att slippa ändra testfallen.

Fler fördelar med ett automatiskt test presenterar Ahmed och Lindhe [4] i sin sammanfattning med följande punkter:

- Testen kan köras snabbare.
- Testen är mera överensstämmande.
- Testen kan köras om och om igen utan någon *overhead*.
- Det tillåter test som inte skulle ha gått att utföra manuellt.
- Det höjer utvecklarens moral och självförtroende.
- Tiden för testningen kan kortas ner.
- Frigör duktiga testare till att lägga mer tid på att rätta till buggar och skriva bättre testfall.
- Produktion av ett tillförlitligt system.
- Höjning av kvalitén på test satsningen.

### 3.8.2 Realistiska förväntningar

Buying a test tool is like joining a health club: the only weight you have lost is in your wallet! You must use the club, sweat it out and invest the time and effort before you can get the benefits.

---

Citatet ovan kommer från Linda Hayes handbok om automatiserad testing [7], det påvisar vikten av att ha realistiska förväntningar. Om man har problem med testningen skall man inte tro att det hjälper att köpa ett dyrt automatiserat testprogram och allting skall lösa sig.

Man ska heller inte börja varsla testarna om uppsägning i tron att det automatiska testet skall göra deras jobb. Antagligen så har företaget redan för få testare och man kan aldrig testa för mycket, frågan företaget får ställa sig är om de vill spara in några testtjänster eller höja produktkvalitén.

### 3.8.3 Vilka test skall man automatisera

Vilka test skall nu automatiseras? Denna fråga är intressant för att det automatiska testet skall bli så effektivt som möjligt. Det är inte nödvändigt att automatisera alla testfall som går att automatisera utan man skall utvärdera alla testfallen för att se vilka som behövs för att nå sina mål med automatiseringen.

Områden där test automatisering kan vara lönsamt har Fewster [5] identifierat följande:

- Okomplicerade test. Med okomplicerade menas att man har kända input och förväntade resultat för testobjektet.
- Test som är svåra att testa manuellt. Vissa tester kan vara svåra att testa manuellt till exempel prestanda test med flera tusen användare.
- Icke funktionella krav, till exempel test av prestanda.
- Regressionstestning. Kontroll av att gammal kod fortfarande fungerar när man ändrar eller introducerar nya komponenter.
- De viktigaste testen, att de viktigaste testen körs varje gång det automatiska testet körs.
- En samling av breda test.
- Test av de viktigaste funktionerna.
- Test som är lätta att automatisera.
- Test som ger snabb återbetalning.
- De test som körs oftast.

Boehmer [9] säger att man måste definiera en tydlig process för att bestämma vad som skall automatiseras. Följande riktlinjer ger han för kriterierna för ett automatiskt test mål:

- Automatiserad regressionstestning: Eftersom vinsten av investeringen från automatiseringen kommer från återanvändningen av testfallen, så skall man börja med att rikta in sig på regressionstesterna. Man skall automatisera alla testfall som skall köras i varje byggning, men de testfall som enbart skall köras en gång är inte värda att automatisera.
- Automatisera testfall för stabila applikationer: Innan man börjar skriva automatiska testfall till en applikation så skall man fundera på om applikationen kommer att ändras mycket i framtiden. Om den kommer ändras mycket så måste man även ändra i de automatiska testfallen och det kan blir mycket extra arbete vilket gör att det inte blir lönsamt att automatisera.
- Automatisera inte tids beroende test: Att man inte skall automatisera om man har komplexa tidsfrågor. Om ett test är svårt att automatisera, kör det manuellt istället.

- 
- Automatisering är inte alltid lösningen, 100 % testfalls täckning ska inte vara målet.
- Automatisera upprepande test: Om ett test är extremt återkommande och tråkigt så är det perfekt att automatisera.
  - Automatiserade test som är nerskrivna: Du måste ha detaljerade testfall som är repeterbara innan du startar att automatisera. Skriv alltid testfallen innan du automatiserar dem. Det försäkras att testen är skrivna för att testa funktionaliteten oberoende av idén att automatisera dem.
  - Begränsa omfattningen: Försök inte att automatisera allting. Uppnå små framgångar, sen kan du öka din omfattning när du gör framsteg. De flesta mjukvaruutvecklingsteam hamnar ofta i tidsbrist ibland, men lägg tid när du har tid över på att utöka det automatiska testet.

Marick [10] rekommenderar att man använder sig av en process där man ställer sig flertalet frågor för att ta reda på om man skall automatisera eller ej. Några av frågorna följer nedan.

- Automatisera ett test kostar mer än att köra det manuellt en gång, hur mycket mera?
- Ett automatiserat test har en begränsad livstid, under den tiden ska den ha sparat igen den extra kostnaden för automatiseringen. När kommer testet att dö ut? Vilka händelser kan få testet att dö ut?
- Under testens livstid, hur stor är chansen att de hittar buggar (förutom de som hittades första gången testet kördes)? Hur mycket väger denna osäkra fördel upp kostnaden för automatisering?

Marick [10] hade även ytterligare två punkter man bör tänka på:

- Att människor kan upptäcka vissa buggar som inte ett automatiskt test kan upptäcka.
- Verktyg är bra när man får exakta resultat som en människa skulle kunna missa.

Som tillägg till det ovan så avslutar Marick med två saker som han tror är bland det viktigaste. Den första är hur man skall göra mätningar på kostnaden för test automatiseringen. Där tror han det bästa sättet är att mäta hur många manuella test som du inte behöver köra och de buggarna som de skulle ha hittat. Den andra är om det automatiska testet uppfyller sitt syfte för det specifika testet. Enligt Marick så ligger värdet i hur bra man lyckas uppfylla dessa två punkter. [10]

### **3.8.4 När man inte skall automatisera**

Det är inte alltid rätt att automatisera, Hayes har några punkter som man bör beakta innan man sätter igång. [7]

- Instabil design, vissa applikationer som t.ex. väderleksapplikationer som bygger på realtiddata är svåra att automatisera då du inte har något känd utdata att jämföra resultatet med.
- Om personen som skriver de automatiska testen inte har nog med erfarenhet från applikationen. Vilket kan göra att testet inte reflekterar applikationens rätta beteende. Ett automatiserat test är aldrig bättre än personen som skrev det.

- 
- Tillfälligt anställda, det är inte heller lönsamt att sätta tillfälligt anställda på att skriva automatiserade test.
  - För lite tid eller resurser, om du inte har resurserna för att hinna med den manuella testningen så ska du inte heller räkna med att ett automatiskt test ska hjälpa dig att hinna. Den inledande investeringen för planering, träning och implementering kommer att ta mer tid än vad du kommer att spara.

### 3.8.5 Underhåll av automatiserade tester

Att kostnaden för underhållet blir högre för automatiska testfall jämfört med manuella ter sig självklart då man även måste ändra i sina automatiska testfall om man gör ändringar i koden. Kommer i detta stycke presentera några av kännetecknen för vad som påverkar underhållet enligt Fewster. [5]

- Antal testfall, desto fler testfall man har i sin testsvit desto fler får man underhålla. En lösning på detta problem är att innan man lägger till ett nytt test funderar på vad testet kommer att tillföra testsvit, både i chansen att upptäcka en defekt och trolig underhållskostnad. Detta kommer att försäkra att man inte lägger till testet för sakens skull utan att man har tänkt på underhållskostnaden först. En annan lösning är att gå i genom testsviten och rensa bort alla överflödiga testfall. Även de testfall som kostar mer än vad man kan spara på dem.
- Datamängd, ju mer testdata det finns desto mer underhåll behövs. Det är inte enbart uppdatera data som tar tid även att hantera den tar också tid.
- Dataformat, ju mer specialiserat dataformat som används, desto större är sannolikheten att en speciell testverktygssupport måste startas.
- Tiden för att köra testfallen, flertalet test är ineffektiva då de är mycket lika varandra samt har låg återanvändningsgrad. Detta kan undvikas genom att hålla funktionella testfall så korta och fokuserade som möjligt.
- Spårbarheten i testfallen. När ett test misslyckas, hur vet jag vad som gick fel? Om den enda informationen som kommer är att testet misslyckas. Då kan fel analys och felrättning vara svår, då användaren inte vet vad som misslyckas.
- Oberoende test, att testen är så oberoende av varandra som möjligt. Att t.ex. utdata från ett test blir indata i ett annat, detta kan skapa följd fel.
- Namnsättning, när antalet testfall ökar så skulle det bli kaos om man inte använde någon förutbestämd namnsättning.
- Test dokumentation, odokumenterade eller dåligt dokumenterade testfall skulle leda till en kaotisk situation och tid skulle försvinna vid underhållet.

---

Att minimera underhållskostnaden är en av de svåraste utmaningarna vid test automatisering. Testfallen behöver vara robusta för ändringar vid defekträttning och vid nya produktversioner. [15]

### 3.8.6 Record/playback metoden

Denna metod fungerar så att man spelar in olika användarfall som datorn sedan spelar upp automatiskt. Ett exempel är att man t.ex. startar upp applikationen man vill testa, startar inspelningen och sedan trycker på de knappar som krävs för att testa den efterfrågade funktionen. När man är klar så kan man spela upp sitt användarfall och datorn simulerar då exakt samma sak som du spelade in innan. [11]

Nackdelar med denna metod är om man t.ex. har med funktioner som en klocka eller datum i applikationen så kommer programmet att meddela att något är fel. Det är även mycket dyrt att underhålla metoden. En annan nackdel är att skripten är hård kodade<sup>2</sup> så om man ändrar något i applikationen så måste man spela in alla användarfall igen. [11]

### 3.8.7 Forskning

På Siemens Building Technologies införde de automatiska test på åtta applikationer. Vars skript kördes på varje testcykel för att verifiera att inga regressionsdefekter uppstår. Undersökningen visade på att skripten minskade den totala testexekverings tiden. I en av applikationerna minskade den från 55 timmar till 6 timmar och i en annan minskade från 21 timmar till 7 timmar. [9]

I en undersökning som utfördes på Ericsson i Karlskrona gjord av Damm och Lundberg där de införde TDD och ett automatiserat test samtidigt så kunde de uppmäta en minskning av antalet defekter innan systemtestningen. På två olika produkter uppmättes förbättringar på över 29 % på antalet upptäckta defekter i systemtestningen jämfört med innan. [23]

I de fall som automatiseringen har lyckats så har man fokuserat på att automatisera vissa delar istället för att försöka att automatisera allting. Man har även haft tillgång till kompetent personal skriver Kerry i sin artikel. [6]

---

<sup>2</sup> Hård kodad – När man använder ett värde i koden på flera olika ställen istället för att använda en variabel där man bara skulle behöva ändra på ett ställe istället för flera olika ställen i koden.

---

## Kapitel

# 4 Testning på UIQ

---

### 4.1 Inledning

På UIQ är testningen uppdelad i tre större delar, *unit* testning, integrationstestning samt systemtestning. Den första delen utförs av utvecklaren själv medan de andra utförs av *product verification* som har hand om enbart testning.

### 4.2 Unit testning

*Unit* testningen ligger på utvecklarens ansvar att utföra samt dokumentera. Innan kodningen startar skall man skriva en *unit test approach* som skall bygga på en redan befintlig *unit test* specifikation. *Unit test* approachen skall beskriva hur *unit test* specifikation följs i ett givet *work package* (WP). Den skall lista testobjekten (komponenter/klasser) samt definiera testfokus för varje testobjekt.

Idag så görs *unit* testningen i slutet av utvecklingen. Men i vissa fall under tidspress och dylikt så finns risken att *unit* testningen prioriteras bort i slutet vilket även intervjuerna som utfördes visade på.

Det som skall testas på *unit* testnivån står beskrivet i *unit test* specifikationen. Om det är möjligt så skall man bland annat testa efter problem med minnesläckor, prestanda, *heap failure*, *boundary* m.m. I kapitel åtta så har rapporten en mer utförlig presentation av alla testfall.

### 4.3 Systemtestning

I denna fas så testas *product verification* systemets funktionalitet och stabilitet samt icke-funktionella krav så som prestanda och tillförlitlighet. I systemtestningen så använder man sig oftast av *Black-box* tekniken, alltså att man skickar in ett värde och jämför det med förväntad utdata utan att ta hänsyn till hur det fungerar där i mellan.

### 4.4 Testprogramvara

På UIQ finns det tillgång till flertalet olika testprogram, vilka de vanligast kommer att beskrivas närmare nedan.

#### 4.4.1 TEFUnit

*Test Execution Framework Unit* (TEFUnit) är ett egenutvecklat testverktyg som introducerades på företaget hösten 2004. TEFUnit är ett ramverk där man lägger in sin

---

kod tillsammans med sina egenutvecklade testfall. Resultatet presenteras sedan i html format där man kan se vilka av sina testfall som har passerat eller misslyckats.

#### 4.4.2 Blueberry

Blueberry är ett egenutvecklat testverktyg för att testa mjukvarans *user interface (UI)*. Blueberry utvecklades för det inte fanns något bra verktyg för att testa UI med idag på marknaden. I den ej releasade UIQ mjukvaran finns det ett inbyggt stöd så att alla sidor representeras i XML format. Med hjälp av XML representationen jämför Blueberry det testade UI elementet med det förväntade utfallet.

#### 4.4.3 CodeTest

Detta testverktyg används ej idag men finns tillgängligt. CodeTest är ett verktyg för att se hur mycket av koden som exekveras. Detta kan vara användbart för att se hur stor del av utvecklarens kod som testfallen täcker. Med hjälp av detta program skulle man sedan kunna få reda på hur stor del av koden som är testad.

#### 4.4.4 Lint scan

Lint scan kan man likna som en noggrannare kompilator. Programmet testar koden efter programmerings fel som kompilatorn missat att upptäcka. Man får sedan resultatet presenterat i två olika värden. Först en poäng som räknar om alla fel och varningar till en totalsumma. Sedan får man även en nivå som är fördelad på en skala noll till fyra där fyra är en näst intill felfri kod medan ett noll värde betyder att man har allvarliga fel i koden. Man kan även konfigurera Lint scan så att den ändrar känslighet.

Idag så skall Lint scan köras regelbundet men när vi körde ett provtest på en komponent så upptäckte vi flertalet fel och varningar vilket tyder på att de slarvar med användningen. Även enkät resultatet visade på det då 10 % svarade att de inte använde Lint scan.

#### 4.4.5 Leave scan

Leave scan är ett testverktyg utvecklat av Symbian. I Symbian programmering så skall alla funktionsnamn där funktionen *leavar* (se definition i kapitel 14) sluta med bokstaven L. Detta för att t.ex. tredjeparts utvecklare skall veta att den funktion *leavar* för att då kunna ta hand om eventuella fel.

#### 4.4.6 Emulator

UIQ har en egen utvecklad emulator som simulerar en UIQ plattform. I denna finns det inbyggda funktioner för att bland annat testa efter *heap failure* med mera. Emulatorn kommer ej att gå och implementera i ett AUT då man inte kan automatisera den.

#### 4.4.7 Doxygen

Detta verktyg kontrollerar kommentarer i koden. Doxygen ser till att koden man skriver kan omvandlas till en godkänd SDK.

---

## Kapitel

# 5 Produktutvecklingsprocessen på UIQ

---

### 5.1 Inledning

I detta kapitel presenteras läsaren lite kort om de olika delarna i produktutvecklingsprocessen på UIQ.

Tre viktiga uttryck i detta kapitel är milstolpe, *toll gate* samt *work package* (WP). En milstolpe är en viktig mätbar tidpunkt i projektet, när denna nås skall några förutbestämda krav vara uppfyllda. Efter varje milstolpe så håller man en milstolpes granskning där man samlar en grupp människor som hjälper projektledaren att fatta ett beslut om man skall fortsätta till nästa fas eller ej. Men det är alltid projektledaren som fattar det avgörande beslutet om man skall fortsätta.

*Toll gate* är en brytpunkt då man stannar upp och utvärderar om man skall fortsätta med projektet eller ej.

När man har ett större projekt brukar man bryta ner det i mindre delar och på UIQ kallas varje mindre del för *work package*. Som exempel är det senaste projektet uppdelat i cirka 40 WP.

### 5.2 D-modellen

D-modellen är en variant av PROPS som är anpassad till UIQ för att möta deras krav. En av fördelarna med D-modellen jämfört med PROPS är att den hela tiden uppdateras och förbättras med hjälp av erfarenheter från gamla projekt. D-modellen förbättrar även *team* produktiviteten genom att tillhandahålla alla *team* medlemmar lätt access till samma följdlinjer och mallar för alla aktiviteter i projekt livscykeln. Det finns även en R-model där R står för *release* men den används enbart när man ska göra en *release*.

I figur 2 i appendix III kan man se att D-modellen är uppdelad i samma fyra faser som PROPS. *Pre-study*, *feasibility*, *execution* och *conclusion*. Man ser även fem *toll gates* (TG0-TG5) där man stannar upp för att se om man skall fortsätta med projektet eller ej. Som appendix till rapporten finns mer information om D-modellen och dess *toll gates*. D-modellen används enbart på projekt inom UIQ internt så det är aldrig aktuellt med en *release* med D-modellen till en kund.

### 5.3 Mjukvaruutvecklingsprocessen

Processen är byggd runt ett koncept där man använder sig av *work package* (WP) som beställs av ett projekt och levereras tillbaka av utvecklingsteamet. Det är *team* ledarens ansvar att utveckla samt leverera WP efter mjukvaruutvecklings modellen de gula delarna

---

i figur 1 i appendix III. Ett WP består av en samling närliggande krav som man ger till ett team som uppgift att utveckla. *Team* ledaren delar sedan upp WP i mindre delare som fördelas till teamets utvecklare. Utvecklaren har sedan som uppgift att utveckla funktionaliteten som att de uppfyller kraven, *unit* testa samt releasa sin kod.

I appendix III figur ett kan man se en översiktsbild hur produktutvecklingsmodellen ser ut, den del som vi är intresserade av är den gul markerade PROPS delen. Figuren är uppdelad i flertalet olika rutor, varje ruta motsvarar en milstolpe och den skall vara avklarad innan man går över till nästa. Mellan milstolparna S3A och S3B använder man sin valda utvecklingsmetodik som vi kommer att gå igenom nedan.

## 5.4 Utvecklingsmetodiken

Utvecklingsmetodiken är den metodik som utvecklaren använder för att skriva sin kod med tillhörande testning. Idag så använder sig företaget av metodiken där man först skriver sin kod för att sedan skriva och exekvera sina testfall som man skrev i *unit* test specifikationen. Under intervju fas två så visade det sig att det fanns de som skriver vissa egna testprogram under kodningsfasen. Nedan kan ni se hur dagens utvecklingsmetodik ser ut på UIQ.

Specifikation/analys → Design → Kodning → Testning → *Release*

De två vanligaste utvecklingsmetodikerna som används idag är vattenfallsmodellen samt *ad-hoc* modellen. Vattenfallsmodellen är tydligt uppdelat i olika faser och milstolpar, den utförs i ordningen analys, design, programmering och testning. *Ad-hoc* modellen är snarlik men med skillnaden att de olika faserna och rollerna kan överlappa varandra. [12]

## 5.5 Release/integration

När komponenten uppfyller några visst förutbestämde krav så kan man göra en *release* till *baseline*. *Baseline* byggs om ca två gånger i veckan av en ansvarig som ser till att alla komponenter fungerar ihop. Ett problem som finns idag är ibland så grundar sig en komponents kod på en gammal baseline vilket kan ge problem då t.ex. API kan ha blivit ändrade i en annan komponent som kommer att generera ett fel.

---

## Kapitel

# 6 TDD, Agile och XP

---

### 6.1 Inledning

I detta kapitel kommer vi att undersöka utvecklingsmetodikerna TDD och XP närmare. Vi kommer även att undersöka Agile Modelling (AM) vilket är en metodik för att skapa dokumentation vid *agile* utveckling (som t.ex. XP och TDD). Anledningen till att vi undersöker nya utvecklingsmetodiker är att utvecklingsmetodiken som används idag på UIQ där man skriver testfallen sist gör att de lätt kan prioriteras bort vid tidsbrist [15]. Därför finns det ett behov av en metodik där man skriver testen innan man utvecklar koden.

### 6.2 Test-Driven Development

Test-driven development practice showed, during functional verification and regression tests, approximately 40% fewer defects than a baseline prior product developed in a more traditional fashion. [13]

Test-Driven Development (TDD) är en mjukvaruutvecklings metodik som har använts sporadiskt i flera decennium. Redan i tidiga 60-talet använde NASA TDD i sitt Mercury projekt [13]. Men inte förens i samband med att XP har börjat användas mera så vändes blickarna mot TDD igen.

Det som skiljer TDD från de flesta andra utvecklingsmetodiker är att man skriver sina testfall innan man börjar skriva sin kod. Detta gör så att utvecklaren blir tvingad att skriva testfall så man undviker att utföra en *release* där koden inte är testad ordentligt.

TDD fungerar så att efter man skrivit ett testfall så skriver man endast så mycket kod att man precis klarar testet. Sedan börjar man om och skriver ett nytt testfall som man sedan skriver kod till osv. Detta fortsätter tills man är klar med sin uppgift, då har utvecklaren en fungerande kod med tillhörande samling testfall.

Till skillnad från XP där implementeringen inte föregås av någon formell design så stöder TDD både design och *unit* testning. Men vissa tror att om man följer TDD strikt så kan man minimera om inte eliminera behovet av *upfront* design. Hur som helst så är TDD flexibelt och kan stoppas in i flera olika processer. Inkluderat de som använder sig av *upfront* låg nivå design.

George och Williams's skriver att: "If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding". Detta sammanfattar idén med TDD, om man inte kan skriva ett enkelt testfall så klarar man heller inte av att skriva

---

koden. Att man tvingar utvecklaren att tänka till lite innan han startar med kodningen gör att utvecklaren kommer att få bättre förståelse för sin uppgift. [8]

### 6.2.1 Forskning

Det finns flertalet undersökningar som har jämfört TDD med mer traditionella utvecklingsmetodiker (först koda sen testa). Dessa undersökningar har haft blandade resultat vilka jag kommer att presentera ett urval ifrån nedan.

I en undersökning av George och Williams utförd år 2003 där de använde sig av både TDD och par programmering så kom de fram till att det tog 16 % längre tid men man passerade 18 % fler funktions Black-box testfall genom att använda sig av TDD jämfört med vattenfallsmodellen. Gruppen som använde sig av vattenfallsmodellen skrev nästan heller inga automatiserade test som var användbara. [8]

En annan undersökning som gjordes på 19 studenter på University of Karlsruhe visade på följande resultat när de jämförde TDD med vattenfallsmodellen:

- Det fanns inga skillnader i utvecklingstiden mellan grupperna.
- Koden hade lägre tillförlitlighet efter implementerings fasen medan den hade högre tillförlitlighet efter *acceptance* fasen för TDD.
- TDD gruppen hade statistiskt sätt färre fel när koden skulle återanvändas.
- Sammanfattningsvis så kan man säga att tidsmässigt samt kvalitets mässigt så är metodikerna likvärdiga. Det som talar för TDD är att dess kod blev mer lättförståelig. [14]

IBM gjorde ett försök med TDD som är extra intressant då det som på UIQ var en plattform som användes i försöket. Där jämförde de sedan sin första *release* med sin sjunde. I de första sex så använde de sig av *ad-hoc* metodiken medan i den sjunde så använde de sig av TDD och skapade 2400 automatiserade *unit* test. Teamet upptäckte en 40 % minskning i densiteten (defekter/rader kod) på funktion verifikations defekter av ny/ändrad kod när man jämförde med ett erfaret *team* som använde sig av *ad-hoc* metodiken. [13]

En svensk undersökning som gjordes på Ericsson AB i Karlskrona där man testade att använda sig av TDD fick de fram preliminära resultat som pekade på att konceptet minskade utvecklingstiden markant. [15]

### 6.2.2 Nackdelar

Självklart finns det även nackdelar med TDD, George och Williams [8] nämner fyra områden där de tycker att TDD kan vara sämre än vid vanlig utveckling. Den första är saknaden av design då det inte är ett krav att man använder sig av *upfront* design. Nackdelen med det är att vid ett senare tillfälle så kanske någon annan än utvecklaren vill gå in och ändra i koden och då tar det längre tid att sätta sig in i koden. Nästa nackdel är att det visat sig i efterhand att tekniker och notationer utvecklade för design av mjukvaran blivit integrerade i implementerings processen. Sådan integration har tenderat att sudda eller åtminstone förvirrat skillnaden mellan design och implementering. Den tredje

---

nackdelen är att när koden blir för komplex så kan det bli mycket svårt att städa upp i den och *refactoring* är nödvändig för att underhålla och reducera komplexiteten i koden. Den sista nackdelen som George och Williams nämner är den att man måste ha erfarna utvecklare för att kunna skapa avancerade testfall. Medel utvecklaren kan sakna den krävda erfarenheten vilket kan resultera i kod som saknar ordentliga testfall samt dokumentation.

### 6.2.3 Fördelar

Fördelarna med TDD är många och kommer att presenteras i detta stycke. En av fördelarna med TDD är att man får kontinuerlig återkoppling på kodkvalitén, det med hjälp av de automatiska testerna som du kan använda regelbundet. Vilka även kan användas vid regressionstestning skriver Damm [15] i sin artikel. Han skriver även att man kan använda testfallen som en del av designen och på så sätt kunna skära ner en del på tiden som läggs på design.

Anthes framhäver att man får självförtroende att våga städa upp i koden då man har automatiserade test som kan bekräfta att all funktionalitet fungerar även efteråt. [16] Även att du alltid har en fungerande kod som du skulle kunna göra en *release*.

Andra fördelar som George och Williams [8] nämner är att en som inte är insatt i koden kan få en bättre förståelse genom att enbart titta på testfallen som utvecklaren har gjort, att utvecklaren ofta förklarar koden med testfall och sin kod istället för med beskrivande ord. En annan fördel är att testfallen alltid är uppdaterade. Det finns studier som visar på att halva uppgiften för utvecklaren vid underhåll är att sätta sig in i koden. George och Williams fortsätter med att TDD är effektivare då defekterna identifieras snabbare när ny kod läggs till därför att felkällan är lättare att bestämma. Baserad på tidigare undersökningar så kompenseras tiden av att skriva och köra testerna med tiden som sparas på grund av effektivare defekt borttagning och även förkortningen av *debug* tiden. TDD driver även utvecklarna till att skriva kod som man kan använda automatiserad test på till skillnad från kod som är utvecklad med traditionell utvecklingsmetodik. Ett exempel är funktioner och metoder som returnerar ett värde som kan jämföras med ett förväntat värde. De skrivna *unit* testerna som skrivs med TDD är värdefulla tillgångar till projektet i senare skeden.

## 6.3 eXtreme Programming

Extreme Programming (XP) är en 13 år gammal mjukvaruutvecklingsmetodik. Metodiken är designad för att leverera mjukvaran som kunden efterfrågade och i tid. XP gör det även möjligt för kunden att komma med kravändringar sent i produktens livscykel.

XP förbättrar ett mjukvaruprojekt på fyra viktiga punkter:

- Kommunikation: XP programmerare kommunicerar med sina programmerare samt med sina kunder.
- Enkelhet: De håller sin kod enkel och ren.
- Återkoppling: De får återkoppling genom att de testar sin kod från dag ett.

- 
- Mod: Modet får de från att de följer de tre punkterna ovan så att de kan leverera systemet så snart som möjligt och kan implementera önskade ändringar från kunden. [17]

### 6.3.1 Forskning

På Sabre Airline så bytte man från en konventionell utvecklingsmetodik till XP efter att ha haft stora problem med en *release* av sin mjukvara. I sin sista *release* när de använde en vanlig utvecklingsmetodik upptäcktes 26 defekter redan de tre första dagarna jämfört med 10 defekter de först två månaderna för den *release* som använde sig av XP. Man bör tillägga att samtidigt som man bytte metodik så bytte man även från c/c++ till Java men en talesman för Sabre säger att det var XP och inte Java som låg bakom den drastiska kvalitetsförbättringen. Ett annat projekt som gjordes på samma företag vilket innehöll 15 000 rader kod där de också använde sig av XP metodiken hade inte en enda defekt 20 månader efter sin *release*. [16]

I en undersökning av Williams som gjordes på 42 studenter på University of Utah där de jämförde par programmerare med singelprogrammerare så gjorde de tre test för att se hur lång tid det tog för utvecklarna att komma in i par programmeringen. I det första testet så tog par programmerarna 60 % mer tid på sig än singelprogrammerarna. I nästa test tog det 20 % och i det sista 10 % mer tid. Efter undersökningens slut så tyckte över 90 % av försökspersonerna att XP var bra samt att de kände sig tryggare när de programmerade i par. [22]

XP är anpassat för projekt med maximalt 20 programmerare men det optimala är 12 programmerare. Det har gjorts en undersökning där de har delat upp ett större projekt i mindre XP grupper. Men McBreen tyckte inte att det är någon bra idé att använda sig av XP i sådana stora projekt. [12].

### 6.3.2 Nackdelar

Det finns även nackdelar med XP, den mest uppenbara nämnde Williams [22] ovan att det tar längre tid. När man använder sig av par programmering så tar det enligt undersökningar ca 10 -16 % längre tid jämfört med en traditionell utvecklingsmetodik. En annan nackdel är att det kan finnas stort motstånd från utvecklarna att prova nya utvecklingsmetodiker.

Den sista nackdelen nämner McBreen [12] i sin artikel. Att det kan vara svårt att kompromissa mellan XP och *upfront* design utvecklingsmetodiker. Vissa XP anhängare påstår att koden är designen när man använder sig av XP.

### 6.3.3 Fördelare

När man jobbar i par så lär man sig mycket av den andra. Ett bra citat som sammanfattar detta är Houglands: "You have the weaker people paired with the stronger people and business knowledge and coding knowledge are transferred very quickly" [16]

Det är alltid två stycken som är insatta i koden. Att man sprider ut kunskapen bland flera. Man kan även flytta om i paren så de kodar olika saker varje gång för att de ska bredda

---

sina kunskaper. Man kan även byta par medlemmar så att programmerings kunskaperna sprids.

Det blir högre kodkvalité. Enligt undersökningar som beskrevs i forsknings stycket så minskar man antalet defekter som uppstår i systemtestningen samt hos kunden när man använder sig av par programmering.[22]

## 6.4 Agile model

Agile models are more effective than traditional models because they are just barely good enough, they don't have to be perfect. [24]

Ovan förklaras grundidén med *Agile modelling* (AM), man ska inte krångla till det med onödig design utan enbart dokumentera det som behövs. AM är en praktiserbar metodik för effektiv utveckling av modeller och dokumentation till mjukvaru baserade system. Enkelt förklarat kan man säga att AM är en samling av värderingar, principer och praktiker för utveckling av modeller för mjukvaruutvecklings projekt på ett lätt och effektivt sätt.

AM kompletterar XP och *Unified process* i dokumentation fasen. Vissa utvecklare tycker att XP innehåller för lite dokumentation, då kan man använda AM för att föra in mer dokumentation i projektet. Medan i *Unified process* känner vissa utvecklare att det är för mycket dokumentation och då kan man använda AM för att minska den.

Några av principerna med AM presenterar Ambler [25] med följande punkter:

- Utveckla modeller med ett syfte, om du inte vet varför och till vem du gör det varför då göra det? Det första man skall göra är att ta reda på vem som skall ha informationen och sedan enbart skapa tillräckligt med information.
- Maximera intressenternas investering, det är de som betalar din lön. Se därför alltid till att uppfylla deras önskemål.
- Res lätt, för varje modell du skapar desto mer underhåll blir det.
- Snabb återkoppling, när man jobbar tillsammans med en modell och använder sig av gemensam model teknik (*whiteboard* m.m.) så får man hela tiden återkoppling på det man gör.
- Tillämpa enkelhet, ofta är den enklaste metoden att lösa ett problem den bästa.
- Anta ändringar, var alltid beredd på att det kommer att bli ändringar under projektets gång.
- Inkrementella ändringar, man ska inte gå in för att få det rätt direkt. Än man skulle försöka så är det stor risk att man misslyckas.
- Kvalitets arbete, ingen gillar slarvigt utförda arbeten. Utvecklaren som skrivit koden kan inte vara stolt över den, de som skall *refactoring* koden blir sura och kunden blir sur för att han får en dålig kod.
- Mjukvara är ditt primära mål, målet är att utveckla den efterfrågade koden åt kunden inte massor med dokumentation eller annat.
- Tillhandahålla en lätt utbyggnad av mjukvaran är det sekundära målet, än man har en fungerande kod som man ger till kunden så är det ett misslyckande om man inte enkelt kan lägga till ny funktionalitet.

---

## Kapitel

# 7 Undersökningar på UIQ

---

### 7.1 Inledning

Detta kapitel kommer att behandla de undersökningar jag utförde på UIQ. För att samla in så mycket information som möjligt så har jag gjort flertalet olika sorters undersökningar.

### 7.2 Intervju fas ett

Syftet med första intervjun var att ta reda på om det var vanligt med minnesläckor/*heap failure* samt vilka verktyg de använde för att upptäcka dem. Intervjupersonerna valdes från de olika avdelningarna för att kunna upptäcka om det skiljde sig åt hur de löste problemen mellan avdelningarna. Försökte även att plocka personer som jobbat på företaget olika länge för att se om de nyanställda eventuellt hade tagit med sig några erfarenheter från skolan eller tidigare arbeten.

#### 7.2.1 Minnesläckor

Det viktigaste resultatet från undersökningen var vilka metoder som användes för att hitta defekterna. De två metoderna som användes bland utvecklarna var att använda UHEAP<sup>3</sup> makron och att stänga ner applikationen för att se om det uppstod något problem. Utvecklarna testar även koden efter minnesläckor regelbundet samt de flesta tyckte även att det var relativt vanligt att man hittade läckor. Tiden för att åtgärda en minnesläcka uppskattade de i genomsnitt till ett par timmar.

#### 7.2.2 Heap failure

Här skiljde sig svaren åt lite mera, både tiden för att åtgärda defekten samt om det var vanligt att man upptäckte defekten skiljde sig åt. Tiden att åtgärda en defekt varierade från ett par timmar upp till en dag. Hur man testade efter *heap failure* var däremot densamma, där alla använde sig av emulatorn.

### 7.3 Intervju fas två

Denna intervju fas gjordes för att få en bättre uppfattning hur om utvecklarna jobbar. Det jag ville få ut av undersökningen var mer information om processen samt eventuella brister. En del av resultaten låg också som underlag till den efterföljande enkäten.

---

<sup>3</sup> UHEAP – Är ett makro utvecklat för att hålla reda på hur mycket minne som är allokerat. Man placerar ut ett makro på var sin sida om koden man vill testa, om sedan nivån på minnet skiljer sig åt avbryts skriptet.

---

Första frågan om vilka dokument de var delaktiga i så var detsamma för alla deltagare förutom konsulten som enbart hade hoppat in som en extra resurs i ett team. De var med och skrev *functional specification* (FS), *execution proposal* (EP), *technical investigation* (TI), *unit test* specifikationen (om den inte ligger i EP) design dokument och estimerings dokument. De enda dokumenten som de ändrade i under kodningen var EP samt eventuella justeringar i estimerings dokumentet.

Som förtydligande vilket inte framgår i formuläret men som jag påpekade under intervjuerna så syftade jag på den egen skrivna *unit test* specifikationen (kan ligga i EP). Tiden som de olika utvecklarna hade planerat in för *unit* testning jämfört med den totala tiden för utvecklingen varierade från 5-18 %. Tiden som de sedan lade på testning blev mindre än vad de planerat, vilket skyldes på tidsbrist.

Varje kodare är ansvarig för att releasa sin egen kod, det finns en process på projektets hemsida hur man skall gå tillväga. Men det man kan notera är att vissa utvecklare lade upp kod som inte hade all funktionalitet men som fungerande, detta för att det är lättare att integrera när man gör en *release* ofta.

Utvecklarna gick inte i genom hela *unit* testspecifikationen efter att de åtgärdat en defekt som upptäcks av *product verification*.

Som väntat så använder sig alla av utvecklingsmetodiken där de kodar först samt testar sen. Några skrev testprogram för att testa funktionaliteten under utvecklingens gång. De flesta tyckte även att utvecklingsmetodiken som de använder nu var dålig då testen kom sist så att om det blev tidsbrist då blev testningen lidande. Ett förslag var att man skulle planera in småtest efter varje del. Alla tyckte även att tiden för testning var för kort för att hinna testa fullständigt.

## **7.4 Intervju fas tre**

Denna intervju gjordes för att få reda på vad de högre instanserna på UIQ hade för förväntningar och tankar om det automatiserade *unit* testet. Resultatet från denna intervju visade på att personen hade förhoppningar på att det automatiserade testet skulle minska utvecklingstiden efter *function complete*. Han hade inte heller som mål att dra ner på personal utan istället att höja kvalitén på koden.

## **7.5 Enkät**

Syftet med enkäten var att få reda på:

- Om utvecklarna är nöjda med dagens utvecklingsprocess?
- Vilken del av processen de är missnöjda med?
- Om utvecklarna tycker att de testar sin kod tillräckligt?
- Hur utvecklarna ställer sig till TDD?
- Hur utvecklarna ställer sig till par programmering?
- Förväntningar på det automatiserade unit testet?
- Om utvecklarna känner sig nöjda med sin testning när de är klara?

---

Enkät hypoteser:

Att utvecklarna:

... som arbetat en längre tid med mjukvaruutveckling kommer att hysa större motstånd mot nya utvecklingsmetodiker.

... hysar en del missnöje mot *unit* test processen.

... har låga/felaktiga förväntningar på det automatiserade *unit* testet.

... ofta hoppar över testningen på grund av tidsbrist.

... känner att de inte hinner testa tillräckligt mycket.

... nästan inte utför någon regressionstestning.

### 7.5.1 Enkät resultat och diskussion

Denna del av rapporten kommer att redovisa enkätresultaten. Jag kommer här att presentera resultaten lite kort i textform, vill läsaren ha mer utförlig statistik samt diagram så får han se appendix II.

Första frågan i enkäten visar att det finns ett utbrett missnöje med dagens *unit* test process bland utvecklarna, över 54 % tycker den är dålig eller mindre bra. Enbart 9,7 % tycker att den är bra.

Vad som kan förbättras i utvecklingsprocessen tror utvecklarna att en tydligare process samt en tydligare *unit* test process skulle vara till mest nytta. Mer design hade lite mer blandade svar där det inte gick att utläsa något tydligt resultat.

I en av hypoteserna var att utvecklaren kände att han inte hade tillräckligt med tid för att testa sin kod. Detta visade sig även stämma då 67,7 % av deltagarna tyckte att de inte hann testa sin kod tillräckligt. Även i frågan om hur ofta testningen blev påverkad av tidsbrist i slutet av ett projekt så svarade över 58 % att det inträffade i de flesta projekten eller alla projekt.

Frågorna som handlade om att de följer tidplanen för *unit* testing gav blandade svar. Min hypotes innan jag gick ut med enkäten var att de planerade mer tid för *unit* testning än vad som sedan utfördes. Men resultaten visade att den planerade tiden var i nästan alla fall den samma som den egentliga tiden. Det fanns t.o.m. de som lade ner mer tid än vad de planerat. Kanske man borde ha omformulerat frågorna så att följdfrågan blev hur mycket de testade jämfört med den planerade tiden.

En av de viktigaste frågorna i enkäten var den om regressionstestning, 83 % skrev att de testade mindre än 25 % av komponenten som de rättat en defekt i. Detta bekräftar en av hypoteserna som jag ställde innan enkäten utfördes om att regressionstestningen knappt existerade.

Vad utvecklarna tror påverkar kodkvaliteten mest är följande:

(siffran i parentesen bakom alternativet är en omräkning av betyget utvecklarna satt, där 1 = "small", 5 = "much")

Kod granskning (4.10)

Design fas (3.81)

---

Förstudie (3.26)

*Unit* test specifikation (3.19)

Överlag så tror utvecklarna att de fyra förslagen påverkar rätt så mycket. Kod granskning är den del som de tror påverkar mest vilket man skulle kunna tolka som att de tycker den är effektiv idag. Men även de andra har fått överbetyg.

Resultatet från förväntningarna på ett automatiskt test är följande. Att enbart ca 20 % tror att det kommer bli mer jobb för utvecklaren. Denna siffra är mycket bra då en av hypoteserna var att utvecklarna hade felaktiga förväntningar på ett automatiskt *unit* test. Även att över 60 % både tror att det kommer bli högre kodkvalité samt att det är tidsbesparande. Dessa svar är också bra då en av förutsättningarna för att automatiskt *unit* test skall lyckas är att utvecklarna tror på idén. Men självklart gäller det även att övertyga resterande 40 % som inte är positiva.

Uppskattningen av automatiskt *unit* test låg också över förväntan. Särskilt fördelen med att enklare kunna regressionstesta där 75 % av utvecklarna tyckte att det var bra eller mycket bra. Att defekterna hittades tidigare uppskattades mycket eller mest av 58 %. Högre kodkvalité uppskattades av 68 %. Hjälp vid *refactoring* uppskattades av 59 %.

Presentation av resultatet vill de flesta (42 %) att det skulle presenteras på en hemsida endast 7 % ville ha mail medan resten ville ha både och. Dessa resultat kommer att stödja idén om att det skulle komma två mail per dag så skulle man till slut sluta att läsa igenom dem. Men fortfarande skall mail skickas om ett allvarligt fel skulle uppstå.

Andelen som ville ha statistik på hur hög kodkvalité de hade jämfört med andra var jämnt fördelat mellan ja och nej medan 50 % av osäkra.

Testverktygen var som väntat att Lint scan och Leave scan används av nästan alla medan TEFUnit och Doxygen knappt användes. Gick även att utläsa att över 35 % inte använde Doxygen och inte heller ville börja. TEFUnit användes redan av 7 % och 65 % ville börja använda det.

Över 90 % av utvecklarna har inte använt sig av TDD. Samtidigt som 75 % av dem var intresserade av att prova på.

50 % hade provat på par programmering innan. Endast 10 % var emot att använda sig av par programmering vilket talade emot en av mina hypoteser innan där jag trodde att det skulle finnas ett motstånd till att prova på nya metodiker. I frågan om hur de ville jobba om de själva fick välja så var det en liten övervikt som ville sitta ensamma.

## 7.5.2 Samband

I denna del kommer jag att presentera olika intressanta samband och några av hypoteserna som jag ställde i början av rapporten.

I tabellerna nedan så har jag kört två frågor mot varandra för att se om det finns några tydliga samband mellan hur de svarat. Överst kan man utläsa den ena frågan och till vänster kan man utläsa frågan jag jämförde med.

		How long time have you been working at UIQ?				Total
		0-1 år	1-2 år	2-4 år	<4 år	
Are you prepared to try pair-programming if there were some course in how to use it?	Yes	3	6	6	4	19
	No	0	2	1	0	3
	I dont know	1	1	3	3	8
	Total	4	9	10	7	30

Tabell 1: Samband

		How long time have you been working at UIQ?				Total
		0-1 år	1-2 år	2-4 år	<4 år	
Are you prepared to try TDD if there were some courses in how to use it?	Yes	3	8	6	6	23
	I dont know	1	2	4	1	8
	Total	4	10	10	7	31

Tabell 2: Samband

I tabell 1 och tabell 2 så kan man utläsa att det inte fanns något tydligt samband mellan hur länge de har jobbat på UIQ och hur de ställer sig till par programmering och TDD.

		Would you like to have statistics at your code quality compared with other developers?			Total
		Yes	Maybe	No	
Do you feel that you are testing your code sufficiently?	Yes	3	5	2	10
	No	4	11	6	21
	Total	7	16	8	31

Tabell 3: Samband

Ett intressant samband som man kan utläsa från tabell 3 är att de utvecklare som inte tycker att de hinner testa sin kod ordentligt (fråga 3 i enkäten) inte heller vill ha någon statistik på hur väl deras kod står sig mot de andras (fråga 12 i enkäten).

		How long time have you been working at UIQ?				Total
		0-1 år	1-2 år	2-4 år	<4 år	
Do you feel that you are testing your code sufficiently?	Yes	2	2	2	4	10
	No	2	8	8	3	21
	Total	4	10	10	7	31

Tabell 4: Samband

Undersökte om det fanns några samband mellan hur länge de jobbat på företaget och om de var nöjda med sin testning. Men svaren blev rätt så utspridda, förutom att de som jobbat 1- 4 år klagade på att de inte hann testa sin kod.

		How often does your testing get affected at the end of a project due to lack of time?				Total
		No project	Some project	Most of the project	All projects	
Do you feel that you are testing your code sufficiently?	Yes	2	4	3	1	10
	No	0	5	9	5	19
	Total	2	9	12	6	29

Tabell 5: Samband

För att få ett litet mått på hur tillförlitliga svaren är så undersökte jag sambandet mellan frågan om de testade ordentligt samt frågan om hur ofta testningen blev påverkad av tidsbrist. Svaret blev lite oroande då åtta av tio sade att några projekt, de flesta eller alla projekt blev påverkade samtidigt som de svarade att de kände att de testade tillräckligt.

		Are you prepared to try pair-programming if there were some course in how to use it?			Total
		Yes	No	I dont know	
Are you prepared to try TDD if there were some courses in how to use it?	Yes	15	2	5	22
	I dont know	4	1	3	8
	Total	19	3	8	30

Tabell 6: Samband

Finns ett starkt samband mellan vilka som vill testa på TDD som till vilka som vill prova på par programmering.

		How much of your total development time (design + code + testing) do you plan for unit testing?			Total
		0-10 %	10-20 %	20-30 %	
How much of your total development time (design + code + testing) do you use for unit testing?	0-10 %	9	5	1	15
	10-20 %	3	5	2	10
	20-30 %	1	1	3	5
	Total	13	11	6	30

Tabell 7: Samband

Man kan utläsa från tabell 7 att 57 % av utvecklarna testar lika mycket som de planerat att göra medan 17 % planerar mindre tid än vad de lägger och 26 % lägger mindre tid än vad de hade planerat.

### 7.5.3 Kommentarer från enkäten

Här följer en samling kommentarer uppdelade efter vilken fråga de kom på. De flesta enkäterna innehöll inga kommentarer alls, vilket kan ha olika orsaker (läs mer i metod delen kap 2.2).

#### Fråga 16: If you did not want to try TDD, why?

“I don’t think the development time will be shorter. Maybe in a long term though quality is better. It seems hard to program in this way would therefore like to test it. OH sorry I did want to try it”

“I don’t need a course for that” (he wanted to try but he had not used TDD before)

“Can this be used in Application development?”

“Seems like an efficient way of rising code quality and of saving time (since only producing as much code as needed to pass the test case)”

“I am not sure if something like this work with such complex software”

“It is not how I think when I program and therefore does not come naturally for me.”

#### Fråga 19: If you did not want to try pair-programming, why?

“Don’t like the idea with someone sitting beside me and watch me coding. I will perform worse then”

“Preferre [sic!] to work alone when programming”

“I think having a sounding board raises the code quality, easies to find small bugs and come up with the best solution if you have someone to discuss concerns with”

“Too slow and not necessary with a bit of pair programming (for interaction with somebody else while coding)”

“Might be stressful and frustrating to do a whole day but an hour or two would work imho”

“Waste of time”

---

**Fråga 23: If you have some comments or questions about this questionnaire please write them here (if you need more space, use the other side)**

“Svårt att komma igång med automatiserade testning”

”Poor options on automated test results (question 11)”

#### **7.5.4 Felaktigheter i enkäten**

Man skulle ha haft med att de fick skriva på svenska på de öppna frågorna, detta då det kan ha funnits folk som inte skrev något p.g.a. de var dåliga på engelska.

- Råkade vända < - tecknen åt fel håll, men det ska inte ha påverkat resultaten då man inte kunde missuppfatta alternativen.
- Dålig svarsalternativ på fråga 7. En annan miss i enkäten var att det första alternativet var ”enbart komponenten” medan det andra var ”0 - 25 %” så vissa kryssade i båda rutorna, därför räknade jag ihop de två svaren till en klumpsumma då jag utgår från att ingen komponent motsvarar mer än 25 % av totala koden. Även att frågan var rätt så diffus då jag inte specificerade vad som ingick i ”komponenten” kan ha förvillat den utfrågade.
- Haft tydligare frågeställning på fråga 23 då de flesta svarade att de hade arbetat 0-2 år på ett annat företag. Men det resultatet låter osannolikt då många av de anställda utvecklarna jag pratat med är nyexaminerade.
- Att frågan om hur mycket statistik de ville ha var lite luddig så att utvecklarna kan ha varit osäkra på vad de svarade på därför den höga andelen med osäkra.

#### **7.6 Dokument analys**

Jag undersökte några *unit test approchar* (som låg i EP) för att se vilken kvalité de höll.

Några exempel på fel som jag upptäckte var följande:

”Memory allocations will be tested using leave scan, a tool that simulates memory allocation failure.”

“Since we lack a usable Unit Test tool (QTEF isn’t ready), testing at class level is not required.”

---

## Kapitel

# 8 Analys av testfall

---

### 8.1 Inledning

I början av examensarbetet så arbetade jag i en *unit* testgrupp på tre personer inklusive mig själv. Syftet med gruppen var att få igång ett testramverk kallat *Test Execution Framework* (TEF). Under min tid i gruppen så arbetade jag med att utveckla testexempel som utvecklarna skulle kunna använda för att få idéer om hur de skulle kunna testa efter minnesläckor och *heap failures* i sin kod.

Testfallen som jag har valt att analysera är hämtade från QTEF fas 2 kravspecifikationen. Informationen om dem har jag hämtat via samtal med anställda och erfarenheter från utvecklingen av exempel testfallen.

Det viktiga när man analyserar om testfallen är automatiserbara eller ej är inte enbart tidsåtgången att skriva testfallet som man skulle kunna tro, utan andra faktorer som står beskrivna i kapitel 3.2.3.

### 8.2 Minnesläckor

Minnesläckor är en defekt som uppstår när man inte frigör allt minne som man allokerat i sin funktion. Symbian OS har en inbyggd funktion för att upptäcka minnesläckor, den fungerar så att man har ett makro som gör en markering för vilken den aktuella nivån i stacken är. Sen så lägger man till ytterligare ett makro i slutet av sin kod därefter jämförs de två olika nivåerna och om de två nivåerna skulle skilja sig åt så kommer programmet att avbrytas.

Denna metod är mycket lätt använd och kommer inte att ställa till några större problem för utvecklaren att testa sin kod med. Testet täcker även upp andra möjliga minnesläckor problem om man bara använder det på rätt sätt. Minnesläcks testning är en typ av testfall där om man har gjort det en gång så går det snabbt att göra om det.

Automatiserbart: Ja

### 8.3 Heap failure

*Heap failure* är när du får problem med din minnesallokering på stacken. Ett exempel kan vara när man får ett *out of memory* (OOM) fel när man vill skapa en länkad lista och minnet tar slut efter att du enbart har lagt in en halv post. Då kan problemet med att ena delen av den posten inte rensas bort ordentligt när man skall allokera av minnet och då uppstår en minnesläcka.

---

*Heap failure* är också lätt att testa efter om man kan hantera UHEAP makrona. I dessa testfall så testas man också efter en slags minnesläcka, men i detta fall så skapar man OOM fel för att se att funktionen kan hantera att rensa upp efter delvis allokerade objekt. Detta testfall är också lätt att testa samt att återanvända.

Automatiserbart: Ja

## **8.4 Funktionalitetstestning**

Funktionalitetstesting är unikt för varje komponent man ska testa. I det testexempel som vi använder testas vi bland annat att lägga till och ta bort element på en länkad lista för att ge ett enkelt exempel. Dessa test kommer nog att skilja sig mycket åt i svårighetsgrad att testa. Man kommer nog inte heller att kunna återanvända så mycket av testen då det hela tiden är nya funktionaliteter man skall testa. Antalet funktionalitetskrav test som krävs för att täcka upp alla krav beror mycket på hur insatt utvecklaren är i uppgiften, detta kräver god förståelse i vad kravspecifikationen vill att man ska göra

Automatiserbart: Ja

## **8.5 Boundary**

*Boundary* testing är när man testas funktionsanrop med olika argument så som dess maximum, minimum, *out of range* och normala värden. Dessa test hänger delvis ihop med funktionalitetstestningen då de kommer att styra vilka värden som är godkända.

Automatiserbart: Ja

## **8.6 Out Of Disk**

*Out of disk* (OOD) testning är ett test då man testas hur programmet reagerar när minnet blir fullt. Dessa test passar sig bra för att automatisera då utvecklaren enkelt med förslagsvis en *while-loop* kan fylla upp disken för att se hur programmet reagerar.

Automatiserbart: Ja

## **8.7 File Access**

Testa att funktionen kan hantera om en fil saknas, är felaktig eller låst. Dessa saker kan ta ett tag att automatisera men när det väl är gjorda så kommer man lätt att kunna återanvända det.

Automatiserbart: Ja

---

## **8.8 Prestanda**

Prestanda test passar sig mycket bra för att automatisera. Då man redan innan var tvungen att göra ett slags skript för att testa prestandan. Ett problem som UIQ bör fundera på är vilket värde som dålig/bra prestanda har, men detta är ej ett problem som är knutet till automatiserade testning.

Automatiserbart: Ja

## **8.9 Server connection handling**

Detta test är till för att testa att mobilen inte har några problem med att kommunicera med servern. Tror att det kan vara lite svårare att automatisera detta test då man behöver starta upp en server att testa mot.

Automatiserbart: Nej

## **8.10 Response time**

Respons tiden är den tiden det tar för ett program att svara. Lite av samma problem som med prestanda, men om man väl vet vilken tid som är acceptabel så ska det inte vara några problem med att automatisera.

Automatiserbart: Ja

## **8.11 Localization**

Testar att ramverks komponenterna kan klara av längre strängar samt testar att trunkeringen fungerar.

Automatiserbart: Ja

## **8.12 Visual testing of graphical components**

Som namnet låter så behöver man titta på resultatet för att se om det stämmer. Detta kallas *flicker* testning och det menas att man tittar på displayen för att se hur ofta bilden ritas om. Om den gör det för ofta så går onödiga prestanda till det viket slöar ner systemet.

Automatiserbart: Nej

---

### **8.13 Redundant or unused code testing**

Undersöker koden efter oanvänd samt överflödig kod. Detta test blir svårt att göra automatiskt utan här behöver man göra det manuellt. Men det finns vissa program som ser vilken kod som körs när man provkör systemet men för att kunna använda dem så måste man veta att man verkligen provkör absolut alla kombinationer av testfall som finns.

Automatiserbart: Nej

### **8.14 In source documentation of source code**

Testa så att kommentarerna i koden är rätt så att de kan omvandlas till en korrekt SDK hjälptext. Till detta används idag verktyget Doxygen, vilket man antagligen kan automatisera.

Automatiserbart: Ja

### **8.15 Compile testing of source code**

Att det inte blir några varningar eller fel när man kompilerar. Detta kontrolleras enklast med Lint scan verktyget.

Automatiserbart: Ja

---

## Kapitel

# 9 Kostnad/vinst beräkningar på ett AUT

---

### 9.1 Inledning

Det är svårt att sätta några siffror på hur stora vinsterna eller kostnaderna kommer att bli om man inför ett automatiserat test. Även om slutsatsen blir att det kommer att löna sig så skall man inte räkna med att det kommer att betala sig förrän tidigast efter nästa projekt eller senare [21].

### 9.2 Kostnader

Testning är en process som enbart kostar resurser men inte genererar några inkomster. Därför är det inte konstigt att organisationen väljer att spara på testningen när de vill höja slutvinsten. Men utan ett ordentligt test som kan bekräfta kvalitén på koden så kommer det bli svårt att sälja produkten till kunden. Hoffman identifierade följande punkter som kostnader för ett manuellt test: [21]

- Träning av personal
- Produktanalys och inläring
- Design av testfall
- Test exekveringen
- Analys av testresultat
- Test *management*

Hur är det möjligt att minska kostnaden för dessa aktiviteter? Ett förslag skulle kunna vara att ta bort en av ovan nämnda. Men tar man bort en av design av testfall, test exekvering eller analys av test resultatet skulle de göra att övriga skulle bli värdelösa. Om man däremot skulle ta bort produktanalys, träning av personal eller test *management* så skulle det fungera men man skulle inte veta vad man skulle göra eller hur man skulle göra det. Skulle man däremot minska alla lite så skulle det fungera men kodkvalitén skulle försämrats. [21]

Ahmed och Lindhe tar upp följande punkter som kostnader för ett automatiskt test: [4]

- Design av testfall anpassade för automatisering
- Implementering av testfall
- Design och implementering av en automatiserad miljö
- Underhåll av testverktyg och miljö
- Verktygslicenser och support för automatiseringen
- Introducera och träna utvecklarna i att använda det automatiserade testet
- Automatiserade verktygs operationer
- Skapande av Oracle
- Analys av testprocessen plus eventuella ändringar

- 
- Nya uppgifter framtvungade av testautomatiseringen
  - Identifikation av testfalls kandidater för automatisering

Vissa av kostnaderna är kopplade till introduktionen av det automatiska testet medan de andra är löpande utgifter. Kostnaderna som är knutna till introduktionen är näst intill fasta medan kostnaderna för underhåll och övrigt löper över tiden. Underhåll av det automatiska testet är en kostnad som är helt ny jämfört med manuella test. I underhållet kan man räkna in saker som analys av data och ytterligare utveckling av det automatiska testet. Blir även extra arbete om man gör en kravändring, då man även skulle behöva ändra de tillhörande automatiska testerna också.

Att försöka jämföra de olika kostnaderna för automatisk testning jämfört med manuell är svårt. Det finns olika bud om det är möjligt att göra kostnads jämförelse mellan de två. Bach [19] säger: "Hand testing and automated testing are really two different processes, rather than two different ways to execute the same process". Med vilket han menar att det är meningslöst att göra en direkt jämförelse mellan de två.

Hendrickson tror också att det är svårt att mäta vinsterna av test automatisering i pengar precis som det är svårt att mäta de extra kostnaderna för automatiseringen. Hon säger bland annat: "Did the automated tests suddenly become twice as valuable because I ran them 10 times instead of 5?" [20]

Vilket är ett intressant resultat då värdet för att ett test körs mer än en gång är svårt att räkna fram, vilket belyser en av värdering frågorna. Hoffman däremot påstår att man kan beräkna kostnaden av test automatiseringen och den manuella testningen med en framtagen *return of investment* formel. Denna kan läsas mer om i kapitel 9.4.

### **9.3 Vinster**

En grundläggande förutsättning för att ett testfall skall vara lönsamt att automatisera är att det kommer att köras mer än en gång [6]. Så en av de viktigare vinst frågorna är hur ofta man har behovet av att köra ett testfall, skulle man t.ex. bara köra det en gång så är en automatisering ej lönsam. Men de flesta testfallen kan och behövs köras oftare och när väl testfallet är skapat så kostar det inget att köra det igen.

Antalet gånger testfallen körs beror bland annat på vilken utvecklingsmetodik man använder sig av. Om man skulle använda sig av TDD så kör man antagligen testen flera gånger jämfört med en normal utvecklingsmetodik eftersom TDD metodiken bygger på upprepande test. Även med en normal utvecklingsmetodik så har du nytta av att kunna återanvända dina testfall för att veta att din tidigare utvecklade funktionalitet fortfarande fungerar. Även vilken typ av testfall har betydelse, vissa är enklare än andra att automatisera och andra kanske inte passar alls för automatisering.

Även att de mest komplexa testfallen inte kommer att automatiseras då det skulle kosta mer än vad det smakar [6].

---

## 9.4 Regressionstestning

En annan viktig aspekt för kostnad/vinst beräkningen är hur stor del av defekterna som upptäcks i systemtestningen som är regressionsdefekter. Många av dessa defekter skulle kunna upptäckas direkt om man skulle ha haft ett automatiserat test som man kunde köra efter man rättat till en defekt för att hitta nya defekter. Medan om man skulle ha haft enbart manuella test så skulle man fått sitta och köra igenom alla testfall ytterligare en gång, vilket inte görs idag på UIQ enligt enkäten.

En siffra som figurerade är att för var tredje defekt som åtgärdas så skapas en regressionsdefekt. Dessa siffror skulle motsvara att var fjärde defekt är en regressionsdefekt (om man bortser från att var tredje regressionsdefekt möjligtvis skapar ytterligare en regressionsdefekt).

## 9.5 Räkneexempel

Som appendix till rapporten finns det två räkneexempel där de undersökt hur stora besparingarna man skulle få när man inför ett automatiserat test. Första exemplet är ett GUI exempel och det andra ett byggtest exempel. De båda exemplen använder sig av olika formler för att beräkna *return of invest* (ROI) värdet. Det andra exemplet använder sig av en formel där man inte behöver identifiera det exakta antalet eller kostnaden för att köra automatiserade eller manuella test utan istället använder uppskattningar baserade på personal och uppgifter som testgruppen använder. [20]

I första exemplet räknar de på ett projekt där de använder dagliga byggningar och test körningar (5 gånger/veckan). Där skulle de nå break-even (ROI = 1,013) efter sex månader och efter 18 månader skulle man få tillbaka 179 % (ROI = 1,786) av de satsade pengarna. Men dessa värden är mycket beroende på hur ofta man kör testen samt hur mycket underhåll de kräver.

Det andra exemplet räknar på automatisering av ett GUI test exempel vilket kanske är lite mer likt förutsättningarna på UIQ. Där räknade de på längre tid samtidigt som resultaten inte blev lika positiva som det andra exemplet. Efter 12 månader så var det t.o.m. en liten förlust (ROI = 0,874) medan efter 24 månader så fick man ett värde på ROI = 1,552 vilket motsvarar en vinst på 55 % på satsat kapital.

---

## Kapitel

# 10 Implementerings förslag

---

### 10.1 Inledning

I detta kapitel kommer jag att ge ett förslag på hur ett automatiskt *unit* test på UIQ skulle kunna se ut. Efter mina undersökningar på UIQ så har jag kommit fram till att det går bra att införa ett automatiserat *unit* test då deras produkt är relativt stabil och det finns behov av regressionstestning då utvecklarna idag inte regressionstestar ordentligt enligt enkäten som utfördes. Även tidigare undersökningar har visat på förhöjd kodkvalité.

### 10.2 Testets struktur

Det automatiska *unit* testet skulle kunna sammanlänkas av ett program (skript). Detta program kommer att exekvera de olika delarna så som TEFUnit, Lint scan, Leave scan, Doxygen och CodeTest så att de körs efter varandra eller eventuellt parallellt. När exekveringen är klar presenteras resultatet på önskat sätt.

### 10.3 När ska testet köras?

Förslagsvis så ska det finnas tre tillfällen då ett automatiskt *unit* test kan köras. Det första tillfället är varje gång en utvecklare integrerar sin kod mot *teambranch*. På detta sätt så slipper man i ett senare skede att få upp en kod innehållande onödiga defekter på *baseline* vilket skulle göra att byggningarna skulle gå mycket smidigare.

Andra gången det automatiska testet körs är varje gång någon integrerar mot *baseline*. Vid detta tillfälle så skall alla programmerings defekter som skulle kunna ha upptäckts av Lint scan, Leave scan och Doxygen vara borta. Detta skulle innebära att man inte skulle behöva köra testverktygen (Lint scan, Leave scan och Doxygen) varje gång utan enbart inrikta sig på TEFUnit testfallen för att öka snabbheten på AUT.

Tredje tillfället när AUT kan köras är manuella körningar som startas av utvecklaren själv, dessa tillfällen skulle kunna vara t.ex. vid regressionstestning eller efter rättning av en defekt. Kanske är missvisande att kalla det för automatiskt då man kör det manuellt men man kan se det som att testfallen körs automatiskt efter varandra.

### 10.4 Vad ska testet innehålla?

Basen i det automatiska *unit* testet kommer att vara TEFUnit där det mesta från funktionalitet till minnesläckor kommer att testas det vill säga alla testfall som utvecklaren skapar.

---

Lint scan och Leave scan som är testverktyg som testar koden är också passande att införa i ett AUT. Idag så skall dessa verktyg användas men det slarvas en hel del med det. Om man införde att dessa verktyg körs automatiskt så skulle det tvinga utvecklarna att åtgärda sådana fel direkt istället för att de väntar tills sista dagen och då får onödiga varningar.

Doxygen skulle passa mycket bra i ett automatiserat *unit* test, då verktyget letar efter fel/missade kommentarer i koden. Resultatet får man sedan tillbaka i statistik form som är lätt att redovisa för användaren. Idag så slarvas det en hel del med kommentarer vilket gör att det blir svårare för andra utvecklare att förstå kod vilket skulle förbättras om Doxygen kördes automatiskt.

För att få bättre uppfattning om hur mycket av koden som täcks av testfall kan man använda CodeTest som undersöker hur stor del av koden som exekveras när man kör sina testfall. Detta kan ge missvisande resultat då utvecklaren skulle kunna utveckla ”dummytestfall” som enbart kör igenom koden utan att testa den.

### **10.5 Hur ska resultatet från testet lagras?**

Att man skall lagra alla resultat från ett automatiskt test ser jag som en självklarhet då man kommer att kunna utläsa intressant statistik där ifrån. Antingen så lagrar man resultatet uppdelat efter komponenter eller efter testprogram (Lint scan, Leave scan, Doxygen eller TEFUnit). Det senare är nog lättare att implementera men det första ger intressantare information. Man skulle också kunna lagra informationen från testverktygen var för sig i en databas för att sedan hämta den informationen man är ute efter.

### **10.6 Presentation av resultatet**

Om man inte presenterade resultaten från testen så skulle det vara värdelöst. När vi diskuterade hur man skulle presentera dem så kom vi fram till att en hemsida samt e-mail skulle vara den bästa lösningen. Men det finns en stor risk att om man skickar e-mail efter varje körning så skulle utvecklaren till slut sluta bry sig. Därför skall e-mail endast skickas till utvecklaren vid följande tillfällen:

- Första körningen efter han lagt upp ny kod på *baseline* för att bekräfta att testet har körts och alla testfall eventuellt har passerat. Även för att se att utvecklarens kod inte har skapat defekter på någon annans kod.
- Om en regressionsdefekt skulle uppstå på hans komponent.

Ett problem som kan uppstå är att informationen inte kommer rätt, idag så kan man knyta varje *release* till en utvecklare. Men om han t.ex. slutar så måste man ha en ny ansvarig för koden. Samt ha ett system som följer upp att alla defekter antingen hamnar i TeamTrack eller blir omhändertagen av en utvecklare direkt.

---

Även *team* ledaren och projektledare skall få rätt information. Denna information kan bland annat användas för att se hur hög testfalls täckning utvecklaren har och hur de olika sektionerna står sig emot varandra i antal defekter/varningar samt hur hög testfalls täckning de har.

För att ledningen skall kunna bilda sig en uppfattning om det automatiska *unit* testet behövs även statistik som kan framhäva eventuella förbättringar på kodkvalitén. Det är därför viktigt att den som ansvarar för testet även sköter insamling, dokumentering och analys av data.

### **10.7 Utbildning och information**

Den informationen utvecklarna har fått hittills är hur TEFUnit fungerar och vad det gör medan de inte fått bakgrunden till varför *unit* testet infördes. Det som behövs för att få utvecklarna positiva till det automatiska *unit* testet är informationsmöten där de får helheten och dess fördelar förklarade. När de sedan har helheten och förstår syftet med det automatiska *unit* testet så kommer de själva vilja börja använda det. När man sedan har fått dem att börja använda det automatiska *unit* testet så kan man ha ytterligare informationsmöten där man nöter in detaljer så som vilka testfall som är viktigast att skriva, vilka testfall som är mest effektiva och hur man skall skriva testfallen för att de skall vara lätta att underhålla m.m.

### **10.8 Förväntningar**

Man ska inte förvänta sig att några stora kodkvalitets förbättringar kommer att märkas redan i de första projekten. Antagligen kommer det att kosta mer än vad det smakar nu i början, plus att kanske utvecklarna kommer kanske få en falsk trygghet när då de tror att det räcker att regressions testa med det automatiska *unit* testet redan nu.

Erfarenheten som utvecklarna får nu kommer att vara mycket värda i framtida projekt. Förhoppningvis kommer de flesta då också använda TDD vilket kommer att ge fler samt bättre testfall vilket kommer att medföra högre kodkvalité.

---

## Kapitel

# 11 Diskussion

---

En viktig definition som används i denna rapport är uttrycket *unit*. I kapitel 3.1.1 presenterar Lindhe och Ahmed [4] olika definitioner på vad en *unit* är men jag skulle vilja påstå att ingen av dem passar in på UIQ. Istället skulle jag vilja säga att en *unit* är: den kravspecifikation som utvecklaren får i uppgift att utveckla. *Unit* test blir alltså då den samling med test som man bestämt innan skall utföras på utvecklarens kravspecifikation.

Vad *unit* testningen skall täcka upp är en intressant fråga, personligen tror jag att det är en avvägning mellan att försöka automatisera allt och se hur mycket det kostar jämfört med tiden man sparar. Idag så finns det en omfattande *unit* test specifikation som antagligen täcker upp det mesta men problemet är att den inte följs på grund av att utvecklarna inte har tid eller vet hur man skall utföra alla testfallen.

I kapitel 3.1.2 så räknar Fewster [5] upp fyra attribut som påverkar kvalitén på ett testfall där ett av dem är: ”Ett bra test skall testa mer än en sak, där igenom reduceras det totala antalet testfall” men i kapitel 3.2.5 så påstår han att: ” att testen är så oberoende av varandra som möjligt”. Dessa två uttalande är lite motstridiga men jag anser att det värt att ha fler testfall istället för att klumpa ihop dem så att de blir beroende av varandra. Men de övriga punkterna anser jag viktiga för att hålla en hög kvalitet på testfallen vilket man vinner på i längden.

Tre fördelar med automatisk testning beskriver Hayes som följande: *repeatability*, *leverage* och *accumulation*. *Repeatability* anser jag att vara den viktigaste, för att höja kodkvalitén så behöver man upptäcka defekterna så tidigt som möjligt. När man använder sig av ett automatiserat test så kommer utvecklaren att köra igenom sina testfall oftare jämfört med att man testat manuellt då det skulle bli för tidskrävande. Detta kommer att medföra att defekterna hittas tidigare. Även att man kan köra prestanda test och dylikt som inte skulle ha gått att göra manuellt är positivt.

Att kodkvalitén skulle förbättras med automatiserade tester stöds av flertalet undersökningar. I undersökningen på Ericsson i Karlskrona så upptäcktes över 29 % färre defekter på systemtestnivå efter man använt sig av ett automatiserat test och TDD. Att detta sedan leder till höjd kodkvalité eller kortare utvecklingstid bör man kunna anta.

Även att kunna använda ett automatiskt *unit* test för att upptäcka regressionsdefekter är en fördel. Med det automatiska testet skulle man kunna testa sin kod efter att ha åtgärdat den. Detta skulle göra att regressionsdefekter skulle upptäckas direkt istället för långt senare då det också skulle bli dyrare att åtgärda defekten [15]. Detta är en fördel som kommer att påverka kodkvalitén positivt då det idag knappt utförs någon regressionstestning enligt enkäten som utfördes.

---

Utvecklarna hade realistiska förväntningar på fördelarna med ett automatiserat test. Detta är positivt då det betyder att man inte behöver gå ut lika mycket information till utvecklarna.

Skall man även skriva automatiserade testfall till gammal kod? Man skulle kunna spåra utsatta komponenter som innehåller extra mycket defekter för att sedan skriva automatiserade testfall enbart för dem. Förslagsvis skulle dessa utsatta komponenter tas fram genom att analysera gammal mätdata. Detta skulle kunna vara en bra investering för UIQ då de återanvänder mycket kod från gamla projekt.

Hayes räknar i kapitel 3.2.4 upp tillfällena då det inte är värt att automatisera, dessa kommer jag nu gå i genom för att visa att automatisering passar sig bra på UIQ. Hennes första varning var att vid instabila program så som väderleksapplikationer så är det svårt att automatisera, där har UIQ inget att oroa sig för då dess produkt är stabil. Nästa varning var om att personen som utvecklar applikationen har för lite erfarenhet, detta kan stämma in på UIQ då ibland nyanställda sätts på att skriva automatiserade testfall. Tredje varningen är att sätta tillfälligt anställda på att skapa testfall, detta stämmer också till viss del då det finns konsulter här på UIQ. Den sista varningen är om man ligger under tidspress så man inte hinner med den manuella testningen så skall man inte heller försöka sig på den automatiska, detta gäller nog för alla IT bolag då de ofta ligger under stark tidspress i slutet av projekten.

Underhållet är något man bör tänka på när man skriver testfallen, detta då det automatiserade *unit* testet kan komma att innehålla flera tusen testfall. Alla dessa testfall måste självklart underhållas om koden skulle ändras. Det viktigaste då är att man verkligen har haft en tanke bakom varje testfall och inte enbart skriver det för sakens skull.

Annars tycker jag att UIQ har lyckats att uppfylla flertalet av punkterna som Marick nämner i kapitel 3.2.5. Namnsättningen och spårbarheten är väl genomtänkta sen blir det upp till utvecklarna att skriva oberoende test, dokumentera testen samt göra alla testfallen relevanta.

Forskningen inom automatiserade tester stödjer hypotesen om att antalet defekter skulle minska om man införde ett automatiserat test. Men i de flesta undersökningar som gjordes så införde de TDD eller någon variant av TDD samtidigt som de införde det automatiska testet. Detta stödjer hypotesen om att utvecklingsmetodiken också skulle behöva bytas ut.

Följande program kom jag fram till att de skulle passa att ingå i ett automatiskt *unit* test:

- TEFUnit
- CodeTest
- Lint scan
- Leave scan
- Doxygen

---

Fördelen med att införa de tre sista testverktygen är att idag så används de enbart sporadiskt. Om man skulle införa dem i ett automatiskt test så skulle utvecklaren slippa sitta sista dagen och åtgärda Lint scan fel vilket även skulle höja kodkvalitén då de kommer att upptäcka defekterna tidigare. CodeTest skulle hjälpa till att få fram statistik på vilken kodkvalité som finns m.m.

Blueberry skulle kunna användas i ett automatiska test, men risken är stor att det kommer att kosta mer än vad det smakar. Då det skulle ta tid att lära sig Blueberry för utvecklaren samt att många utvecklare inte har så pass mycket kod att det är möjligt att göra UI test. Att sedan *product verification* också kommer att testa UI senare gör att det finns risk för dubbelarbete så det bästa är nog att utvecklaren koncentrerar sig på att utveckla funktionalitets testfall

Utvecklingsmetodik som används idag fungerar men ofta vid förseningar så är det testningen i slutet som prioriteras bort. Detta är ytterligare en orsak till att man bör överväga att byta utvecklingsmetodik. Skulle det ha varit en engångshändelse att testning blev lidande p.g.a. tidsbrist så skulle det inte ha gjort något. Men nu visar min enkät på att fallet inte är så utan att nästan alla projekt drabbas av förseningar. Lösningen till detta är inte att införa en ny utvecklingsmetodik utan detta kan enbart lösas med fler resurser. Men med t.ex. TDD så skulle man i alla fall ha all kod testad ordentligt innan man var tvungen att releasa den dock med mindre funktionalitet. Detta är på lång sikt bättre då man istället för att behöva sitta och rätta defekter ska kunna fortsätta utveckla funktionalitet. Enligt forskning som gjorts så har det även visat på kortare utvecklingstid för TDD vilket skulle kunna undvika förseningar.

Enligt undersökningar som gjorts där de jämfört TDD med andra utvecklingsmetodiker så visar ofta TDD på högre kodkvalité samt mer lättförstålig kod. Hur utvecklingstiden skulle påverkas finns det ingen entydig forskning på utan i vissa undersökningar har den ökat medan i andra har den minskat. Nackdelarna med TDD som jag beskrev i kapitel 6.1.2 ska jag försöka argumentera bort nedan:

- Skillnaden mellan de olika faserna suddas ut  
Tror inte att man behöver dela upp utvecklingen i olika faser som det ser ut nu, om man vill ha ett mått på hur långt man kommit kan man istället undersöka hur mycket funktionalitet som utvecklats.
- Designfasen innan saknas  
Fortfarande kommer det att finnas design som förklara koden i stora drag, det som kan försvinna är låg nivå designen. Men enligt undersökningar så kommer testfallen bli designen som förklarar koden.
- *Refactoring* och behov av erfarna utvecklare  
Att koden kan bli så komplex att det blir svårt att utföra *refactoring* hänger lite ihop med att ha erfarna utvecklare. Detta skulle kunna ställa till problem för UIQ då de har många oerfarna utvecklare som kanske inte kan hantera koden. Men detta problem skulle även ha uppstått med en mer traditionell utvecklingsmetodik.

Men fördelarna för TDD tycker jag väger upp nackdelarna. Man kan märka om man jämför fördelarna med automatiserad testning och TDD att de är nästan desamma. Detta

---

för att de är så pass beroende av varandra, ett automatiskt test utan TDD skulle bli ett automatiskt test utan testfall medan det motsatta skulle bli automatiska testfall som man inte utnyttjar.

Att införa XP tror jag skulle vara för extremt, det som talar mest emot det är att det passar sig bäst för mindre projekt (upp till 20 personer). Men en del i XP som jag tror man skulle kunna ha användning för hos UIQ är par programmeringen. Den programmeringsformen skulle vara bra att använda sig av i upplärningen av nyanställda. Kunskaperna från den mera erfarna programmeraren skulle överföras snabbt samtidigt som den nyanställda har nya kunskaper som den erfarna kan ha nytta av.

I intervju fas två så var det utvecklare som uttryckte att de tyckte tiden för testning var för kort. Detta kan man nog tolka som att de tyckte den totala utvecklingstiden var för kort, då utvecklaren själv lägger fram förslag på hur lång tid varje del ska ta. Vad det beror på att utvecklaren inte planerar för mer tid för testningen kan nog bero på både oerfarenhet och att de inte vill framstå som dåliga utvecklare som behöver mycket tid på sig.

Ifrån intervju fas tre så kan man dra slutsatserna att en av cheferna har rätt förväntningar på det automatiserade testet. Ser det även mycket positivt på att han var positivt inställd till automatiserad testning, att ha stöd från högre instanser är nog ett måste för att testet skall lyckas och finnas kvar i framtiden.

När jag undersökte kvalitén på några *unit test approaches* så visade sig att vissa innehöll en hel del fel. De flesta felen var också rätt så lika vilket tyder på att utvecklarna inte tänker efter utan enbart kopierar en annan utan att tänka efter vad som skall testas. Detta är rätt så allvarligt då utvecklaren antagligen inte kommer få samma förståelse för koden vilket kan medföra att han producerar sämre kod.

Är det nu lönsamt att automatisera testningen? Min personliga uppfattning är att man inte kan räkna fram den teoretiska besparingen (om det nu blir en sådan) utan man får lita på tidigare undersökningar som visat att kodkvalitén har förbättrats efter man infört ett automatiserat test. Men jag tror att oberoende vilken metod för beräkning av lönsamheten man väljer så kommer det ta lång tid att få fram ett grovt uppskattat värde. Därför tror jag på att det bästa är att införa testet i mindre skala för att sedan bygga på det efterhand, sen får man göra regelbundna mätningar på kodkvalitén för att se om några förbättringar har uppstått.

Då de initiala kostnaderna redan är gjorda idag på UIQ där bland annat ett ramverk för testfallen (TEFUnit) och utvecklarna har fått en introduktion hur man skall använda det automatiserade testet så tycker jag inte att man skall lägga ner någon energi på att göra några kostnadsberäkningar. Utan att istället invänta de första resultaten.

En av fördelarna med ett automatiserat test är att de enkelt kan köras vid regressionstestning. Om man skulle kunna upptäcka regressionsdefekterna direkt efter man rättat en defekt så skulle det vara tidsbesparande istället för nu då *product verification* upptäcker den nya defekten. Regressionsdefekter skulle kunna upptäckas

---

redan av utvecklaren efter han rättat till sin kod istället för att koden skickas tillbaka till *product verification* och de eventuellt hittar defekten. I systemtestningen används inte samma sorts test som ingår i *unit* testningen utan de testar mer omfattande testning. Därför kan de missa defekter som skulle ha upptäckts på *unit* test nivå som skulle kunna orsaka defekter i framtiden.

I räknepapiret så hade jag med två exempel där det första visade på ett rätt så bra resultat medan de andra var lite sämre. Men man bör väga in att i det sista exemplet så var det när man utvecklat ett program från början medan på UIQ kommer man antagligen alltid kunna återanvända en stor del av testfallen vilket kommer att göra att *breakeven* antagligen kommer att uppnås tidigare.

Är nu automatiska test en ersättning av alla manuella test eller enbart ett komplement? Alla testfall går inte att automatisera och därför måste man fortsätta delvis med den manuella testningen. Därför är det viktigt att UIQ skapar en process för hur de även skall få med en beskrivning av hur de manuella testfallen skall utföras.

## **11.1 Validering av hypoteserna**

Nedan följer en uppföljning av hypoteserna jag ställde i början av rapporten och i enkät delen:

### **Dagens produktutvecklingsprocess släpper i genom onödiga defekter p.g.a.:**

#### **... dagens *unit* test process är otydlig och kan förbättras**

Enligt enkäten fråga ett så tycker utvecklarna att *unit* test processen fungerar mindre bra idag. Detta kan tolkas som att den är otydlig och skulle behövas ses över.

#### **... saknad av ett automatiserat test**

Enligt undersökningar som gjordes i kapitel 3.2.1 där de börjat använda automatiserade tester så visar de på att kodkvalitén har förbättras. Detta skulle man kunna tolka som att defekter som passerar *unit* testet idag som skulle kunna ha upptäckts av ett automatiserat test.

#### **... utvecklaren inte använder alla tillgängliga testverktyg**

Enkäterna har visat på att det finns brister i användningen av vissa testverktyg som skall användas. Lint scan (10 % använder inte), Leave scan (7 % använder ej) och Doxygen (36 % använder ej) är testverktyg som skall användas enligt *unit* test specifikationen men ej används regelbundet enligt enkäten. Antagligen så är även siffrorna i underkant, då när vi provkörde Lint scan på en komponent så upptäckte vi massor med defekter som inte borde ha uppstått.

### **Möjligheten finns att minska antalet defekter:**

#### **... genom att inför en ny utvecklingsmetodik**

Genom att införa TDD som ny utvecklingsmetodik så skulle antalet defekter minska enligt olika undersökningar som har gjorts. Även utvecklarna på UIQ är positivt inställda till att använda sig av TDD enligt enkäten vilket skulle underlätta ett byte av utvecklingsmetodiken.

---

### **... genom att upptäcka defekterna tidigare**

Ju tidigare man hittar en defekt desto billigare blir det att åtgärda den [15]. Billigare i detta fall är lika med kortare utvecklings tid vilket gör att man kan lägga mer tid på att hitta andra defekter vilket ger en kod innehållandes färre defekter.

### **... med ett automatiserat *unit* test samtidigt som kostnaderna är samma eller lägre**

Undersökningar visar att det inte behöver ta längre tid att skriva automatiserade test med TDD än vad ett automatiskt test skulle ta att skriva manuella test med dagens utvecklingsmetodik.

### **Att utvecklarna:**

#### **... som arbetat en längre tid med mjukvaruutveckling kommer att hysa större motstånd mot nya utvecklingsmetodiker**

Det visade sig att det inte fanns några större motstånd bland de utvecklare som jobbat en längre tid på företaget. Detta kan bero på att många på företaget inte arbetat inom IT branschen så länge. Annars tror jag fortfarande att en utvecklare som suttit och programmerat hela sitt liv på ett visst sätt inte kommer att vilja byta utvecklingsmetodik.

#### **... hyser en del missnöje mot *unit* test processen**

På de flesta företag så brukar det finnas ett missnöje hur saker och ting skall utföras. Speciellt på IT företag kan jag tänka mig att det ändå är mer tydligt med alla förseningar och övertid som det för med sig. Hypotesen visade sig stämma då över hälften tyckte dagens *unit* test process fungerade dåligt eller mindre bra. Vilket skulle kunna vara positivt på det sättet att de kanske blir mer öppna för förändringar.

#### **... har låga/felaktiga förväntningar på AUT**

Överlag så hade utvecklarna korrekta förväntningar på ett automatiskt *unit* test enligt de undersökningar som finns. Att utvecklaren skulle få mer att göra däremot trodde de flesta, detta tror jag dock inte kommer att stämma. Då den extra tid som utvecklaren lägger på att skriva automatiserade testfall kommer han sedan spara in när han skall *unit* testa. Men genom att ge utvecklarna mycket information om fördelarna så kan man undvika att de får felaktiga förväntningar.

#### **... ofta hoppar över testningen på grund av tidsbrist**

I kapitel 7.3.2 så har jag undersökt sambandet mellan hur mycket de planerar för test och hur mycket tid de lägger på test, 57 % svarade att de lägger lika mycket tid som de har planerat medan 26 % lägger mindre tid än planerat och 17 % mer tid än planerat. Dessa resultat var lite överraskande då jag trodde att fler skulle ha svarat att de lade mindre tid än de planerat för. Tror att det kan bero på att frågan var lite otydlig eller så är det så att utvecklarna har så pass mycket erfarenhet av *unit* testing så att de har lärt sig att planera rätt. En idé till varför 23 % svarat fel kan vara att de är nya och då vill man inte verka dålig och ger därför en felaktig prognos till sin *teamleader*.

#### **... känner att de inte hinner testa tillräckligt mycket**

Över 67 % av utvecklarna inte hinner testa sin kod tillräckligt, denna siffra vore intressant att undersöka efter man använt sig av ett automatiserat test en tid för att se om de känner sig mer nöjda med sin testning.

---

**... nästan inte utför någon regressionstestning**

Detta visade sig stämma då över 80 % testade mindre än 25 % av sin kod. Det är detta som ett automatiserat test kommer att förbättra, då utvecklarna kommer enkelt och snabbt att kunna regressionstesta sin kod efter att åtgärdat en defekt.

---

## Kapitel

# 12 Sammanfattning

---

Det UIQ ville få ut av detta examensarbete var hur/om de skulle kunna öka kodkvalitén med hjälp av ett automatiserat *unit* test. Rapporten kommer att visa att det går att införa ett automatiserat test och att man kommer att få höjd kodkvalité samt andra fördelar så som tydligare kod m.m.

Följande punkter har jag kommit fram till att de skulle behöva ändras för att införa ett automatiskt test och höja kodkvalitén:

- Införa en variant av TDD som utvecklingsmetodik.
- Införa en enmanna tjänst som underhåller samt bearbetar information från det automatiska testet.
- Att utvecklarna inte skriver sin *unit test approach* i samma dokument som *execution proposal*.
- Vid upplärning av nyanställda, använd par programmering.

Att införa en slags variant av TDD som utvecklingsmetodik skulle innebära många fördelar jämfört med dagens utvecklingsmetodik. Den största fördelen är den att utvecklaren tvingas skriva användbara automatiserade testfall och utan dem kommer hela det automatiska testet bli värdelöst. Ytterligare en fördel är att utvecklaren alltid kommer att ha en fungerande kod som han kan releasa. Det finns även undersökningar som visar att om man använder sig av TDD så höjer man kodkvalitén.

Om man vill få ut så mycket som möjligt av sitt automatiserade *unit* test så rekommenderar jag att man har en tjänst som analyserar statistiken, underhåller testfall och lär upp nyanställda hur de skall använda testet. Om man inte inför denna tjänst så jag blir det ett halvhjärtat försök vilket kommer att leda till ett misslyckande som enbart har kostat företaget resurser och pengar. Förslagsvis skall tjänsten ligga under *quality assurance* och inte *product verification*.

Det kan vara så att utvecklarna inte lägger ner nog med tid på dagens *unit test approach*. Då de redan här skriver felaktigheter så är risken stor att de även testar fel eller inte vet hur de skall testa. Detta problem kommer inte att lösas med ett automatiskt test men man kan kanske lösa det med mera information samt separera *unit test approachen* från *execution proposal*. När utvecklaren lägger ner mer tid på att komma på vad som skall testas innan de startar att koda så kommer de även få bättre förståelse för sin uppgift.

Undersökningar visar att man kan lära upp nyanställda mycket snabbare genom att låta dem par programmera med en mer erfaren utvecklare. Även den erfarna utvecklaren kan säkert lära sig lite från den nya då han måste förklara det han gör vilket kommer att tvinga honom att tänka efter vad han egentligen gör.

---

Följande fördelar har ett automatiskt test:

- Defekterna upptäcks tidigare.
- Går mycket snabbare/enklare att regressionstesta.
- Testverktygen kommer att utnyttjas bättre (kanske går att lösa utan ett automatiskt test).

Med ett automatiskt test kan man minska antalet defekter som upptäcks av systemtestningen så de kan lägga ner mer tid på att hitta mer svår hittade defekter vilket de annars inte skulle ha haft tid med.

Hur man ska få utvecklarna att skriva så många testfall som möjligt:

- Statistik
- Test driven Development
- Information

Undersökningar har visat att statistik är en bra morot för att få folk att göra saker bra/bättre. Om man hela tiden redovisar hur stor täckning testfallen har samt hur många testfall utvecklarna har utvecklat jämfört med andra så tror jag det kommer att skrivas fler testfall.

För att utvecklarna skall få helheten med ett automatiserat test så behövs information där man inte enbart presenterar hur ett automatiserat test fungerar utan även fördelar m.m. Det kommer också att behövas information om vilka testfall som är mest effektiva att automatisera m.m. för att utvecklaren skall skriva mer relevanta testfall.

---

## Kapitel

# 13 Framtida examensarbeten

---

- Analysera statistiken från det automatiska *unit* testet, för att se om det blev några kodkvalitets förbättringar
- Undersöka olika sorters testfall för att se vilka som är mest effektiva att automatisera

---

## Kapitel

# 14 Definitioner och förkortningar

---

### 14.1 Definitioner

Vanligt förekommande uttryck som kan vara svåra för en ej insatt att förstå.

*Automatiserad* – I namnet på denna rapport används uttrycket automatiskt, i detta fall så menas att utvecklaren inte skall behöva göra något utan att testet t.ex. körs automatiskt varje natt.

*Baseline* – Den senast sammansatta fungerande koden.

*Defekt* – När ett program inte fungerar tillförställande så kallas det som gör att det inte fungerar för defekt.

*Enhandsfattning* – Att man kan hantera mobiltelefonens menyer med enbart en hand.

*Hård kodad* – Att man skriver in t.ex. ett värde på flertalet ställen i koden som eventuellt kan ändras i framtiden. Detta kan undvikas genom att definiera värdet i början och sedan använda variabeln i koden.

*Integration* – När man integrerar en funktion eller liknande i ett projekt.

*Leave* – När en funktion returnerar ett värde.

*Milstolpe* – Översättning av engelskans *milestone*, är ett delmål i ett projekt.

*Personal branch* – Utvecklarens personliga katalog där han lägger sin kod.

*Plattform* - Vad en plattform är till en mobiltelefon kan man jämföra med vad ett operativsystem är till en dator. Men med den stora skillnaden att en mobil plattform får använda en bråkdel av resurserna som en dator har tillgång till. Så som t.ex. minne och processor. Detta ställer mycket höga krav på utvecklaren.

*Process* – Tillvägagångssätt för att lösa en uppgift.

*Refactoring* – När man ändrar om i koden så att den blir enklare att förstå. Man lägger alltså inte till någon ny funktionalitet eller rättar defekter.

*Release* - När man lägger upp sin kod på en gemensam baseline.

*Smartphone* – Mobiltelefon som innehåller vissa datorfunktioner men inte är en bärbar dator. Diffus gräns vad som vad.

*SPSS* – Statistik program som användes för att sammanställa enkäten.

*TeamTrack* – Här lägger man upp defekter som skall åtgärdas, man kan även följa vem som skall åtgärda samt vem som lade dit den m.m.

*Testfall* – Något som man testar.

*Testsvit* – En samling med testfall som hör ihop.

*Work package* – Vid större projekt delar man upp det i flertalet arbetspaket (work package).

*Unit* – När man delar upp ett projekt/program i mindre delar för att underlätta utvecklingen så kallas de nya delarna för enheter (eng. *unit*)

*Unit test* – En samling av testfall som skall utföras när man är klar med sin del av koden och skall releasa den.

---

## **14.2 Förkortningar**

- AUT - Automatiskt *Unit* Test
- GUI - Graphical User Interface
- OS - Operativ System
- SDK - Service Development Kit
- TDD - Test Driven Development
- XP - eXtreme Programming

---

## Kapitel

# 15 Referenser

---

- [1] Trost J., Enkätboken
- [2] “An introduction to Software Testing”, paper, IPL information Processing Ltd, 2002
- [3] Watkins J., “Testing IT”, ISBN 0-521-79546-X (paperback)
- [4] Ahmed A., Lindhe M., “Efficient and Maintainable Test Automation”, Master thesis, March 2002
- [5] Fewster M., Graham D., “Software Test Automation”, ISBN 0-201-33140-3
- [6] Kerry, ”A Perspective”, web-article, <http://www.testingstuff.com/autotest.html>, 2005-01-24
- [7] Hayes, L.G., “Automated testing handbook”, 1995
- [8] George B., Williams L., “A structured experiment of test-driven development”, paper, 2003
- [9] Boehmer B., Patterson B.,”Software Test Automation – Developing an Infrastructure Designed for Success”, paper
- [10] Marick B., “When should a Test Be Automated?” paper, 1998
- [11] Zambelich, K.,”Totally Data-Driven Automated Testing”, white paper, 1998
- [12] McBreen P., “Questioning extreme programming”, The XP series, 2002
- [13] Williams L., Maximilien E. M., Vouk M., “Test –Driven Development as a Defect-Reduction Practice”, IEEE-article, North Carolina State University, 2003
- [14] Müller M., Hagner O.,”Experiment about test-first programming”, IEEE-article, 2002
- [15] Damm L., Lundberg L., Olsson D.,”Introducing Test Automation and Test-Driven Development: An Experience Report”, article, 2004
- [16] Anthes G., “By using extreme programming practices, Sabre Airline Solutions has reduced bugs and development times for its software products”, ELIN database article
- [17] [www.extremeprogramming.org/what.html](http://www.extremeprogramming.org/what.html), 2005-01-24
- [18] Müller M., “Are Reviews an alternative to Pair Programming?”, article, 2004
- [19] Bach, J., “Cost Benefits Analysis of Test Automation”, paper, 1999
- [20] Hendrickson, E., ”Bang for the Buck Test Automation”, paper, 2001
- [21] Hoffman D., “Cost Benefits Analysis of Test Automation”, white paper, 1999
- [22] Williams L., “Pair programming Experiment”, website, 1999, <http://www.extremeprogramming.org/stories/pair6.html>
- [23] Damm L., Lundberg L., “Results from Introducing Component-Level Test Automation and Test-Driven Development”, BLEKINGE INSTITUTE OF TECHNOLOGY LICENTIATE SERIES ISSN 1650-2140, ISBN: 91-729-056-0

- 
- [24] [www.agilealliance.org/programs/roadmaps/Roadmap/modeling/modeling\\_index.htm](http://www.agilealliance.org/programs/roadmaps/Roadmap/modeling/modeling_index.htm), 2005-02-11
- [25] Ambler, S., <http://www.agilemodeling.com/principles.htm>, 2005-02-11

---

# Innehållsförteckning – Appendix

<b>INNEHÅLLSFÖRTECKNING – APPENDIX .....</b>	<b>56</b>
<b>APPENDIX I – ENKÄTEN .....</b>	<b>57</b>
<i>Test Driven Development</i> .....	60
<i>Pair-programming</i> .....	60
<i>Background questions</i> .....	61
<b>APPENDIX II – ENKÄT RESULTAT .....</b>	<b>62</b>
<b>APPENDIX III – FIGURER .....</b>	<b>84</b>
FIGUR 1: SOFTWARE DEVELOPMENT.....	84
FIGUR 2: UIQ D-MODEL .....	85
<b>APPENDIX IV – INTERVJUFRÅGOR FAS ETT .....</b>	<b>86</b>
MINNESLÄCKOR .....	86
HEAPFAILURE.....	86
<b>APPENDIX V – INTERVJUFRÅGOR FAS TVÅ .....</b>	<b>87</b>
UNDERSÖKNING AV DEN NUVARANDE PROCESSEN .....	87
<i>Unit testing</i> .....	87
<i>Release frågor</i> .....	87
<i>Regression testing</i> .....	87
<i>Övrigt</i> .....	87
<b>APPENDIX VI – INTERVJUFRÅGOR FAS TRE.....</b>	<b>88</b>
INTERVJUFRÅGOR OM AUTOMATISERAD TESTNING .....	88
<i>Blandade frågor</i> .....	88
<i>Förväntningar</i> .....	88
<b>APPENDIX VII – RÄKNEEXEMPEL .....</b>	<b>90</b>
<i>Automation of build tests examples:</i> .....	90
<i>Automation of GUI tests example:</i> .....	91

---

## APPENDIX I – Enkäten

---



### Automated unit testing

I am writing a thesis about how to implement an automated unit test process in the development organization. The purpose of this investigation is that the result will provide and create a base for future proposals and improvements.

The investigation is totally anonymous. After I have collected the questionnaires I am going to put them together in statistical tables and your individual answers will not in anyway be traced.

This questionnaire will be distributed to all developers at UIQ.

If you have any questions or feedback, do not hesitate to contact me.

In advance, thank you for your participation!

Ronneby in November 2004-11-22

*Daniel Sandberg*

Master of Science in Electrical Engineering student  
Blekinge Institute of Technology  
+46702102695  
daniel.sandberg@uiq.com



---

8. How do you think that the following parts affect the code quality?

(1-None, 5-Much)

	1	2	3	4	5
Code review	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Unit test specification	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Design phase	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pre-study phase	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

---

9. What are your expectations for an automated unit testing process?

(1 – small, 5 – big)

	1	2	3	4	5
Higher code quality	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Time saving	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
More work for the developer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

---

10. How much do you appreciate the advantages with an automated unit test process?

(1- lowest, 5 – highest)

	1	2	3	4	5
More confidence when you are going to regression testing	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Finding the defects earlier	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Higher code quality	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
More confidence when refactoring (rebuild the code without changing the functionality)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

---

11. How do you want to have the result from an automated unit test presented?

E-mail	<input type="checkbox"/>
Website	<input type="checkbox"/>
Both	<input type="checkbox"/>

---

12. Would you like to have statistics at your code quality compared with other developers?

Yes	<input type="checkbox"/>
Maybe	<input type="checkbox"/>
No	<input type="checkbox"/>

---

13. Do you use these test tools?

	Yes	No	I want to start
Lint scan	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Leave scan	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
TEFUnit	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Doxygen	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

---



---

18. Are you prepared to try pair-programming if there were some courses in how to use it?

Yes

No

I don't know

---

19. If you did not want to try pair-programming, why?

.....

.....

.....

---

20. What would you choose if you had a choice?

Single programming

Pair programming

I don't know

---

### Background questions

---

21. How long time have you been working at UIQ?

0-1 year

1-2 year

2-4 year

< 4 year

---

22. Have you been working at some other IT-related company, in that case for how long?

0-2 year

2-5 year

5-10 year

< 10 year

---

23. If you have some comments or questions about this questionnaire please write them here (if you need more space, use the other side)

.....

.....

.....

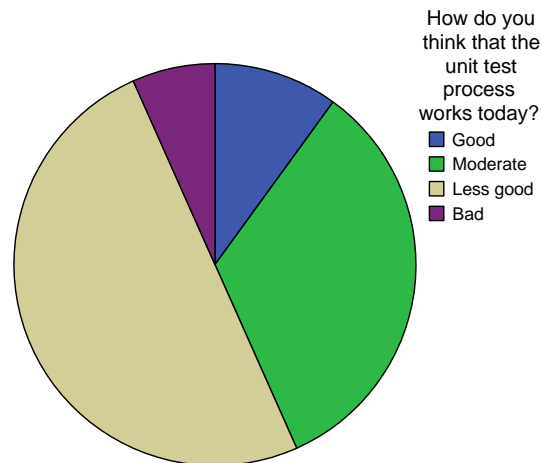
Once again, thank you for your help!

---

## APPENDIX II – Enkät resultat

Fråga 1: How do you think that the unit test process works today?

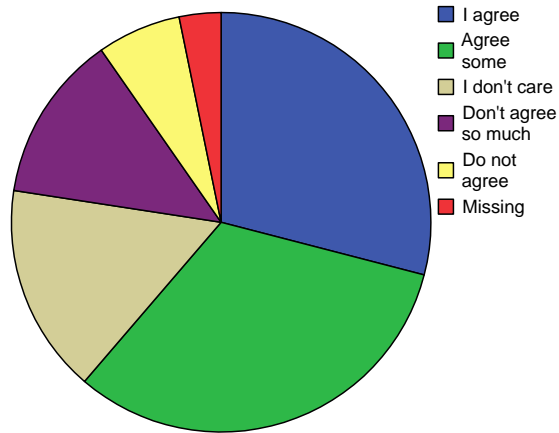
		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Good	3	9,7	10,0	10,0
	Moderate	10	32,3	33,3	43,3
	Less good	15	48,4	50,0	93,3
	Bad	2	6,5	6,7	100,0
	Total	30	96,8	100,0	
Missing	System	1	3,2		
	Total	31	100,0		



Fråga 2: What do you want to improve in todays development process? A more clear process

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	I agree	9	29,0	30,0	30,0
	Agree some	10	32,3	33,3	63,3
	I don't care	5	16,1	16,7	80,0
	Don't agree so much	4	12,9	13,3	93,3
	Do not agree	2	6,5	6,7	100,0
	Total	30	96,8	100,0	
Missing	System	1	3,2		
	Total	31	100,0		

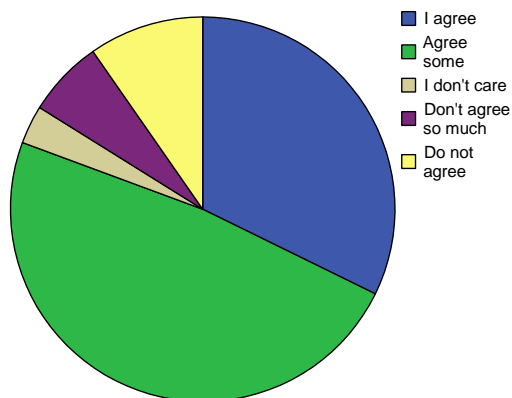
**What do you want to improve in today's development process? A more clear process**



Fråga 2: What do you want to improve in today's development process? A more clear unit test process

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	I agree	10	32,3	32,3	32,3
	Agree some	15	48,4	48,4	80,6
	I don't care	1	3,2	3,2	83,9
	Don't agree so much	2	6,5	6,5	90,3
	Do not agree	3	9,7	9,7	100,0
	<b>Total</b>	<b>31</b>	<b>100,0</b>	<b>100,0</b>	

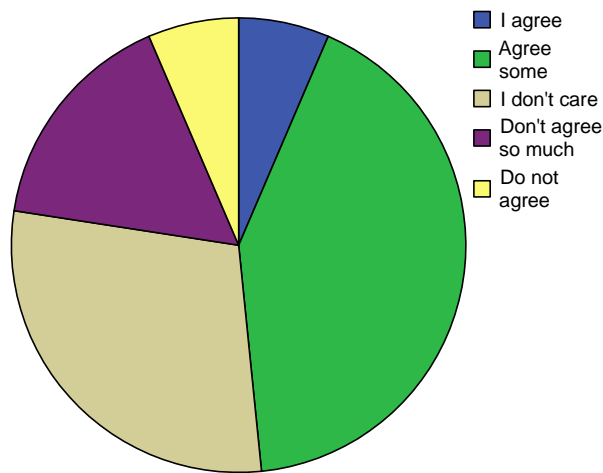
**What do you want to improve in today's development process? A more clear unit test process**



Fråga 2: What do you want to improve in today's development process? More design

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	I agree	2	6,5	6,5	6,5
	Agree some	13	41,9	41,9	48,4
	I don't care	9	29,0	29,0	77,4
	Don't agree so much	5	16,1	16,1	93,5
	Do not agree	2	6,5	6,5	100,0
	Total	31	100,0	100,0	

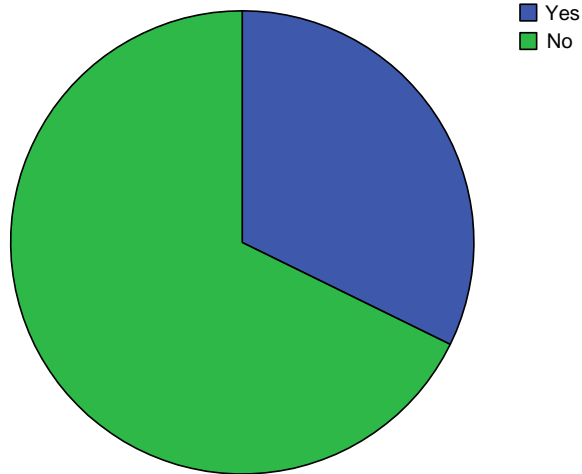
What do you want to improve in today's development process? More design



Fråga 3: Do you feel that you are testing your code sufficiently?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	10	32,3	32,3	32,3
	No	21	67,7	67,7	100,0
Total		31	100,0	100,0	

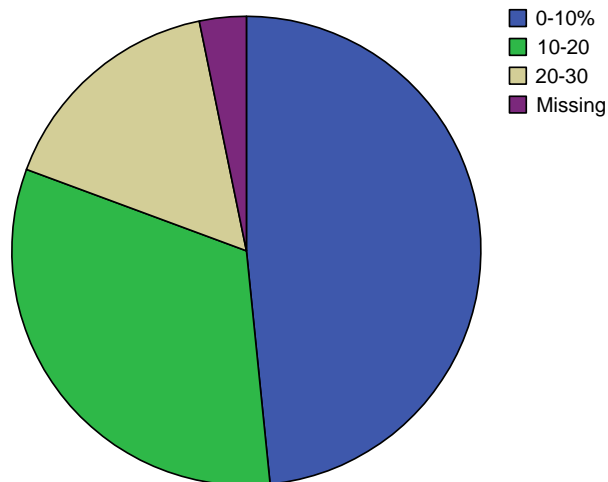
**Do you feel that you are testing your code sufficiently?**



**Fråga 4: How much of your total development time (design +code + testing) do you use for unit testing?**

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	0-10%	15	48,4	50,0	50,0
	10-20	10	32,3	33,3	83,3
	20-30	5	16,1	16,7	100,0
	Total	30	96,8	100,0	
Missing	System	1	3,2		
	Total	31	100,0		

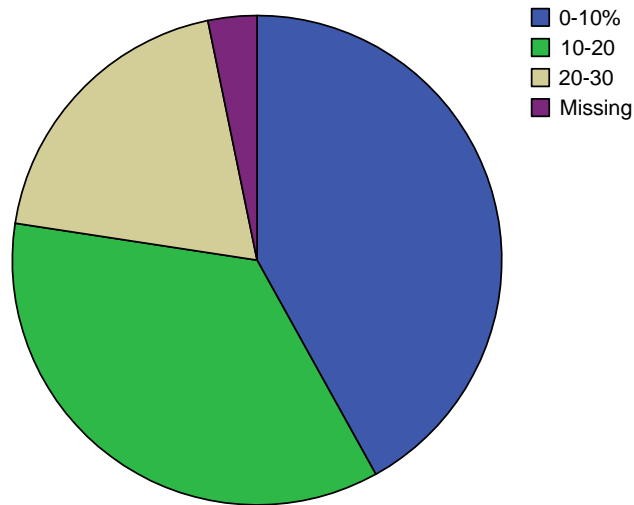
**How much of your total development time (design +code+testing) do you use for unit testing?**



Fråga 5: How much of your total development time (design + code + testing) do you plan for unit testing?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	0-10%	13	41,9	43,3	43,3
	10-20	11	35,5	36,7	80,0
	20-30	6	19,4	20,0	100,0
	Total	30	96,8	100,0	
Missing	System	1	3,2		
	Total	31	100,0		

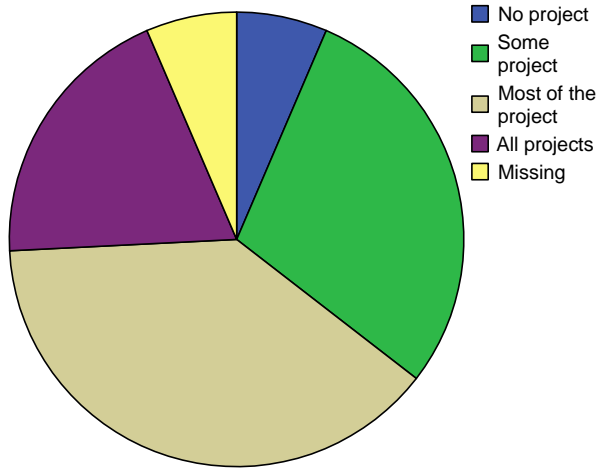
**How much of your total development time (design +code+testing) do you plan for unit testing?**



Fråga 6: How often does your testing get affected at the end of a project due to lack of time?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	No project	2	6,5	6,9	6,9
	Some project	9	29,0	31,0	37,9
	Most of the project	12	38,7	41,4	79,3
	All projects	6	19,4	20,7	100,0
	Total	29	93,5	100,0	
Missing	System	2	6,5		
	Total	31	100,0		

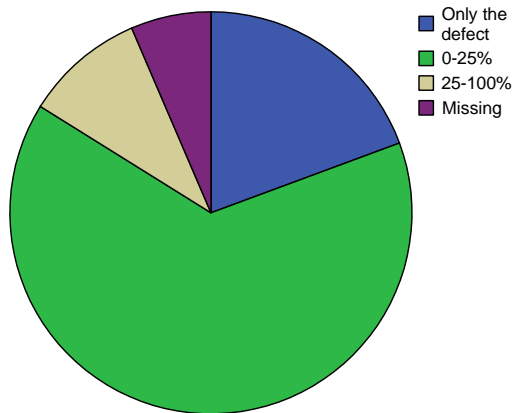
**How often does your testing get affected at the end of a project due to lack of time?**



Fråga 7: After you have corrected a defect that have been discovered at product verification, how much of your component do you regression test?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Only the defect	6	19,4	20,7	20,7
	0-25%	20	64,5	69,0	89,7
	25-100%	3	9,7	10,3	100,0
	Total	29	93,5	100,0	
Missing	System	2	6,5		
	Total	31	100,0		

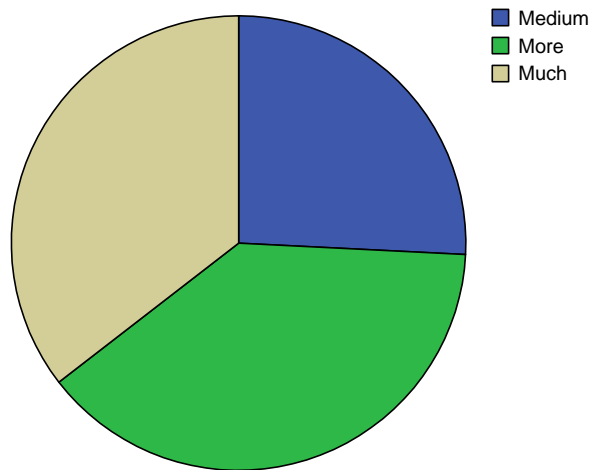
**After you have corrected a defect that have been discovered at product verificatio, how much of your component do you regression test?**



Fråga 8: How do you think that the following parts affect the code quality? Code review

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Medium	8	25,8	25,8	25,8
	More	12	38,7	38,7	64,5
	Much	11	35,5	35,5	100,0
	Total	31	100,0	100,0	

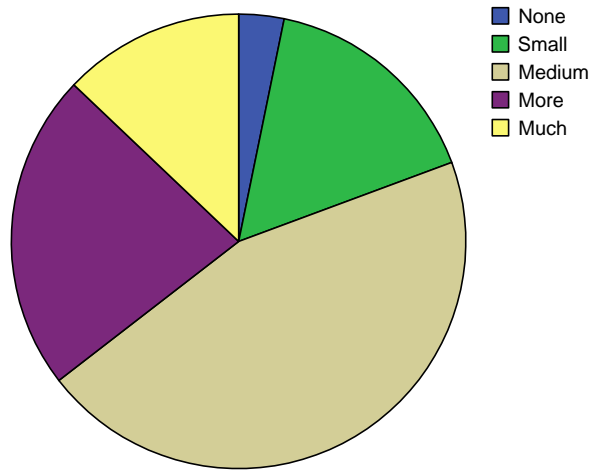
How do you think that the following parts affect the code quality? Code review



Fråga 8: How do you think that the following parts affect the code quality? Unit test specification

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	None	1	3,2	3,2	3,2
	Small	5	16,1	16,1	19,4
	Medium	14	45,2	45,2	64,5
	More	7	22,6	22,6	87,1
	Much	4	12,9	12,9	100,0
	Total	31	100,0	100,0	

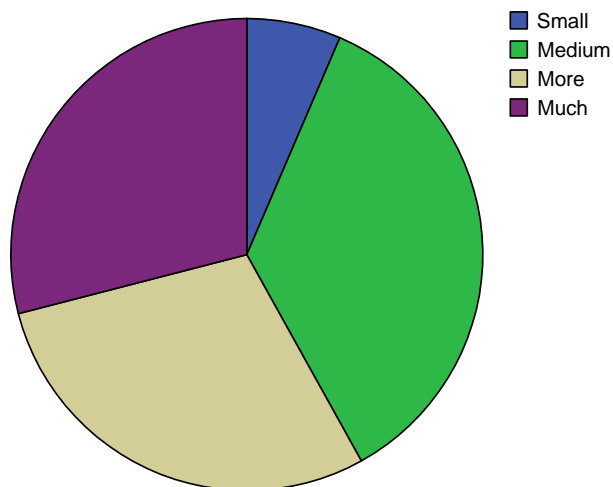
**How do you think that the following parts affect the code quality? Unit test specification**



**Fråga 8: How do you think that the following parts affect the code quality? Design phase**

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Small	2	6,5	6,5	6,5
	Medium	11	35,5	35,5	41,9
	More	9	29,0	29,0	71,0
	Much	9	29,0	29,0	100,0
	Total	31	100,0	100,0	

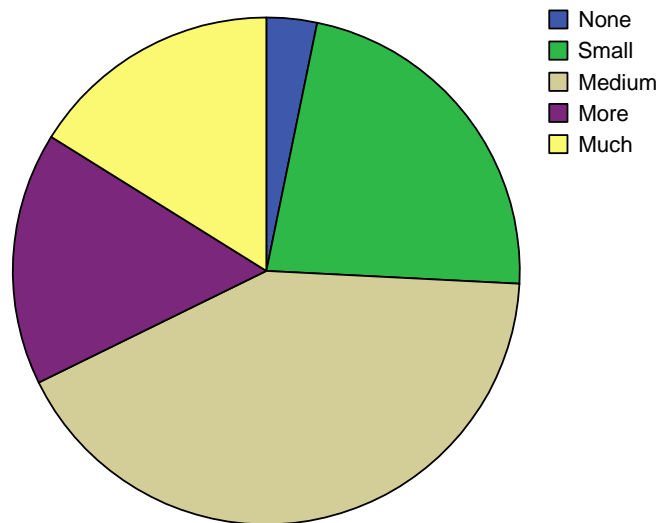
**How do you think that the following parts affect the code quality? Design phase**



Fråga 8: How do you think that the following parts affect the code quality? Pre-study phase

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	None	1	3,2	3,2	3,2
	Small	7	22,6	22,6	25,8
	Medium	13	41,9	41,9	67,7
	More	5	16,1	16,1	83,9
	Much	5	16,1	16,1	100,0
	Total	31	100,0	100,0	

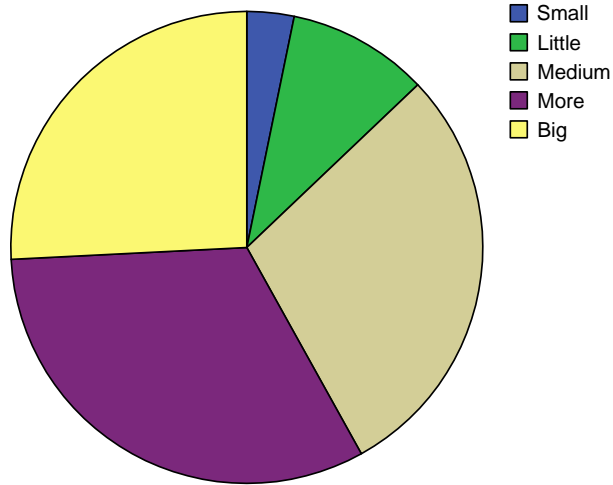
How do you think that the following parts affect the code quality? Pre-study phase



Fråga 9: What are your expectations for an automated unit testing process? Higher code quality

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Small	1	3,2	3,2	3,2
	Little	3	9,7	9,7	12,9
	Medium	9	29,0	29,0	41,9
	More	10	32,3	32,3	74,2
	Big	8	25,8	25,8	100,0
	Total	31	100,0	100,0	

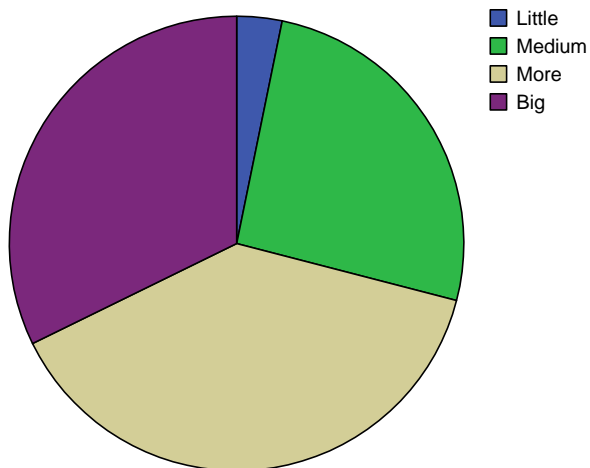
**What are you expectations for an automated unit testing process? Higher code quality**



**Fråga 9: What are you expectations for an automated unit testing process? Time saving**

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Little	1	3,2	3,2	3,2
	Medium	8	25,8	25,8	29,0
	More	12	38,7	38,7	67,7
	Big	10	32,3	32,3	100,0
	Total	31	100,0	100,0	

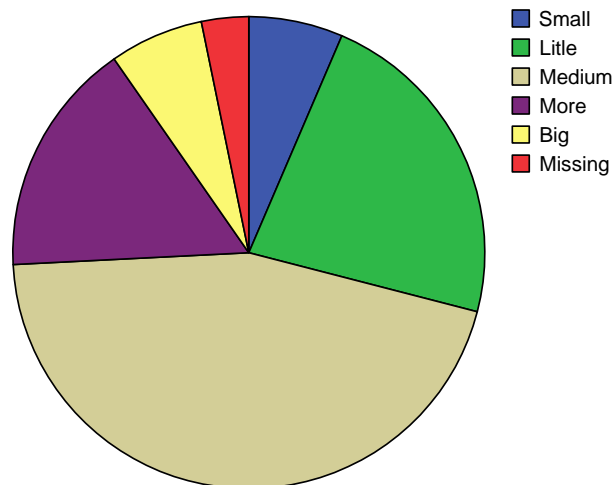
**What are you expectations for an automated unit testing process? Time saving**



Fråga 9: What are you expectations for an automated unit testing process? More work for the developer

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Small	2	6,5	6,7	6,7
	Little	7	22,6	23,3	30,0
	Medium	14	45,2	46,7	76,7
	More	5	16,1	16,7	93,3
	Big	2	6,5	6,7	100,0
	Total	30	96,8	100,0	
Missing	System	1	3,2		
Total		31	100,0		

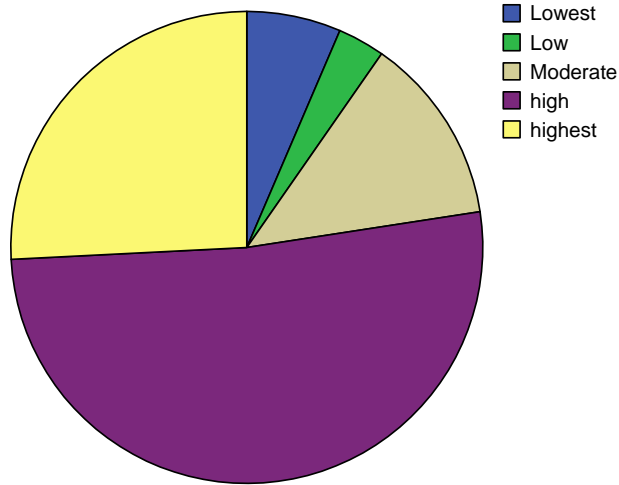
What are you expectations for an automated unit testing process? More work for the developer



Fråga 10: How much do you appreciate the advantages with an automated unit test process? More confidence when you are going to regression testing

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Lowest	2	6,5	6,5	6,5
	Low	1	3,2	3,2	9,7
	Moderate	4	12,9	12,9	22,6
	High	16	51,6	51,6	74,2
	Highest	8	25,8	25,8	100,0
	Total	31	100,0	100,0	

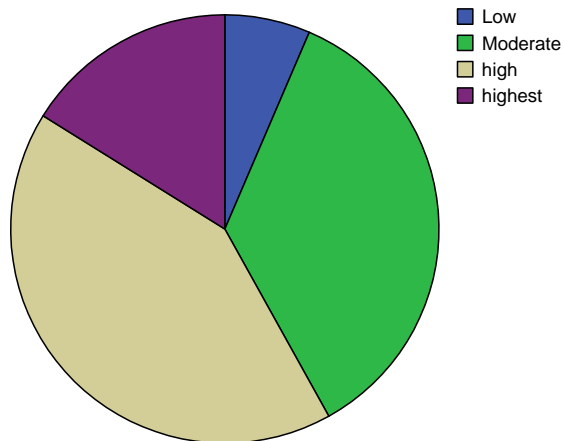
**How much do you appreciate the advantages with an automated unit test process? More confidence when you are going to regression testing**



Fråga 19: How much do you appreciate the advantages with an automated unit test process? Find the defects earlier

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Low	2	6,5	6,5	6,5
	Moderate	11	35,5	35,5	41,9
	High	13	41,9	41,9	83,9
	Highest	5	16,1	16,1	100,0
	Total	31	100,0	100,0	

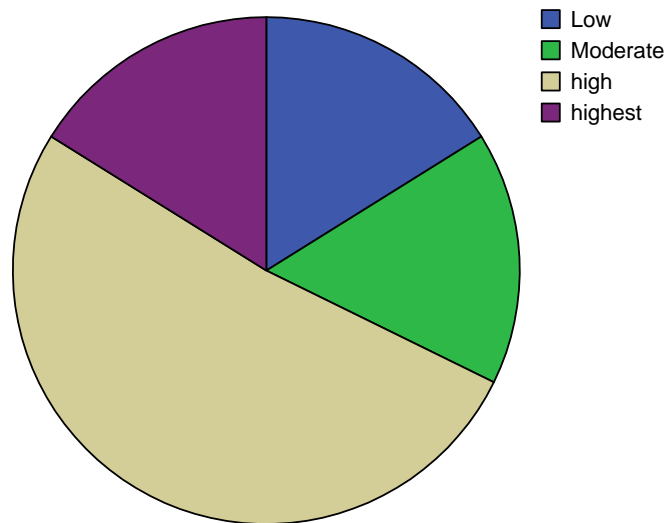
**How much do you appreciate the advantages with an automated unit test process? Find the defects earlier**



Fråga 10: How much do you appreciate the advantages with an automated unit test process? Higher code quality

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Low	5	16,1	16,1	16,1
	Moderate	5	16,1	16,1	32,3
	High	16	51,6	51,6	83,9
	Highest	5	16,1	16,1	100,0
	Total	31	100,0	100,0	

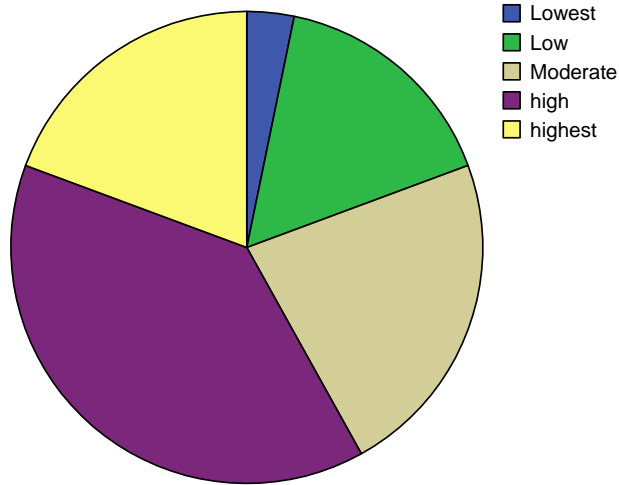
**How much do you appreciate the advantages with an automated unit test process? Higher code quality**



Fråga 10: How much do you appreciate the advantages with an automated unit test process? More confidence when refactoring

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Lowest	1	3,2	3,2	3,2
	Low	5	16,1	16,1	19,4
	Moderate	7	22,6	22,6	41,9
	High	12	38,7	38,7	80,6
	Highest	6	19,4	19,4	100,0
	Total	31	100,0	100,0	

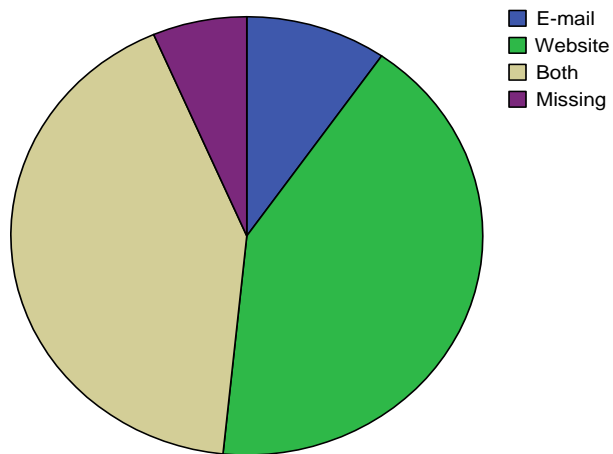
**How much do you appreciate the advantages with an automated unit test process? More confidence when refactoring**



Fråga 11: How do you want to have the result from an automated unit test presented?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	E-mail	3	9,7	10,3	10,3
	Website	13	41,9	44,8	55,2
	Both	13	41,9	44,8	100,0
	Total	29	93,5	100,0	
Missing	System	2	6,5		
	Total	31	100,0		

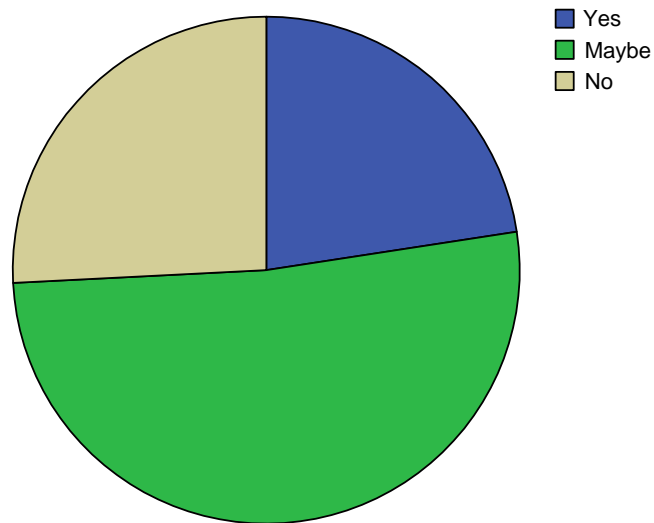
**How do you want to have the result from an automated unit test presented?**



Fråga 12: Would you like to have statistics at your code quality compared with other developers?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	7	22,6	22,6	22,6
	Maybe	16	51,6	51,6	74,2
	No	8	25,8	25,8	100,0
	Total	31	100,0	100,0	

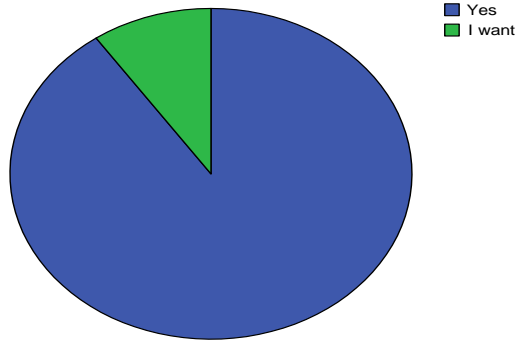
**Would you like to have statistics at your code quality compared with other developers?**



Fråga 13: Do you use these tools? Lint scan

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	28	90,3	90,3	90,3
	I want	3	9,7	9,7	100,0
	Total	31	100,0	100,0	

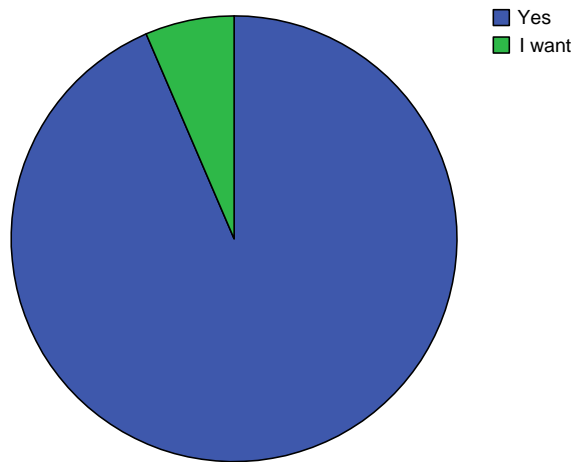
Do you use this tools? Lint scan



Fråga 13: Do you use these tools? Leave scan

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	29	93,5	93,5	93,5
	I want	2	6,5	6,5	100,0
	Total	31	100,0	100,0	

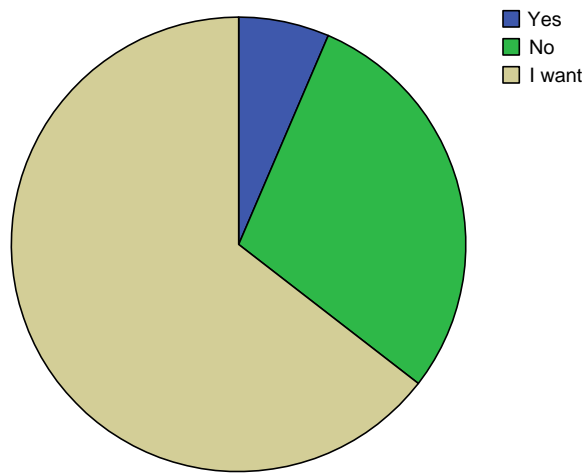
Do you use this tools? Leave scan



Fråga 13: Do you use these tools? TEFUnit

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	2	6,5	6,5	6,5
	No	9	29,0	29,0	35,5
	I want	20	64,5	64,5	100,0
	Total	31	100,0	100,0	

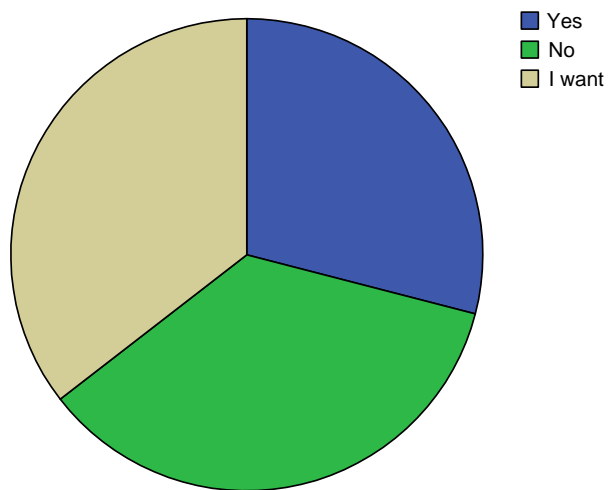
**Do you use this tools? TEFUnit**



**Fråga 13: Do you use these tools? Doxygen**

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	9	29,0	29,0	29,0
	No	11	35,5	35,5	64,5
	I want	11	35,5	35,5	100,0
Total		31	100,0	100,0	

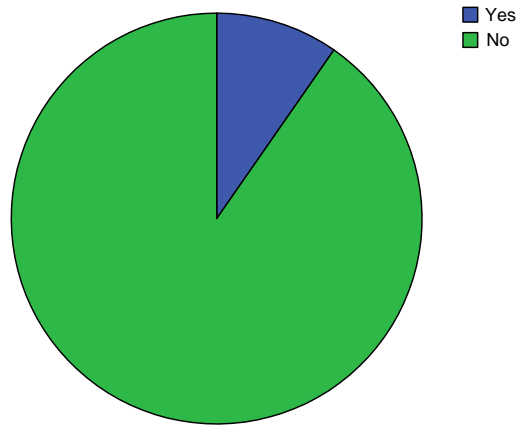
**Do you use this tools? Doxygen**



Fråga 14: Have you ever used Test driven development?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	3	9,7	9,7	9,7
	No	28	90,3	90,3	100,0
Total		31	100,0	100,0	

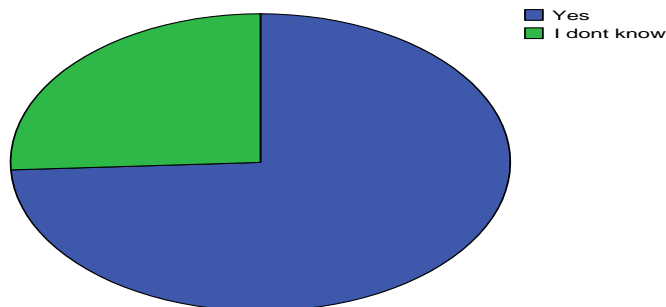
Have you ever used Test driven development?



Fråga 15: Are you prepared to try TDD if there were some courses in how to use it?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	23	74,2	74,2	74,2
	I dont know	8	25,8	25,8	100,0
Total		31	100,0	100,0	

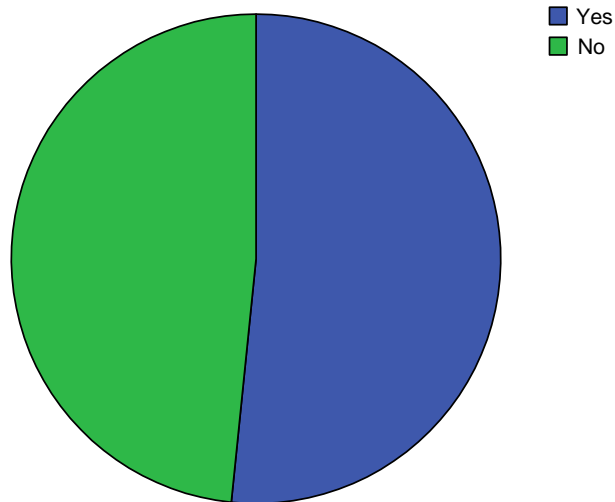
Are you prepared to try TDD if there were some courses in how to use it?



Fråga 17: Have you ever used pair-programming?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	16	51,6	51,6	51,6
	No	15	48,4	48,4	100,0
	Total	31	100,0	100,0	

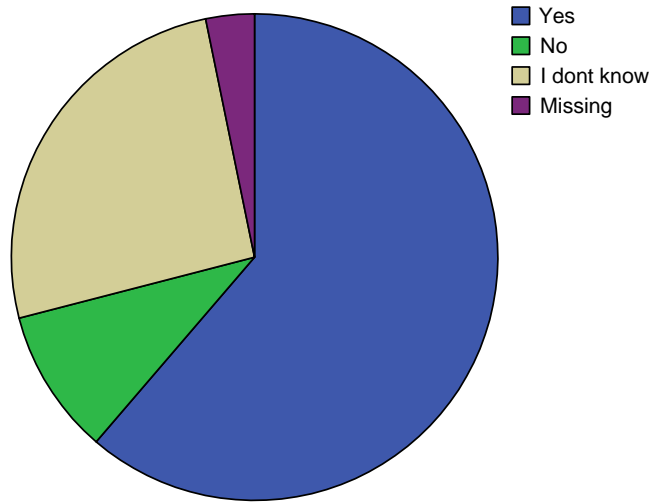
Have you ever used pair-programming?



Fråga 18: Are you prepared to try pair-programming if there were some course in how to use it?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Yes	19	61,3	63,3	63,3
	No	3	9,7	10,0	73,3
	I dont know	8	25,8	26,7	100,0
	Total	30	96,8	100,0	
Missing	System	1	3,2		
	Total	31	100,0		

Are you prepared to try pair-programming if there were some course in how to use it?



Fråga 20: What would you choose if you had a choice?

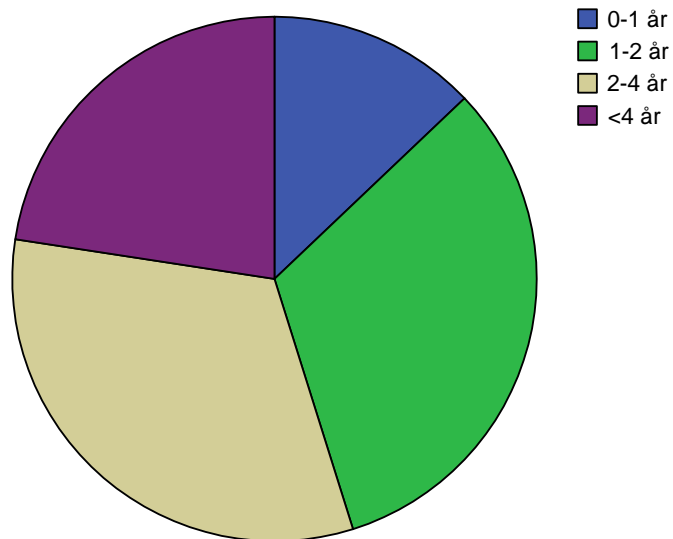
		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Single	13	41,9	41,9	41,9
	Pair	11	35,5	35,5	77,4
	I dont know	7	22,6	22,6	100,0
	Total	31	100,0	100,0	

---

Fråga 21: How long time have you been working at UIQ?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	0-1 år	4	12,9	12,9	12,9
	1-2 år	10	32,3	32,3	45,2
	2-4 år	10	32,3	32,3	77,4
	<4 år	7	22,6	22,6	100,0
	Total	31	100,0	100,0	

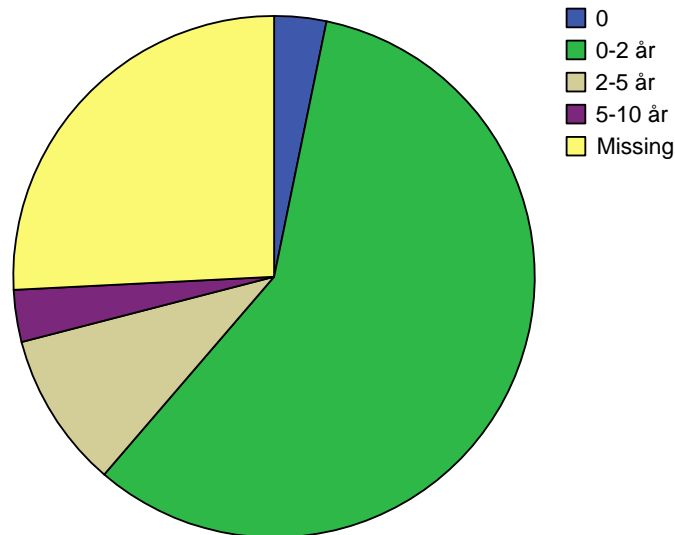
**How long time have you been working at UIQ?**



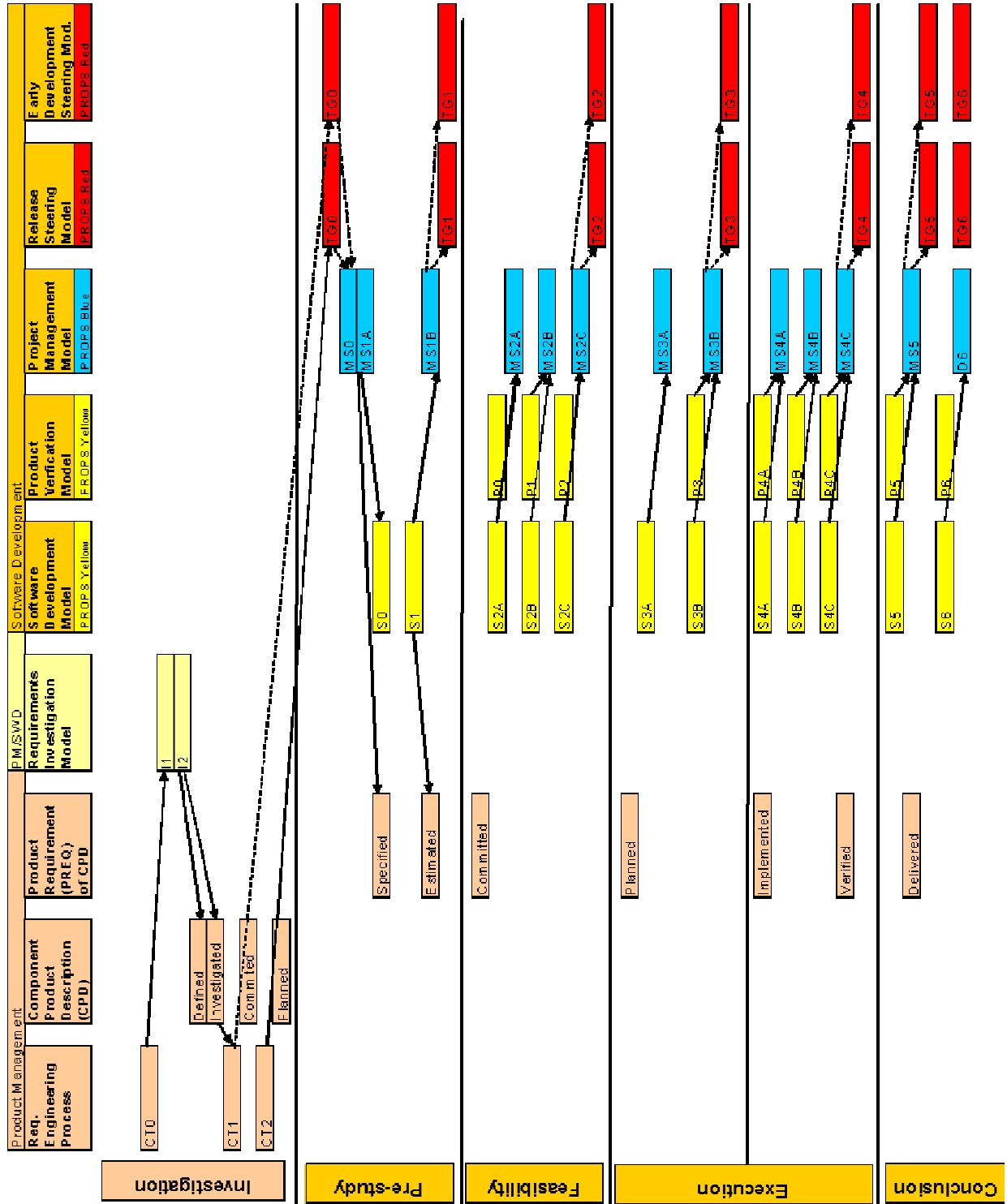
Fråga 22: Have you been working at some other IT-related company, in that case for how long?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	0	1	3,2	4,3	4,3
	0-2 år	18	58,1	78,3	82,6
	2-5 år	3	9,7	13,0	95,7
	5-10 år	1	3,2	4,3	100,0
	Total	23	74,2	100,0	
Missing	System	8	25,8		
	Total	31	100,0		

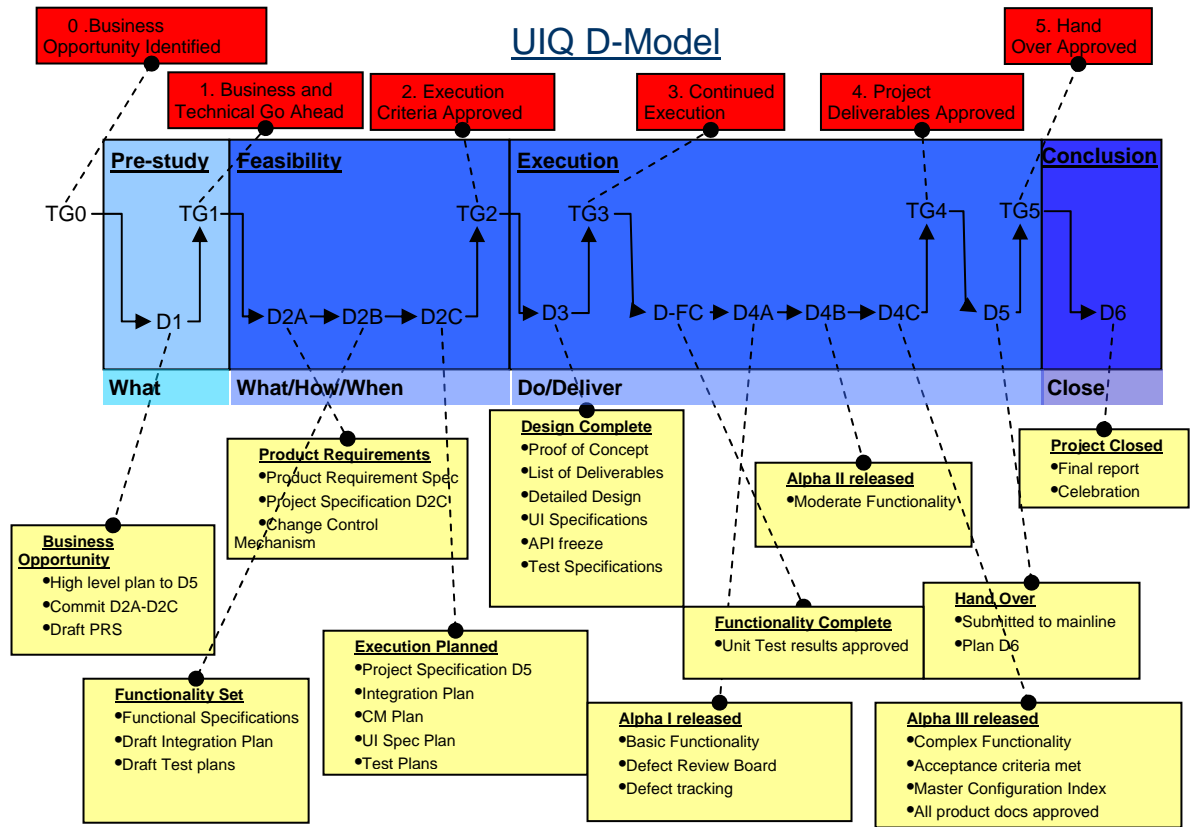
**Have you been working at some other IT-related company, in that case for how long?**



# APPENDIX III – Figurer



Figur 1: Software Development



Figur 2: UIQ D-modellen

---

## Appendix IV – Intervjufrågor fas ett

---

### ***Minnesläckor***

Hur länge har ni arbetat på företaget?

Vilken befattning har ni?

Vilken avdelning arbetar ni på?

Testar ni koden efter minnesläckor?

Är det vanligt att ni hittar minnesläckor?

Brukar det ta lång tid att åtgärda dem?

Kan ni uppskatta hur lång tid det tar:

Brukar test teamet återkomma med några minnesläckor?

Vilka metoder använder ni för att upptäcka minnesläckorna?

Använder ni någon sorts återupprepande test?

### ***Heapfailure***

Testar ni koden efter *heapfailure*?

Är det vanligt att ni upptäcker *heapfailures*?

Brukar det ta lång tid att åtgärda dem?

Kan ni uppskatta hur lång tid det tar:

Brukar test teamet återkomma med några *heapfailure*?

Vilka metoder använder ni för att upptäcka *heapfailure*?

Använder ni någon sorts återupprepande test?

---

## APPENDIX V – Intervjufrågor fas två

---

### ***Undersökning av den nuvarande processen***

Hur mycket är ni (utvecklaren) inblandad i dokument skrivandet innan kodningen startar?  
I så fall vilka dokument?

Vilka dokument skriver ni under själva kodningen?

### **Unit testing**

Följer ni unit test specifikationen?

Brukar ni titta på *unit* specifikationen innan slutet av processen?

Hur mycket tid avsätter ni till *unit* testing i jämförelse med hela er uppgift (kodning, dokument skrivande och testning)?

Brukar ni lägga den avsatta tiden på *unit* testing?

Om inte, varför då? (tidsbrist, ej vet hur man ska testa allt m.m)

### **Release frågor**

Är varje utvecklare ansvarig för att hans kod blir releasad?

Hur går en *release* till?

Kan ni lägga upp saker i *perforce* innan de är helt klara eller lägger ni enbart upp saker som ni unit testat till 100 %?

### **Regression testing**

Om test teamet skulle hitta en defekt som ni åtgärdar, kör ni i genom hela *unit* test specifikationen igen?

### **Övrigt**

Skriver ni några test innan ni börjar koda? Eller testar ni enbart efter att koden är så gott som klar?

Tycker ni dagens utvecklingsmetod fungerar bra? Vad är inte bra?

Tycker ni att det är tillräckligt med testning? Vill du ha mera tid för testning?

Jobbar ni självständigt eller jobbar man parvis?

Jobbar alla i teamet med samma utvecklings metod, eller väljer man den själv?

---

## APPENDIX VI – Intervjufrågor fas tre

---

### **Intervjufrågor om automatiserad testning**

Med Mr X (har en chefsposition på UIQ, 2005-02-02)

Allt som är skrivet i rött är min (Daniel Sandberg) tolkning av vad intervjuobjektet svarat.

### **Blandade frågor**

Är ni nöjda med dagens kvalitetsnivå?

Vi är inte nöjda med kodkvaliteten vid function complete, men vi är däremot nöjda med antalet defekter som når kunden.

Vad har ni för kvalitetsmål?

Vi har som vision att noll defekter skall nå kunden medan målet är kunden inte kommer med några ”returns” (UIQ inte får tillbaka koden för att åtgärda defekter).

Är ni positivt inställd till införandet av ett automatiskt test?

Ja, mycket!

Varför vill ni införa ett automatiserat test?

- Öka kvalitén, minska utvecklingstiden eller dra ner på personal

Minska utvecklingstiden totalt sett.

Tror du att det automatiska testet kommer att finnas kvar på UIQ på längre sikt?

Ja, med lite tvekan.

### **Förväntningar**

Vilka är dina förväntningar på ett automatiserat unit test?

- Öka kvalitén genom att hitta fler defekter

Ja, särskild på koden som når functional complete.

- Kortare utvecklingstid för projekten

Ja, men endast på tiden från functional complete och därefter.

- Personalresurser

Kommer nog inte påverkas.

- Kostnad för det automatiska testet (löpande, fasta)

På sikt en lägre kostnad, annars skulle vi inte satsat på ett automatiskt test

- Få bättre/mer statistik på hur kodkvaliteten är

Ja, men den skall inte användas för att jämföra teamen i mellan utan snarare för att jämföra med tidigare projekt.

- 
- Vad tror du om möjligheten att införa TDD på UIQ

Vi har inte diskuterat test driven utveckling ännu då det automatiska testet är så pass nytt. Men vi kommer att undersöka TDD närmare.

Vi bestämde även att jag skulle skicka mina artiklar om TDD till intervjuobjektet så han kunde sätta sig in närmare i hur TDD fungerar samt vilka för och nackdelar som finns.

---

## APPENDIX VII – Räkneexempel

---

Båda exemplen är hämtade från Hoffman D., “Cost Benefit Analysis of Test Automation”, white paper, 1999

### Automation of build tests examples:

Assumptions and computational values:

- Daily builds and test runs (5 times a week)
- Manual tests take 5 days to design, 2 hours to run
- Only half the manual tests would be run on any given day (1 hour) with the other half run the following day
- Automated tests take 15 days to design and implement, automatically run (zero cost)
- Automation is done with batch scripts and integrated into the build process, requiring \$1,000 in added hardware, with a useful life of 3 years
- Automated tests need to be maintained every 25 runs, one day of work required
- Periods of time ( $t$ ) selected: 6 months (125 days) and 18 months (375 days)
- People cost \$100,000 per year = \$400 per day = \$50 per hour

$\Delta B_a$  (in time  $t$ ) =

$\Sigma$  (improvement in fixed costs of automated testing times ( $t/Useful\ Life$ )) +  $\Sigma$  (variable costs of running manual tests  $n_2$  times during time  $t$ ) –

$\Sigma$  (variable costs of running automated tests  $n_1$  times during time  $t$ )

$$\Delta B_a \text{ (in 6 months)} = 0 + (1 \text{ hours} * 125) - (0 * 125) = (\$50 * 125) = \$6,250$$

$$\Delta B_a \text{ (in 18 months)} = 0 + (1 \text{ hours} * 375) - (0 * 375) = (\$50 * 375) = \$18,750$$

$$\Delta C_a \text{ (in time } t) = \Sigma \text{ (increased fixed costs of automated testing times } (t/Useful\ Life) ) +$$

$$\Sigma \text{ (variable costs of creating automated tests) –}$$

$$\Sigma \text{ (variable costs of creating manual tests) +}$$

$$\Sigma \text{ (variable costs of maintaining automated tests) times } (n_1/N)$$

$$\Delta C_a \text{ (in 6 months)} = (\$1,000 * (6/36)) + (15 * \$400) - (5 * \$400) + (\$400 * (125/25)) \\ = \$167 + \$6,000 - \$2,000 + \$2,000 = \$6,167$$

$$\Delta C_a \text{ (in 18 months)} = (\$1,000 * (18/36)) + (15 * \$400) - (5 * \$400) + (\$400 * (375/25)) \\ = \$500 + \$6,000 - \$2,000 + \$6,000 = \$10,500$$

$$ROI_{\text{automation}} \text{ (in 6 months)} = \$6,250 / \$6,167 = 1.013 \text{ [about break even]}$$

$$ROI_{\text{automation}} \text{ (in 18 months)} = \$18,750 / \$10,500 = 1.786 \text{ [80\% return]}$$

The return for 6 months is just above the investment cost and there is substantial return over 18 months. When the automated tests are run less frequently or require more maintenance there is much lower return. The return is also helped substantially by the low cost of creating and maintaining these tests.

---

## Automation of GUI tests example:

Assumptions and computational values:

- A new product with all new tests
- 5 people-years developing manual tests, 15 for automated tests
- 1 person maintenance after 1<sup>st</sup> year for automated tests
- 10 people full time running manual tests, 1 person for automated
- Fixed costs for automated tests of \$90,000 with useful life of 3 years
- Periods of time ( $t$ ) selected: 12 months (250 days) and 24 months (500 days)
- People cost \$100,000 per year = \$400 per day = \$50 per hour

$\Delta B_a$  (in time  $t$ ) =

$\Sigma$  (improvement in fixed costs of automated testing times ( $t/Useful\ Life$ )) +  
 $\Sigma$  (variable costs of running manual tests during time  $t$ ) –  
 $\Sigma$  (variable costs of running automated tests during time  $t$ )

$$\Delta B_a \text{ (in 12 months)} = 0 + (10 \text{ people} * \$100,000) - (1 \text{ person} * \$100,000) \\ = \$900,000$$

$$\Delta B_a \text{ (in 24 months)} = 0 + (10 \text{ people} * \$200,000) - (1 \text{ person} * \$200,000) \\ = \$1,800,000$$

$\Delta C_a$  (in time  $t$ ) =  $\Sigma$  (increased fixed costs of automated testing times ( $t/Useful\ Life$ )) +  
 $\Sigma$  (variable costs of creating automated tests) –  
 $\Sigma$  (variable costs of creating manual tests) +  
 $\Sigma$  (variable costs of maintaining automated tests during time  $t$ )

$$\Delta C_a \text{ (in 12 months)} = (\$90,000 * (1/3)) + (15 * \$100,000) - (5 * \$100,000) + \$0 \\ = \$30,000 + \$1,500,000 - \$500,000 = \$1,030,000$$

$$\Delta C_a \text{ (in 24 months)} = (\$90,000 * (2/3)) + (15 * \$100,000) - (5 * \$100,000) + \$100,000 \\ = \$60,000 + \$1,500,000 - \$500,000 + 100,000 = \$1,160,000$$

$$ROI_{\text{automation}} \text{ (in 12 months)} = \$900,000 / \$1,030,000 = 0.874 \text{ [small loss]}$$

$$ROI_{\text{automation}} \text{ (in 24 months)} = \$1,800,000 / \$1,160,000 = 1.552 \text{ [55\% return]}$$

The return on investment is lower and later than in the first example, but this example includes much more infrastructure in fixed costs and much higher investments in running both the manual and automated tests. By using the alternative forms of the ROI equations we computed it without needing to identify the exact numbers or costs of running either manual or automated tests, and used realistic estimates based upon staffing and job assignments of the test group.