TESTDESIGN FÖR PROGRAMVARA

TESTDESIGN FÖR PROGRAMVARA

SÅ TAR DU FRAM BRA TESTFALL

Av Torbjörn Ryber

No Digit Media
Ymsenvägen 3
120 38 Årsta

info@uppkopplat.se
www.nodigitmedia.se

Omslag: **Martin Roos**
Sättning: **Nina Mattsoff**
Textredigering: **Ulrika Halvarsson**
Tryck: **WS Bookwell, Borgå, Finland 2006**
ISBN: **91-976062-1-9**
© Torbjörn Ryber

Dedikation

Till Karin och Elin

Test är en extremt kreativ och intellektuellt utmanande uppgift.

Glenford Myers, The Art of Software Testing

DEL I 21
1 Inledning 23
1.1 Vem vänder sig boken till? 24
1.2 Bokens uppbyggnad 25
1.3 Vad finns inte med? 26
1.4 Författaren 26
1.5 Tack till 26
2 Introduktion 29
2.1 Otillräckliga krav 29
2.2 Oändligt antal möjliga testfall 31
2.3 Mindre tid 31
2.4 Fler tips 33
3 Vad är test? 35
3.1 Hur ställer vi frågorna? 36
3.2 Vilka frågor ställer vi? 37
3.3 Hur ser vi svaren? 37
3.4 Hur vet vi om svaren är korrekta? 37
3.4.1 Heuristiker 38
3.4.2 Automatiserad testning 38
3.5 Exempel låneansökan 38
4 Utvecklingsmodeller 41
4.1 Vattenfallsmodellen 41
4.1.1 Påverkan på test 43

4.2 V-modellen ... 43
4.2.1 Påverkan på test ... 45
4.3 W-modellen ... 46
4.3.1 Påverkan på test ... 47
4.4 Unified Process ... 47
4.4.1 Faser i processen ... 48
4.4.2 Påverkan på test ... 48
4.5 Agile ... 49
4.5.1 Påverkan på test ... 50
5 Angreppssätt ... 51
5.1 Testnivåer ... 51
5.1.1 Enhetstest ... 52
5.1.2 Enhetsintegrationstest ... 52
5.1.3 Systemtest ... 52
5.1.4 Systemintegrationstest ... 52
5.1.5 Acceptanstest ... 53
5.1.6 Testnivåer i produktion ... 54
5.2 Vilka tester ska vi göra? ... 56
5.3 Allt skrivet och planerat i detalj i förväg ... 56
5.4 The Context-Driven School ... 58
5.5 Den gyllene medelvägen ... 59
6 Utforskande testning ... 61
6.1 Definitionen av utforskande testning ... 62
6.2 Andra benämningar på utforskande testning ... 64
6.3 Se på det som ett angreppssätt - inte en teknik ... 66
6.4 Utforskande kontra skriptad testning ... 66
6.5 Hur känner man igen utforskande testning? ... 67
6.6 Varför prata om utforskande testning? ... 68
6.7 Utforskande testning från utsidan ... 69
6.8 Utforskande testning inifrån: en aktivitet som kräver skicklighet ... 71
6.8.1 Testdesign ... 72
6.8.2 Noggrann observation ... 73
6.8.3 Kritiskt tänkande ... 74
6.8.4 Diversifierade idéer ... 75
6.8.5 Rikliga resurser ... 76
6.8.6 Styrning av ditt eget arbete ... 77
6.8.7 Snabb inlärning ... 77
6.8.8 Statusrapportering ... 78
6.9 Utforskande testning beror på situationen ... 78
6.10 Att handleda utforskande testning ... 80
6.11 Var passar utforskande testning in? ... 82
6.12 Utforskande testning i verklig användning ... 83
6.13 Produktiviteten hos utforskande testning ... 86
6.14 Tack till ... 87
7 Testdesigntekniker - en översikt ... 89
7.1 Statiska och dynamiska tekniker ... 89
7.2 Statiska tekniker ... 89
7.2.1 Inspektion ... 90
7.2.2 Genomgång ... 91
7.2.3 Teknisk granskning ... 91
7.2.4 Informell granskning ... 91
7.2.5 Alternativ till granskning ... 91
7.3 Dynamisk testdesignteknik ... 92
7.3.1 Data ... 92
7.3.2 Flöden ... 92
7.3.3 Logik: regelverk, formler ... 93
7.3.4 Kombinatorik ... 93
7.3.5 Riskbaserad testning ... 93
7.3.6 Avancerad testning ... 94
7.4 Svart, vitt och grått ... 94
7.4.1 Beteendebaserad testteknik – Black Box ... 94
7.4.2 Strukturell testteknik – White Box ... 94
7.4.3 Kombinerat - Grey Box ... 95
7.5 Icke-funktionella tester ... 95
7.6 Hur många designtekniker finns det? ... 96
7.7 Bokens urval ... 97

7.8 Internationella standarden ISTQB ... 98
7.9 Den akademiska världen ... 98
7.10 Företag som är långt framme inom test ... 100
7.11 Övriga källor ... 103

DEL II ... 105
8 Övergripande process för testdesign ... 107
8.1 Förberedelser ... 107
8.2 Steg ett - skapa en modell över kraven ... 108
8.3 Steg två täck modellen, skapa grundtestfall ... 109
8.4 Steg tre - komplettera med testdata ... 109
8.5 Steg fyra - avancerad testning ... 109
9 Modeller ... 111
10 Ekvivalensgrupper och gränsvärden ... 115
10.1 Numeriska värden ... 116
10.1.1 Ekvivalensgrupper ... 116
10.1.2 Gränsvärden ... 117
10.1.3 Fler faktorer som påverkar grupperingen ... 118
10.1.4 Öppna gränser ... 119
10.2 Värden med gemensamma egenskaper ... 120
10.3 Textfält ... 122
10.4 Tidsberoende variabler ... 124
10.5 Logiska värden ... 125
10.6 Relationer mellan fält ... 125
10.7 Sekvenser med värden ... 126
10.8 Sammansatt lista exempel med bokbeställning ... 127
10.9 Luddiga grupper ... 128
10.10 Domäntester ... 129
10.11 Tips för ekvivalensgruppering ... 132
10.12 Mall och exempel ... 132
10.13 Övning ... 135
11 Test av verksamhetsprocesser ... 137
11.1 Modellen ... 138
11.2 Grundtestfall ... 140
11.3 Testdata ... 141
11.4 Avancerad testning ... 143
11.5 Test med hjälp av prototyper ... 143
12 Från användningsfall till testfall ... 145
12.1 Vad är användningsfall? ... 145
12.2 Användningsfall ... 146
12.2.1 Exempel: Användningsfall - Ta ut pengar ... 146
12.3 Modellen - ta fram flödesgraf ... 149
12.4 Skapa grundtestfall ... 150
12.4.1 Lista alla scenarier ... 150
12.5 Komplettera med testdata ... 151
12.5.1 Identifiera styrande variabler ... 151
12.5.2 Gruppera data ... 152
12.5.3 Skapa testfall för varje scenario ... 154
12.6 Avancerad testning ... 154
12.7 Övning ... 156
12.7.1 Användningsfall AF001 Bli Kund ... 156
13 Tillståndsgrafer ... 161
13.1 Modellen ... 163
13.1.1 Bankomaten ... 163
13.2 Skapa grundtestfall ... 164
13.2.1 Sätt att täcka grafen ... 164
13.2.2 Täckning enligt Chow ... 165
13.2.3 Skapa testfall ... 165
13.2.4 Övergångspar ... 167
13.2.5 Utöka tabellen med en kolumn ... 167
13.2.6 Skapa en dual graf ... 168
13.2.7 Övergångstriplar och mer ... 169

13.3 Testdata ... 170
13.4 Avancerad testning ... 170
13.5 Exempel: Klockan i Windows ... 171
13.6 Exempel: Pensionsförsäkring ... 174
13.7 Övning 1 Alkolås ... 177
13.7.1 Användningsfall Alkolås ... 177
13.7.2 Uppgift ... 179
13.8 Övning 2 - Bandspelare ... 180
14 Beslutstabeller och beslutsträd ... 181
14.1 Modellen ... 181
14.1.1 Identifiera parametrar och gruppera ... 183
14.1.2 Sammanfatta reglerna i en tabell ... 184
14.1.3 Kvalitetssäkra tabellen genom att skapa ett beslutsträd ... 185
14.2 Skapa grundtestfall ... 187
14.3 Testdata ... 188
14.4 Övning: Beslutstabell för kontroll av fält i en dialog ... 189
14.5 Övning: Beslutsträd på komplicerat regelverk ... 190
15 Grundläggande jämförelser ... 193
15.1 Boolsk algebra och matriser ... 194
15.2 Modellen ... 195
15.3 Skapa grundtestfall ... 197
15.4 Testdata ... 197
15.4.1 Tabell över data ... 197
15.4.2 Förslag på kompletta testfall ... 198
15.5 Avancerad testning ... 199
15.6 Övningsuppgift ... 200
16 Testa alla par ... 201
16.1 Teorin bakom parvisa tester ... 202
16.2 Programmet Allpairs ... 203
17 Test av datacykel ... 207
17.1 Modellen ... 208
17.2 Skapa grundtestfall för varje objekt ... 208
17.3 Testdata ... 209
17.4 Bestäm ordning och förutsättningar för testfallen ... 209
17.5 Exempel: variant på CRUD-tabell ... 209
17.6 Övningsuppgift ... 210
18 Test av syntax ... 211
18.1 Modell övergripande syntaxtester ... 211
18.2 Testdata övergripande syntaxtester ... 212
18.2.1 Checklista värden ... 212
18.2.2 Checklista layout ... 213
18.3 Modell - Detaljerad syntaxkontroll ... 214
18.4 Hur stor är texten? ... 215
19 Test av tidscykler ... 217
19.1 Modellen ... 217
20 Riskbaserad testning ... 219
20.1 Klassisk riskhantering ... 220
20.2 Heuristisk riskbaserad testning ... 221
20.2.1 Att undersöka riskerna inifrån ... 222
20.2.2 Riskerna utifrån och in ... 222
20.3 Riskkataloger ... 224
20.4 Attack patterns ... 225
20.5 Taxonomier ... 226
20.6 Q-patterns - frågemönster ... 226
21 För utvecklaren ... 231
21.1 Test av programlogik ... 232
21.2 Test av dataflöden ... 236
21.3 Verktygsstöd ... 238

DEL III ... 241
22 Organisera testfallen ... 243
22.1 Testfall ... 244
22.1.1 Storlek och uppdelning i testfall ... 245
22.1.2 Beståndsdelar ... 246
22.1.3 Exempel testfall för system- och acceptanstest ... 248
22.1.4 Fördelar och nackdelar med att dokumentera så här detaljerat ... 249
22.2 Testkedjor ... 249
22.2.1 Beståndsdelar ... 250
22.2.2 Resultat ... 252
22.2.3 Hur stor är en testkedja? ... 252
22.3 Testpaket ... 252
22.3.1 Beståndsdelar ... 252
22.3.2 Hur vet du vad som är ett bra testpaket? ... 253
22.4 Genomförandeplan ... 253
22.4.1 Innehåll ... 254
22.4.2 Exempel genomförandeplan översikt ... 255
22.4.3 Exempel genomförandeplan i detalj ... 256
23 Avvikelsehantering ... 257
23.1 Verktygsstöd ... 258
23.2 Arbetsprocess ... 259
23.2.1 Avvikelsen identifieras ... 259
23.2.2 Analys och beslut om åtgärd ... 259
23.2.3 Utredning och åtgärd ... 259
23.2.4 Verifiera rättningen ... 260
23.3 Innehåll i avvikelserapport ... 261
23.3.1 Rubrik ... 261
23.3.2 Upptäckt datum/person ... 261
23.3.3 Ansvarig person ... 261
23.3.4 Klassning ... 261
23.3.5 Beskrivning av avvikelse ... 261
23.3.6 Beskrivning av åtgärd ... 262
23.3.7 Prioritet ... 262
23.3.8 Typ av fel ... 262
23.3.9 Övriga fält ... 262
23.4 Uppföljning ... 263
23.4.1 Avvikelser ... 263
23.4.2 Antal testfall ... 264
23.4.3 Fallgropar ... 264
23.5 Analys i efterhand ... 265
24 När är vi klara med testerna? ... 269
24.1 Fem kriterier för att avsluta testerna ... 269
24.2 Tillräckligt bra kvalitet ... 270
25 Lösningsförslag till övningar ... 273
25.1 Ekvivalensgrupper ... 273
25.2 Tillståndsgraf ... 274
25.3 Användningsfall ... 279
25.4 Beslutstabell ... 283
25.5 Beslutsträd ... 283
25.6 Grundläggande jämförelser ... 286
25.6.1 Lösningsförslag ... 286
25.6.2 Diskussion om uteblivet testfall ... 286
26 Ordlista ... 287
27 Lästips ... 291
27.1 Test ... 291
27.2 Övrig litteratur ... 294
27.2.1 Användbarhet ... 294
27.2.2 Kravhantering och process ... 294
27.2.3 Filosofi, psykologi och vetenskapsteori ... 295
27.3 Internetresurser ... 296
27.4 Standarder inom test ... 296
28 Referenser ... 299
29 Index ... 303

DEL I

Dessa inledande kapitel är till för att ge dig övergripande information om vad test är. De innehåller reflektioner om hur test påverkas av olika utvecklingsmodeller och om olika typer av angreppssätt. Du får också en genomgång av vad du praktiskt måste kunna för att utföra testerna.

Det finns ett särskilt kapitel som handlar om utforskande testning, eller exploratory testing som det också är känt som. Jag har försökt att kategorisera olika typer av tester och testtekniker för att skapa ett sammanhang.

1 Inledning

Varje sann test av en teori är ett försök att falsifiera eller motbevisa den.
- Karl Popper

Det absolut viktigaste för att lyckas med test är bra testdesign. Det spelar ingen roll hur snygga dokument eller bra planer du har om testfallen inte håller måttet. I princip handlar hela boken om att förenkla en komplicerad verklighet till hanterbara modeller och utifrån dessa välja rätt testfall bland de varianter som finns. Det finns en klar styrka i att använda sig av formella tekniker för att hitta fler varianter av testfall på ett mer effektivt sätt.

Jag har arbetat med test inom flera olika branscher från stora jättar inom telekom, bank, försäkring och statliga verk till häftiga små internetföretag. Jag har också läst en hel del litteratur i ämnet. Även om det finns många böcker om test saknar de flesta en bra beskrivning av själva kärnan i testarbetet, nämligen effektiv testdesign. Mycket av testlitteraturen är relativt allmän och fokuserar mer på process, arbetsledning och vilka dokument som ska produceras. Om vi tar bort alla dessa aspekter får vi kvar frågan om vad test verkligen är. Hur gör jag? *Testdesign för programvara* tar upp dessa frågor i detalj.

Det finns en mängd standarder och certifieringar inom test men ingen av dem har sin styrka i testdesign. Beroende på vem du frågar kommer du att få olika svar om vad som är viktigast att kunna. Jag har försökt att samla de tekniker som är mest användbara för att du ska kunna göra ett bra jobb som testdesigner. En testdesigner ansvarar för att analysera och ifrågasätta det underlag som finns och sedan skapa testfall för att kontrollera hur väl programvaran fungerar mot kraven. Ibland används också benämningen testanalytiker på denna roll.

Utvecklingen inom IT går rasande fort. Mer och mer avancerade system byggs på kortare tid och kraven är ofta att det ska fungera utan fel och 24 timmar per dygn. Det som förr kanske gick att hantera med hårt arbete kräver idag ett mer metodiskt angreppssätt. För att du ska kunna göra bättre tester på kortare tid måste du jobba smartare än förut. Nyckeln ligger i att på ett strukturerat sätt angripa problemen med hjälp av en verktygslåda fylld av modelleringskunskaper och testtekniker. Om du behärskar många olika tekniker och vet när de ska användas har du en betydligt större chans att göra ett bra arbete.

I flera anföranden på 2005 års upplaga av den årligen återkommande testkonferensen EuroSTAR tog talarna upp vikten av att öka sin egen kompetens. I framtiden finns det inte någon plats för testare utan bra kompetens inom testdesign. Övriga viktiga områden att behärska är programmering och automatisering. [1] Materialet i denna bok har växt fram under många år och bygger på egna erfarenheter. Målet är att lära ut de tekniker som är viktigast för att du ska lyckas med testdesignen i dina projekt.

1.1 Vem vänder sig boken till?

Boken vänder sig främst till dig som arbetar med någon form av test eller utveckling av system där programvara finns med. Oavsett om du arbetar som utvecklare eller testare är tankarna och teknikerna generella och passar både problem av mer teknisk karaktär och övergripande verksamhetsprocesser.

Innehållet bygger på kursmaterial som tagits fram under många år och innehåller även referenser till de böcker och artiklar som använts som källmaterial. Boken lämpar sig därför väl både som underlag för akademisk undervisning och för självstudier.

1.2 Bokens uppbyggnad

Del I: Dessa inledande kapitel är till för att ge dig övergripande information om vad test är. De innehåller reflektioner om hur test påverkas av olika utvecklingsmodeller och om olika typer av angreppssätt.

Del II: Detta är huvuddelen av boken och handlar om hur du effektivt tar fram bra testfall. De testdesigntekniker som beskrivs är alla praktiskt användbara och kompletterar varandra. Teorierna bakom teknikerna beskrivs grundligt och varje teknik demonstreras med hjälp av exempel som du kan följa steg för steg. Till de viktigaste finns även övningsuppgifter. Att kunna använda teknikerna är ett första steg, att veta hur du ska anpassa dem till just ditt projekt är en kunskap som kommer ju mer du jobbar med dem.

Du kommer ofta att hamna i situationer där du måste anpassa de beskrivna teknikerna eller använda andra tekniker för att lösa problemen. Du är välkommen att dela med dig av detta till författaren för framtida reviderade upplagor.

Del III: Den sista delen i boken tar upp praktiska detaljer om hur du organiserar dina testfall, utför tester och hanterar avvikelser. Här finns också lästips för både test och andra intressanta ämnen.

[1] Windsor, Susan [2005]: EuroSTAR2005 Keynote, Gardiner, Julie [2005]: EuroSTAR2005

1.3 Vad finns inte med?

Tankar kring hur du leder testprojekt, förbättrar din testprocess eller allmän information om kvalitetsarbete ligger utanför bokens omfattning.

Statisk testning går vi igenom endast översiktligt. Icke-funktionell testning och verktyg ligger utanför omfattningen av denna bok.

För att lära dig de allmänna tankarna bakom test rekommenderas att du antingen går en grundläggande kurs i testmetodik som ISTQB Certifiering eller läser någon av de böcker som finns i ämnet. Se litteraturlistan för tips och läs mer på www.sast.se eller www.istqb.org.

1.4 Författaren

Torbjörn Ryber, som är civilingenjör i datateknik, arbetar sedan 1995 med kravhantering och test vid utveckling av datasystem. Han var en av de första i Sverige att certifiera sig enligt ISEB Practitioner. Han är en ofta anlitad föredragshållare och kursledare i testmetodik.

Torbjörn Ryber är också ordförande för föreningen SAST, Swedish Association for Software Testing, som arbetar med att sprida kunskap om test genom att arrangera konferenser. Internationellt har han deltagit i programkommittén för EuroSTAR.

1.5 Tack till

Många personer har varit inblandade i både framtagning och granskning av materialet. Ett stort tack till:

Mina chefer på Handelsbanken, särskilt Mats Eriksson och Lena Gredenhag, som har uppmuntrat mig och låtit mig använda delar av vårt interna undervisningsmaterial i boken.

Expertpanel och flitiga granskare: Ann-Charlotte Bolander, Hans Engberg, Lillemor Ehrner, Jesper Hansén, Dan Hernvall, Hans Larsson, Hans-Gunnar Leytman, Sofie Nordhamrén, Mattias Nordin, Ingvar Nordström, Peter Ohlsson och Gun Ström. Utan era synpunkter hade boken inte blivit vad den blev.

Peter Zimmerer, Harry Robinson och Vipul Kocher för att de låter mig använda delar av sitt eget material i denna bok. Cem Kaner, Rex Black, Michael Bolton och de andra i diskussionsgruppen Software Testing för deras tankar och svar på mina frågor.

Speciellt tack till James Bach för kapitlet om *utforskande testning* och för granskning av upplägget av boken. Dessutom är han en inspiration för oss framåtsträvande testare genom att ifrågasätta det vi tror att vi redan vet. Många av de tankar som finns med i denna bok har inspirerats av James.

2 Introduktion

Kriteriet för att en teori ska anses som vetenskaplig är att den är testbar.
- Karl Popper

Du som testare ställs dagligen inför en mängd olika problem. Kravbilden är ofullständig och ändras med tiden, systemen blir alltmer komplexa och samtidigt ska det gå snabbare att producera ny programvara. Hur löser vi det? Det finns ett antal bra grundtankar för hur du kan lyckas med test.[2]

2.1 Otillräckliga krav

Som underlag för all systemutveckling finns oftast någon dokumentation i form av krav. Ett vanligt problem är att de krav som finns inte är bra nog som underlag för att bygga upp våra tester. Jag skulle nog vilja säga att det i realiteten är omöjligt att få en perfekt kravspecifikation. Det är därför viktigt att en testdesigner engagerar sig tidigt i projektet och har synpunkter på kravens innehåll och testbarhet.

Testbarhet för krav innebär att de ska vara:

- tydliga: att det entydigt ska framgå vad kraven innebär

[2] Ibland skiljer vi i Sverige på rollerna testdesigner, som är specialiserad på att ta fram testfall, och testare, som utför testerna. Det vanligaste i utländsk litteratur är att använda testare som benämning på den person som håller på med både framtagning och utförande av testfall.

- förenliga med varandra: att olika krav inte motsäger varandra
- kompletta: de krav som finns beskriver hela bilden
- mätbara: vi ska kunna skapa testfall för dem

Vårt mål som testare är att köra ett antal testfall och utifrån resultatet av dessa säga om kraven verkar vara uppfyllda eller inte.

Att granska dokument rätt upp och ner med någon formell metod som exempelvis inspektion kan ta bort många fel. Vissa mer informella genomgångar av dokument är vanliga och identifierar de grövsta felen, men enligt min erfarenhet är det få som granskar på ett effektivt sätt då detta kräver en hel del tid och bra rutiner.

Ett effektivt alternativ att få testbara krav är att studera alla underlag du kan få tag på och sedan bygga modeller som beskriver kraven, exempelvis matriser, diagram, grafer eller flöden. Enligt utvecklingsprocesser som exempelvis RUP ska detta göras av en kravanalytiker, men faktum är att det många gånger är testarna som får göra jobbet. Utifrån modellerna kan vi sedan ställa frågor av typen: ”Utifrån vad ni har skrivit/sagt tolkar jag att det är så här det ska fungera. Stämmer det med er uppfattning?” Det fungerar oändligt mycket bättre än att bara protestera mot att underlaget inte räcker till. När det kommer ändringar i kraven är det relativt enkelt för oss att se hur det påverkar våra modeller och då komma på hur vi måste anpassa testfallen.

Om det är riktigt illa ställt och dokumentation av kraven saknas helt är det svårt att göra riktigt bra tester. I de fallen kommer testdesignen i stort att likna kravarbete, modellerna och de färdiga testfallen utgör på något sätt samtidigt kravmassan. Det kan vara aktuellt att ordna möten där utvecklare, kravställare och andra gemensamt får svara på frågor om hur systemet ska fungera. Det går att testa helt utan krav men resultatet blir inte lika bra. Se även kapitel 6 om utforskande testning där det mest extrema fallet är att vi inte har några testinstruktioner alls nedskrivna innan vi börjar utföra tester.

2.2 Oändligt antal möjliga testfall

Även för ett mycket litet system eller en funktion finns det oändligt många möjliga testfall. En funktion kan oftast användas på flera olika sätt, med en kombination av olika indata och med andra händelser som sker samtidigt i systemet. För en testare gäller det att komma på vilka kombinationer och händelser som det är viktigt att testa.

Ett bra testfall har en god möjlighet att hitta ett hittills oupptäckt fel. Eftersom det är omöjligt att täcka in alla olika varianter av händelser och data gäller det att hitta den delmängd av tänkbara testfall som har största möjligheten att hitta flest allvarliga fel. [3]

Det gäller att skriva testfall både för korrekta och inkorrekta händelser. Tänk på att det ofta finns betydligt fler möjligheter att göra fel än att göra rätt. De olika teknikerna för testdesign som finns är bra för olika saker. En enda teknik hittar inte alla typer av fel – i varje projekt använder du dig därför av ett flertal olika. [4]

I princip handlar hela den här boken om att förenkla en komplicerad verklighet till hanterbara modeller och utifrån dessa välja rätt testfall bland de varianter som finns. Du kommer nog att känna igen många av de tekniker som beskrivs – det är vanligt att de används, om än på ett mer informellt sätt. Det finns en klar styrka i att använda sig av formella tekniker för att hitta fler varianter av testfall på ett mer effektivt sätt.

2.3 Mindre tid

Boehms klassiska modell över hur mycket det kostar att hitta fel vid olika tidpunkter i projektet, visar en kurva där kostnaden för att korrigera ett fel ökar exponentiellt med tiden.[5] Cem Kaner skriver i en mer utförlig analys om att det är dyrt att ändra kraven då man redan kodat eller att rätta en produkt som redan sålts till en kund.[6] Även mer nyligen genomförda mätning-

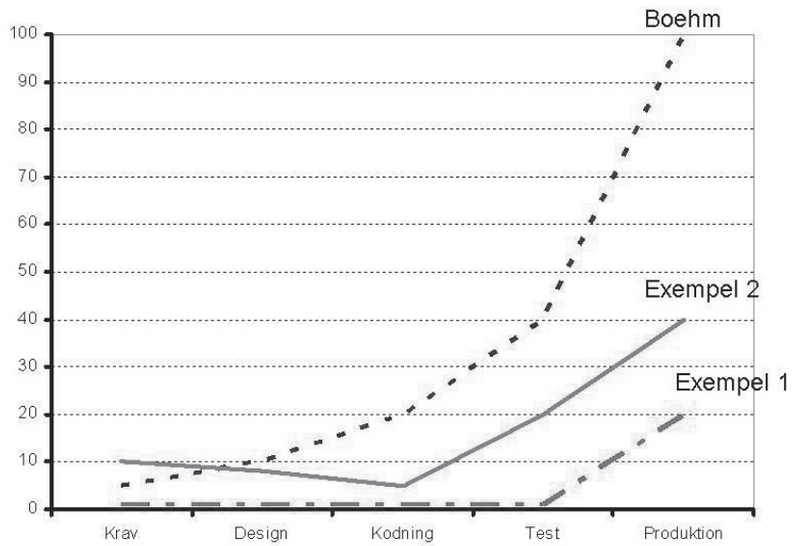
[3] Myers, Glenford [1979]: The Art of Software Testing, s 36. [4] Beizer, Boris [1995]: Black Box Testing, s 11. [5] Boehm, B W [1976]: Software Engineering, IEEE Transactions on Computers, C-25. [6] Kaner, Cem, Jack Falk, Hung Nguyen [1999]: Testing Computer Software, s 31

ar visar att kostnaden för att korrigera fel ökar med en faktor hundra från kravfas till produktion.[7] Jag accepterar de övergripande resultaten men anser att de presenterar en alltför stor förenkling av verkligheten för att ge oss ett bra stöd.

Det finns ingen absolut sanning i hur mycket varje enskilt fel kostar att rätta och ett problem med hur Boehms graf presenteras är att den tolkas som att alla fel alltid ska hittas så tidigt som möjligt. En mer detaljerad analys skulle peka på att du ska hitta *rätt* sorts fel vid *rätt* tillfälle. Fel som har med kraven eller designen av systemet att göra bör du hitta innan kodningen startat – helt rätt. Dessa typer av fel är dyra att korrigera sent i ett projekt då tid har lagts på att specificera, koda och testa den felaktiga designen. För denna typ av fel stämmer Boehms kurva bra. (*Boehm i grafen på nästa sida.)* Vissa typer av kodningsfel bör hittas och korrigeras av utvecklaren då denne kan både testa och korrigera felen på kortare tid än någon annan. Däremot kan de inte hittas tidigare då det inte finns någon kod att testa.

För vissa enklare typer av fel kan kostnaden per fel vara densamma fram till och med test. *(Ex 1 i grafen)*. Dessutom finns det faktiskt vissa typer av fel som är både svårare och dyrare att hitta tidigt i ett projekt. Utan att ha gjort någon avancerad forskning i ämnet sticker jag ut hakan och påstår att vissa fel är både lättare och billigare att hitta i systemtest än i programtest. Detta kan bero på att det ofta kostar väldigt mycket att sätta upp en bra miljö i tidiga faser, att vi inte har hela bilden klar tidigt i utvecklingen eller att enklare kodningsfel är förhållandevis lätta att korrigera även sent i projektet. Dessutom är vi som testare specialister och jobbar därför effektivare med att skriva bra testfall jämfört med utvecklarna. På samma sätt kostar det ibland mer att granska testfallen noggrant än att analysera avvikande resultat efter testfallets utförande. Här kommer ett enkelt exempel för att beskriva detta. I ett projekt skapade jag 120 testfall i form av två filer med sammanlagt 35 000 fält. Att granska alla dessa fält så att testfallen blev helt felfria skulle ta mig flera dagar. Vi körde istället testdata genom programmet som skulle testas och hittade då tolv fel som berodde på testdata. Rättning av testdata, plus *omtest* tog endast två timmar i anspråk, betydligt mindre tid än en analys hade tagit. *(Ex 2 i grafen)*.

Jag är övertygad om att gör vi rätt saker vid rätt tillfälle kan vi göra ett bättre arbete på kortare tid! Det låter som en utopi men jag vet av egen erfarenhet att det fungerar. Modellering och effektiv testdesign är två viktiga steg på vägen.



Figur 2.1: Kostnad för att hitta och att korrigera fel vid olika tidpunkter i utvecklingen. Boehms klassiska kurva visar att kostnaden för att hitta och korrigera fel ökar exponentiellt med tiden. Övriga kurvor visar på hur kostnaden kan variera för specifika typer av fel.

2.4 Flertips

Förutom punkterna ovan finns här några användbara tips för hur du lyckas som testdesigner.

[7] Wiegers, Karl [2003]: Software Requirements, s 17

▶ Var flexibel. Varje projekt är unikt och bär med sig olika utmaningar för dig att lösa, förmågan att situationsanpassa ditt arbete är nödvändig.

▶ Ha som ledstjärna att du ska tillföra nytta till ditt projekt. Tänk på det innan du gör eller säger något så slipper du skaffa dig fiender. Du är som testare ofta bärare av dåliga nyheter så det är viktigt för dig att vara både diplomatisk och positiv.

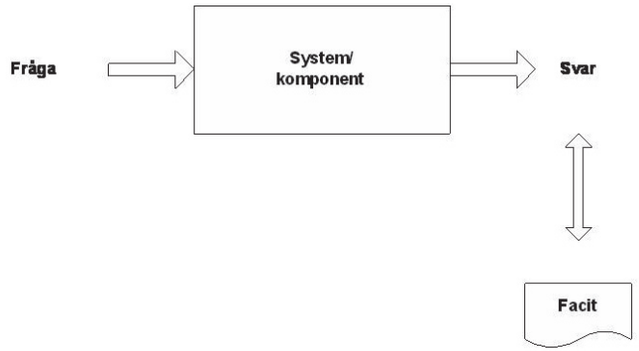
▶ Gör en kompetensplan för dig själv så att du blir mer värdefull. Förutom att det höjer din status och förhoppningsvis din lön så är det oändligt mycket roligare att jobba med test ju bättre du är. Börja med att läsa den här boken och öva på dina projekt - men sluta inte där. Det finns massor av litteratur, information på webben, konferenser och kurser som är värdefulla för din framtid.

▶ Lär dig modellera system och funktioner, du kommer att behöva analysera många krav och då är denna kunskap ovärderlig.

3 Vad är test?

Test är att ställa frågor till ett testobjekt och jämföra svaren med någon sorts facit för att avgöra om en viss aspekt av objektet fungerar som avsett. Andra har formulerat det lite annorlunda.

> Test är processen att använda ett system eller komponent under givna förhållanden, att observera eller notera resultaten och göra en utvärdering ur någon synvinkel av systemet eller komponenten.[8]
> Test är processen att exekvera ett program med syfte att hitta fel.[9]
> Test är att ställa frågor till en produkt med syfte att utvärdera den.[10]



Figur 3.1: Test är att ställa frågor och utvärdera svaren mot någon sorts facit. Alla tester har fyra huvudproblem: hur ställer vi frågorna, vilka frågor ska vi ställa, hur ser vi svaren och hur får vi fram facit?

[8] IEEE Standard 610.12-1990. [9] Myers, Glenford [1979]: The Art of Software Testing, s 5
[10] Bach, James [2005]: Rapid Testing, kursmaterial

Ovanstående bild säger väldigt mycket om vilka utmaningar vi ställs inför då vi arbetar som testare. Vi måste både kunna *ställa* frågor och dessutom veta *vilka* frågor vi ska ställa. Vi måste kunna tolka svaren och ha något att jämföra dem med. I mer detaljerade tester och i enhetstester måste vi även studera vad som händer inuti systemet. Test kan även ses som ett sätt att mäta hur långt vi kommit med utvecklingen.

Förutom ovanstående definitioner finns det som kallas statisk testning. Detta innebär att vi utan att exekvera någon kod undersöker och ifrågasätter om det som finns är korrekt. Teknikerna för granskning skiljer sig helt från vår beskrivning ovan av vad test är. Även om statiska tekniker omfattas av begreppet test enligt de flesta standarder, så saknar samtliga listade definitioner något om detta vilket kan tyckas lite märkligt.

3.1 Hur ställer vi frågorna?

I vissa fall kan vi som användare själva mata in data via ett grafiskt eller textbaserat gränssnitt. Här använder vi oss främst av tangentbord och mus även om det kan finnas andra sätt att påverka systemet som exempelvis fjärrkontroller (tv, stereo), utandningsluft (alkolås) eller trycksensorer (trafiksignaler, hissar).

Det är också vanligt att använda någon sorts filgränssnitt, antingen förutbestämt eller skapat just för test. Det finns även programmerbara gränssnitt (API) som gör det möjligt att i program och insticksmoduler utnyttja funktioner för vissa tjänster som finns tillgängliga i ett annat program eller i en funktionssamling.

Praktiska problem vi ofta råkar ut för är att det är svårt att ställa frågor innan hela systemet är klart. Dessutom kan det gränssnitt som slutanvändaren kommer att ha vara opraktiskt eller omöjligt att använda för att köra den stora mängd testfall vi vill genomföra.

Om det är möjligt kan vi begära att systemet byggs med extra funktionalitet för att utföra tester – d v s inbyggd testbarhet. Det underlättar för testerna om det är möjligt att göra ett urval i den totala mängden data vid batchbearbetningar, exempelvis genom att ange ett kontonummer, ett intervall för namn eller personnummer och använda dessa urvalskriterier på produktionsdata som är avidentifierat.

3.2 Vilka frågor ställer vi?

Ur den oändliga mängd frågor som oftast finns gäller det för oss att välja rätt. Vi måste välja de frågor som är tillräckligt bra och som tillsammans ger en godtagbar täckning. Ju fler svåra frågar vi lyckas ställa, desto mer vet vi om hur bra vårt system fungerar. Hela den här boken handlar om just detta problem som vi kallar testdesign.

3.3 Hur ser vi svaren?

På samma sätt som vi ställer frågorna kan vi också få svaren via ett grafiskt eller textbaserat gränssnitt. Det är också vanligt att vi får en svarsfil eller en pappersrapport och ibland en ljudsignal eller en annan mekanisk reaktion från exempelvis en mobiltelefon, hiss eller mikrovågsugn.

3.4 Hur vet vi om svaren är korrekta?

Ett av våra problem är att få fram ett facit att jämföra med, det kallas ofta orakelproblemet. Beroende på vilken typ av tester vi gör finns det olika orakel att tillgå. Några av de vanligaste är:

- Förväntat resultat beskrivet som en del av kraven.
- Manuellt uträknat.

- Beräkningar i någon sorts verktyg: t ex MS Excel.
- Tidigare versioner av produkten: i de fall vi gör en uppdatering.
- Andra liknande produkter som vi räknar som korrekta: t ex interna och externa kundsystem med samma information och funktioner.
- Färdiga testsviter som någon annan byggt och testat. Används ofta för att testa kompilatorer, webbläsare och SQL-verktyg. [11]

3.4.1 Heuristiker

James Bach lägger till det han kallar heuristiker för orakel vilket är mer allmänna svar på hur ett system ska bete sig. [12]

- Stämmer med hur vi förväntar oss att det ska se ut
- Stämmer vid jämförelse med liknande produkter
- Stämmer överens mellan olika vyer inom systemet
- Stämmer med hur vi tror en kund vill ha det

3.4.2 Automatiserad testning

Vid automatiserade tester för säkerhet och robusthet är det helt andra orakel som gäller. Harry Robinson som är en av de ledande experterna på automatiserad robusthetstestning pratar om att köra systemet länge med massor av indata och kontrollera följande: [13]

- Otillåtna felmeddelanden ska inte dyka upp – t ex sök efter texten *Error.*
- Systemet ska inte krascha.
- Låsningar där du fastnar på något ställe ska inte finnas.

3.5 Exempel låneansökan

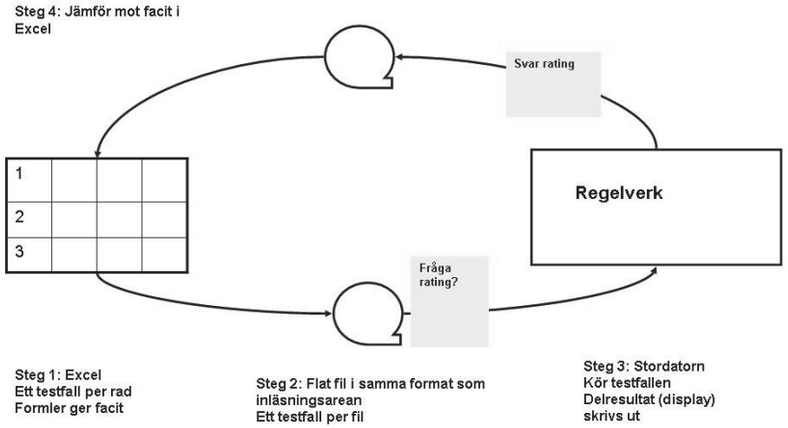
Vi ska testa ett system som hanterar låneansökningar. Ur användarens synpunkt består systemet av en dialog där man matar in vissa uppgifter om kunden som exempelvis lön. Information om kundens tillgångar och befintliga lån samt eventuella anmärkningar finns redan i dialogen. Så ur synvinkeln med systemtest ställer vi både frågor och får svar via en dialog. Vi har dock ett annat problem, på grund av att tiden är begränsad måste vi även testa regelverkskomponenten innan dialogen är färdig.

- **Problem 1:** Testa regelverket innan dialogen är färdig eftersom tiden är begränsad.
- **Problem 2:** Delresultaten: Det är svårt att testa regelverket i detalj utan att även se delresultaten.

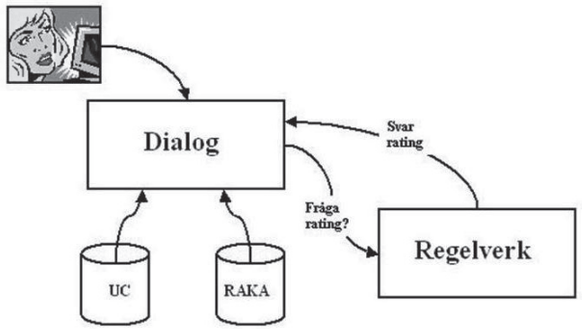
För att testa regelverkskomponenten i förväg krävs gränssnitt både för frågor och för svar.

1. Testfallen skrivs i en textfil som har samma format som den *inläsningsarea* som regelverkskomponenten har. Vi låtsas alltså att informationen kommer från dialogen.
2. Då vi inte bara vill ha svaret utan även vill kontrollera att de ingående variablerna hanteras korrekt ber vi om att delresultaten ska tas fram genom att en så kallad *display* skrivs ut på fil under tiden regelverket arbetar.
3. Vi skapar testfallen i en Excel-mall där regelverket simuleras. Denna mall innehåller både delresultaten och slutligt facit.
4. Vi lämnar alla testfall och förväntat resultat till utvecklaren som själv gör en första test.
5. Vid analys av våra testresultat hittar vi endast ett fåtal fel. På grund av att vi skrivit testfallen tidigt och gett dessa till utvecklaren har vi även bidragit till en bättre enhetstest.
6. Sista steget blir att testa dialogen ihopkopplad med regelverket. Nu vet vi att reglerna är korrekt implementerade och fokuserar på att dialogen skickar rätt data till regelverket. Som vi förväntat oss hittar vi fel i kopplingen mellan dialog och regelverk men inga nya fel i regelverkskomponenten i sig.

[11] Copeland, Lee [2003]: A Practitioner´s Guide to Software Test Design, s 6-7. [12] Bach, James [2005]: Rapid Testing, kursmaterial. [13] Robinson, Harry [2005]: EuroSTAR Tutorial Model-Based Testing



Figur 3.2: Strategi för test av regelverkskomponenten separat. För detta krävs att vi tar fram gränssnitt både för frågor och för svar. Svaret tas fram i en Excel-mall.



Figur 3.3: Test av dialogen kopplad till regelverket. Vi använder dialogens grafiska gränssnitt för både frågor och svar. Facit är samma Excel-blad som tidigare. Fokus är nu på integrationen mellan delarna.

4 Utvecklingsmodeller

Testprocessens kvalitet bestämmer om testinsatsen lyckas. [14]

I början rådde kaos - sen kom den första utvecklingsmodellen. Idag finns en mängd olika modeller som alla har sina styrkor och svagheter. För oss är det intressant att veta hur våra förutsättningar för testarbetet påverkas beroende på hur resten av projektet arbetar.

4.1 Vattenfallsmodellen

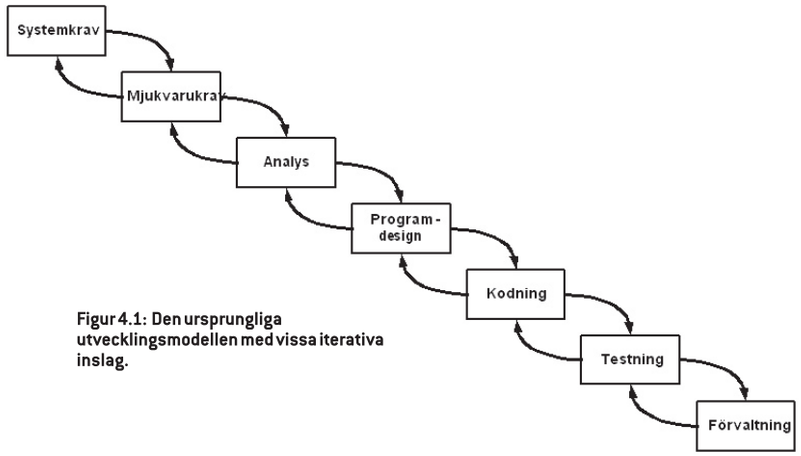
Den som numera kallas vattenfallsmodellen beskrevs första gången redan 1970 i en artikel av Dr Winston Royce. Benämningen vattenfall fanns inte vid denna tidpunkt utan har tillkommit senare. Vi måste komma ihåg att vid tidpunkten då artikeln skrevs var det helt andra förutsättningar som gällde för utveckling av programvara. Det vanligaste sättet att ta fram nya program var enligt Royce att analysera situationen och sen koda tills det fungerade. Detta fungerade för små fristående program men inte för de lite större.[15]

Det finns i artikeln ett flertal olika grafer som beskriver förslag på ut-

[14] Kit, Ed [1995]: Software Testing in the Real World, s 3
[15] Royce, Dr Winston [1970]: Managing the Development of Large Software Systems

vecklingsprocessen. Royce säger redan från början att den första enkla modellen har både risker och problem. Han förespråkar därför en modell med iterativa inslag och föreslår dessutom följande fem förbättringsåtgärder:

1. Övergripande programdesign ska ske innan analysen börjar.
2. Första regeln för styrning av mjukvaruutveckling är ett skoningslöst genomdrivande av dokumentationskrav.
3. Gör en pilot av hela projektet från början till slut.
4. Planera, kontrollera och övervaka test. De flesta fel är enkla och kan hittas genom visuell granskning. Testa alla vägar genom koden.
5. Engagera kunden.



Figur 4.1: Den ursprungliga utvecklingsmodellen med vissa iterativa inslag.

Studerar du modellen som den används idag är det tydligt att de som utgick från Royces ursprungliga artikel slutade läsa efter de första sidorna. Moderna tolkningar saknar helt de kritiska synpunkter som Royce tog upp i sin artikel.

Grundidén bakom det linjära angreppssättet är att det kostar mer att korrigera design och krav ju senare i projektet vi är. Därför lönar det sig att ha en väl genomtänkt design och tydlig målbild innan vi börjar koda. En tillräckligt bra dokumentation betyder att vi inte blir lika personberoende och att alla medlemmar i projektet har samma syn på vad som gäller. Många tycker också om det strukturerade och relativt enkla arbetssätt som presenteras. Större organisationer som NASA och USA:s försvarsdepartement och flygvapen har sina egna vattenfallsmodeller.

Argumenten mot modellen är att den bara fungerar i teorin. Vi kan aldrig helt avsluta ett steg innan vi går vidare till nästa. Det är också omöjligt att veta alla krav i detalj från första början. Svagheten är också att väldigt stor tyngdpunkt läggs på dokumentation vilket innebär att förändringar under projektets gång är mycket omständliga att genomföra.[16] Flertalet av de mer moderna utvecklingsmodeller som finns bygger på att processen delas upp i flera olika steg och att arbetet utförs i mindre etapper precis som Royce redan från början förespråkade.

4.1.1 Påverkan på test

Ett grundläggande problem för test är att testare kommer in väldigt sent i projektet och därför inte kan vara med tidigt och kvalitetssäkra. Risken är att vi får en test av typen *big-bang* – allt på en gång. En fördel med den gedigna dokumentationen är att vi får ett utmärkt underlag att arbeta från då både krav och design är noggrant granskat av andra projektmedlemmar.

4.2 V-modellen

När test började få mer fokus kom nästa version av utvecklingsmodell. V-modellen som är allmänt känd i testvärlden är även den livligt debatterad. Den kommer säkerligen att leva kvar i många år till och använd på rätt sätt ser jag den som väldigt användbar. Precis som alla modeller är den en förenkling av verkligheten och ingen exakt beskrivning av hur arbetet egentligen går till. Jag ser det som en konceptuell modell som visar relationen mel-

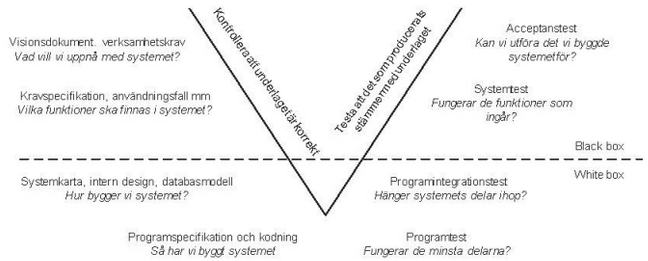
[16] McConnell, Steve [2004]: Code Complete, Second Edition

lan underlag och tester och hur testarbetet börjar med granskning av underlaget på vänster sida för att gå vidare med testutförande på höger sida. Den kan även skiktas i tre delar där den understa är kopplad till utvecklarna och deras programtester, och den mellersta till oss testare i projektet och våra systemtester. Den översta tillhör beställaren som använder acceptanstesten som ett sätt att godkänna leverans av det den beställt.

Ett vanligt misstag är att rita en tidsskala från vänster till höger och försöka följa den. Det ser jag som en alltför förenklad, oftast helt felaktig, modell av det komplicerade arbete det innebär att arbeta i ett projekt. Tänk på att det är en modell och som sådan är den en kraftig förenkling av verkligheten.

Som testare får du vanligen anpassa dig efter den metod ditt projekt väljer att arbeta efter. En riktigt bra testare analyserar varje ny situation och gör det bästa av den. En bra testare är en som tillför nytta genom sitt arbete och du blir som sådan oftast mycket uppskattad av de övriga projektdeltagarna.

Jag har sett åtskilliga varianter på V-modellen och vet inte vilken som ska räknas som den ”rätta”. Så länge du inte försöker lägga in mer i modellen än vad som finns där fungerar nedanstående version bra.



Figur 4.2: Enkel variant av V-modellen. Det centrala är att vi identifierar olika testnivåer samt att arbetet delas upp på förberedelse och utförande. Att lägga en tidsskala från vänster till höger fungerar inte speciellt bra.

Av praktiska skäl delar du in testarbetet i ett antal testnivåer från programtest till acceptanstest. Benämningen testfaser är också vanlig men enligt ISTQB-standarden heter det nivå, och fas är benämningen på en aktivitet som pågår under en viss tid. Hur många de är och hur mycket varje nivå innefattar beror på vilken omfattning projektet har. Varje nivå har olika underlag för testfallen även om vissa underlag används för flera nivåer. På bilden ser vi sambanden mellan testnivåer, arbetsmoment och de underlag som finns. Observera att underlagen kan användas för flera olika nivåer.

Om systemet som testas har många externa systemsamband kan du stoppa in en extra nivå mellan system- och acceptanstest som kallas systemintegrationstest. Hur exakt uppdelningen är beror på testningens formalitet. Jag har varit med om varianter från alla fem nivåer till en enda där vi slog ihop programintegrations-, system- och acceptanstest till en enda nivå som vi kallade systemtest.

Vänstra delen av V-et visar de olika underlag som tas fram i form av krav och design. Det finns en mängd olika varianter på vad de kallas så det finns ingen generell beteckning som alltid är korrekt. Som testdesigner är det viktigt att vara med tidigt för att verifiera alla underlag – speciellt kraven. Din uppgift är att kontrollera kvaliteten på det material du ska testa det färdiga systemet mot. Din granskning och analys är viktig som kvalitetssäkring av underlaget. Bäst blir det om även verksamheten kommer in tidigt i testarbetet.

Högra delen av V-et innebär att du utifrån de underlag som finns på vänstra sidan kontrollerar det som framställts genom att utföra testfall. Beroende på hur ditt projekt arbetar är du som testdesigner mer eller mindre involverad i de tidiga testnivåerna. Då dessa är väldigt tekniska och kräver stor delaktighet av utvecklarna är det av praktiska skäl inte ovanligt att du som testare kommer in tidigast i systemtesten. Jag har dock egen erfarenhet av att som testare aktivt delta i programtesterna med gott resultat.

4.2.1 Påverkan på test

V-modellen är bra genom att den sätter fokus på test och dessutom struk-

turerar upp arbetet i flera olika nivåer. Ett vanligt fel är att man försöker lägga in mer information i den än vad som hör hemma där. Risk finns att kopplingen mellan testnivåer och underlag tolkas som att endast en form av underlag används per testnivå vilket oftast inte är fallet. Elaka röster menar att det bara är ett annat sätt att rita vattenfallsmodellen, men då har man helt missat poängen med de olika testnivåer som V-modellen tillför och som helt saknas i vattenfallsmodellen.

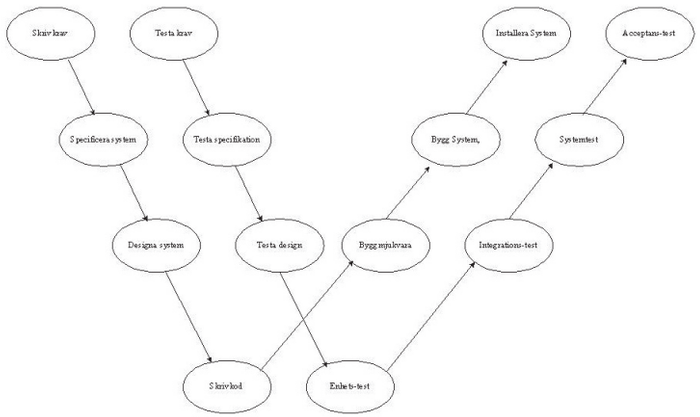
4.3 W-modellen

En ännu mer testcentrerad variant av utvecklingsmodell är *W-modellen* som introducerades av Paul Herzlich 1993.[17] En liknande modell är *The Dotted U-model* av Ed Kit.[18] Båda dessa modeller betonar att för varje aktivitet i utvecklingskedjan finns det en motsvarande testaktivitet. Målet med W-modellen är att ta hand om V-modellens största brister.

Den första bristen är att det sällan finns ett perfekt ett till ett-förhållande mellan utvecklingsstegen och dess dokumentation på vänstra sidan och testaktiviteterna på den högra sidan. Ta systemtest som ett exempel, där används både underlag i form av funktionskrav men även affärskrav och vissa delar av den tekniska designen. Den andra är att det inte står något om statisk testning. Fokus är på utförandet av tester på höger sida av V-et men inget sägs om granskningar av underlagen. Detta är en stor brist som vi måste hantera.

Istället för att fokusera på olika testnivåer, som V-modellen gör, fokuserar W-modellen på de faktiska utvecklingsprodukter som tas fram. Varje utvecklingsaktivitet har en motsvarande testaktivitet. Syftet med testaktiviteterna är att bestämma om leverabeln uppfyller sina krav. Det viktiga är att W-modellen specifikt fokuserar på produktriskerna vid den tidpunkt då test kan vara som mest effektiv. Ibland innebär detta granskning, ibland dynamiska tester.

[17] Herzlich, Paul [1993]: W-model, www.evolutif.co.uk/default.asp?page=services/wmodel.html
[18] Kit, Ed [1995]: Software Testing in the Real World



Figur 4.3: Den ursprungliga W-modellen för test. Varje utvecklingsaktivitet har en motsvarande testaktivitet. W-modellen fokuserar på produktriskerna vid den tidpunkt då test kan vara som mest effektiv. Ibland innebär detta granskning, ibland dynamiska tester.

4.3.1 Påverkan på test

W-modellen har större fokus på att hantera aktuella risker genom att aktivt arbeta med det underlag som tas fram, när det tas fram. Den är mer flexibel än V-modellen och tar bort det konstlade beroendet att ha samma antal dynamiska testnivåer som utvecklingsnivåer. Oavsett om kravframtagning, design och kodning görs i tre eller fem steg kan vi ändå ha tre testnivåer om det passar oss. Det finns inga krav på att vissa dokument som tagits fram ska användas för en viss testnivå även om så ofta är fallet. Den flexibilitet som finns gör att modellen passar nyare utvecklingsmodeller som RUP bättre. Du anpassar själv modellen efter hur många steg i designen och hur många testnivåer du vill ha.

4.4 Unified Process

Unified Process, UP[19], är ett arbetssätt som fått stort genomslag de senaste åren. Den variant vi vanligen kommer i kontakt med kallas Rational Unified Process som är en kommersiell tillämpning. Detaljerna kan skilja men grunderna är desamma. UP är uppbyggd runt sex praxis som ska ta hand om de

[19] Jacobson, Ivar, Grady Booch, James Rumbaugh [1999]: The Unified Software Development Process

problem man har i samband med utveckling av programvara. Dessa är följande: [20]

1. Utveckla iterativt – små delar, riskdrivet
2. Hantera krav – användningsfall, kravanalys
3. Använd komponentbaserad arkitektur - objektorienterat
4. Modellera programvara visuellt – grafiska modeller
5. Verifiera programvarans kvalitet kontinuerligt – tidig test
6. Hantera ändringar av programvaran – kontroll över förändringar

4.4.1 Faser i processen

Projekt i UP är uppdelat i faser som sätter fokus på olika saker, varje fas innehåller en eller flera iterationer. Varje iteration har leverans av något som är helt färdigt och i de flesta fall avslutas de med test (mer om detta i nästa avsnitt). Grunden är att hela processen fokuserar på risker men det är olika typer av risker som hanteras i de olika faserna. *Inception* (uppstart) går ut på att förstå den vision som ligger bakom varför detta nya system behövs. Projektet ska etableras och deltagarna förstå problemet. *Elaboration* (förfining) bygger systemets grund och har fokus på arkitektur. I denna fas hanterar man de viktigaste riskerna som har med arkitekturen att göra. C*onstruction* (konstruktion) är den fas då systemet utvecklas. Den viktigaste beslutspunkten ligger mellan *elaboration* och *construction*. Här är det fokus på att bygga funktionalitet och få en bra kvalitetsnivå. I *transition* (övergång) utvecklas ingen ny funktionalitet utan här är fokus att leverera och uppnå den slutgiltiga kvaliteten mot slutanvändaren.

4.4.2 Påverkan på test

Testarbetet följer med och får olika former beroende på fas. Inception innehåller i de allra flesta fall inte någon exekvering av testfall. Test av arkitektur i elaboration har som mål att bevisa att arkitekturen håller. Viss funktionalitet testas men det handlar om att bevisa att det fungerar, inte att nå fulländad kvalitet. Construction har mer av det klassiska testarbetet i sig men uppdelat i iterationer. I transition finns acceptanstest även om RUP är väldigt tunt på det. Alla iterationer i faserna elaboration, construction och transition skapar något exekverbart och därför testbart.

Det är viktigt att påpeka att det iterativa arbetet betyder väldigt mycket för test. RUP ska konfigureras efter behoven men väljer man bort iterationer så försvinner det som test tjänar mest på. Använt på rätt sätt gör detta arbetssätt att kvalitetstänkande och styrning genomsyrar varje projekt. Tidiga tester av små delar i taget gör att vi inte bara har möjlighet att testa kodens riktighet utan även programvarans ändåmålsenlighet, d v s om den löser de problem vi har.[21] Detta synsätt kan jämföras med W-modellen som har en testaktivitet för varje utvecklingsaktivitet. Kravhantering, analys och design är centrala delar som betonas i processen, detta ger oss som testare ett utmärkt underlag i form av olika modeller att bygga testfall från.

En effekt som gör att vi får mer arbete inom test är att mängden regressionstester ökar då leveransen i nya iterationer måste testas ihop med alla tidigare delar. Automatisering förordas av många men kan bli svårt om de äldre delarna ändras och automatiseringsunderlaget måste uppdateras mycket eller helt göras om.

4.5 Agile

Det finns en paraplymetodik kallad *Agile Alliance* [22]som samlat en uppsättning värderingar, attityder och principer. Under detta paraply finns ett antal olika systemutvecklingsprocesser som omfattar och realiserar dessa värderingar. *Agile* sätter människan i centrum för att kunna stödja systemutveckling utan att skapa onödigt arbete. Kärnpunkten är att tillåta utvecklingsteamet utveckla snabbt och samtidigt hantera förändringar genom att

[20] Kruchten, Philippe [2002]: Rational Unified Process – En introduktion, s 5-16

[21] Test av ändamålsenlighet kallas också validering.
[22] www.agilealliance.org

ha en flexibel process. Framväxten av *Agile* har skett som en reaktion på de tunga systemutvecklingsprocesser som upplevts som ett hinder snarare än ett stöd.

Några kända processer är eXtreme Programming av Kent Beck, Scrum av Ken Schwaber, Lean Software Development av Mary Poppendick, DSDM m fl. Agiles systemutvecklingsprocesser har olika karaktär och därför används med fördel flera olika processer tillsammans som kompletterar varandra, t ex Scrum och eXtreme Programming - XP.

I vårt sammanhang översätts begreppet *Agile* med svenskans lättrörlig. Ordet lättrörlig gäller både den miljö eller omgivning vi befinner oss i och det sätt vi arbetar på. Alla systemutvecklingsprocesser påstår sig adressera problemet med påverkande förändringar. Skillnaden är att *Agile* välkomnar förändringar vilket inte innebär att den förespråkar oordning och kaos. Tvärtom är metodiken ofta mer styrande än i traditionella systemutvecklingsprocesser då det behövs en stark kvalitets- och testmetodik för att kunna välkomna förändringar.

4.5.1 Påverkan på test

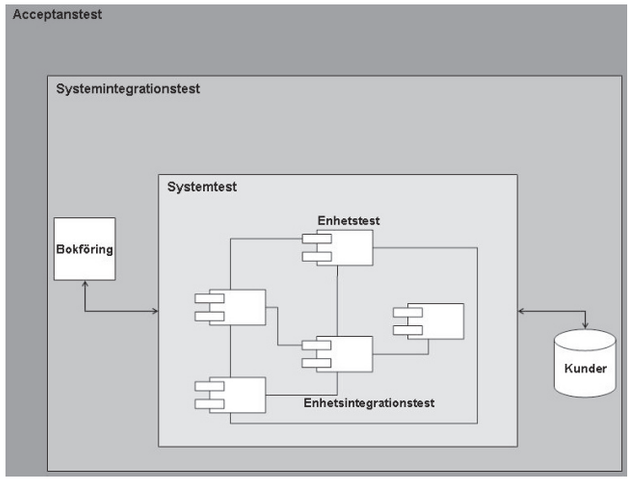
De agila utvecklingsprocesserna har haft stor framgång med att lyfta fram användbar metodik för test och kvalitetssäkring. *Extreme Programming* använder sig exempelvis av *testdriven utveckling* vilket innebär att kod för programtest skrivs innan kodning av programmet påbörjas. [23] Ingen kod får släppas innan lyckade enhetstester genomförts. Regressionstester körs regelbundet då leveranserna kommer tätt och acceptanstester avgör om leveransen är klar eller ej. Det här innebär att ett väl genomfört XP-projekt inte bara levererar ett system med god kvalitet utan även kod för enhetstester samt acceptanstestmaterial för hela systemet. Förvaltningen får därmed en stabil bas för fortsatt utveckling av systemet. Testarna tar över kod som är noggrant enhetstestad och därmed lättare att jobba vidare med.

[23] Beck, Kent [2003]: Test-Driven Development

5 Angreppssätt

5.1 Testnivåer

Här beskrivs den uppdelning av test i nivåer som används i standarden som tagits fram av ISTQB.[24] Det finns inget krav på att du alltid ska använda alla nivåer av test i alla projekt du jobbar med, men det är av praktiska skäl bra att ha åtminstone ett par olika nivåer.



Figur 5.1: Uppdelning i testnivåer. Enhetstest utförs av utvecklarna, systemtest av utvecklingsprojektets testare och acceptanstest av den som beställt systemet. Varje nivå har olika fokus.

[24] ISTQB Kursplan version 0.2. www.sstb.se

5.1.1 Enhetstest

Enhetstestning, även kallat komponent-, modul- eller programtest, syftar till att kontrollera de minsta delar som är testbara separat. För detta används ofta hjälpmedel i form av stubbar, drivers[25] eller någon form av verktyg.[26] Testerna är av teknisk karaktär och kräver kunskap om programspråk. Därför utförs de vanligen som en del av utvecklarnas kodningsarbete.

Som utgångspunkt för testerna används program- och designspecifikationer. Testtekniker som hör hemma här är kontrollflöde och dataflöde.

5.1.2 Enhetsintegrationstest

Enhetsintegrationstestning verifierar gränssnitt mellan komponenter i ett system. Även denna nivå är av teknisk karaktär och handlar om att bygga ihop komponenterna som förberedelse inför systemtesterna. Denna nivå räknas till de strukturella testerna och fokus är på att den interna logiken fungerar.

5.1.3 Systemtest

Nu testas systemet som en helhet utifrån användarnas perspektiv. Både funktionella och icke-funktionella aspekter ingår. Vi gör nu främst *beteendebaserade* tester som innebär att fokus är på yttre funktionalitet snarare än programvarans inre struktur.

5.1.4 Systemintegrationstest

Om vårt system kommunicerar med andra system kan det vara bra att dela upp systemtesterna i en extra nivå. Detta för att först kontrollera att systemet fungerar för sig för att sedan kontrollera alla externa kopplingar som finns. Exempel på externa system är huvudbok, reskontra, kundregister, aktiehandel, kreditkortsbetalning och upplysningscentralen.

[25] Kod som används för att exekvera annan kod
[26] Exempelvis XUnit verktygen, J-Unit för Java, N-Unit för C# och .Net

5.1.5 Acceptanstest

Acceptanstesten är ett sätt för beställaren att godkänna det som levererats. Enkelt sagt kan man säga att om inte systemet löser de problem det byggdes för har vi inte lyckats. Denna nivå är ofta en del av en affärsuppgörelse mellan beställare och leverantör och är som sådan mycket viktig. Acceptanstest kan också vara ett sätt för en driftsorganisation att godkänna produktionssättning.

För att betona vikten av acceptanstesten kommer här två exempel.

Exempel 1: Företag A beställer ett system för bokning och administration av biljetter. Uppdraget utförs av en stor konsultfirma till fastpris. Fastpris betyder att leverantören gör minsta möjliga arbete för den summa den fått. Så fort kunden godkänt leveransen slutar den att vidareutveckla systemet. Alla ändringar efter godkännande går på löpande räkning. I detta fall är acceptanstesterna obefintliga och företag A upptäcker därför inte att systemet vid leveranstidpunkten har stora kvalitetsproblem. Dessa får korrigeras i efterhand på löpande räkning och slutnotan blir dubbelt så hög som den ursprungliga beräkningen. Hade företag A gjort en ordentlig acceptanstest hade konsultfirman tvingats utföra alla de rättningar som krävdes för att systemet skulle uppfylla kravspecifikationen.

Exempel 2: Bolag B beställer ett nytt administrativt system för administration av försäkringar, även detta till fastpris. Konsultbolaget som åtar sig uppdraget lämnar ett pris som är lägre än sina konkurrenter och vinner därför matchen. Då anbudet är så mycket lägre än övriga beslutar sig bolag B för att verkligen kontrollera att de får vad som beställts. De engagerar en driven testledare som sätter ihop en grundlig acceptanstest med stöd av en grupp verksamhetsexperter. Vid de

första tre leveranserna visar acceptanstesterna att systemet inte klarar av att uppfylla de krav som finns i avtalet och går därför i retur till leverantören. Först efter stora förseningar lyckas leverantören få sitt system godkänt. Den insats som lades ner på acceptanstest hos företag B sparade många miljoner på slutnotan.

5.1.6 Testnivåer i produktion

Den sista verifieringen innan du går i produktion med systemet är en kontroll av att systemet fungerar:

- med de volymer data som kommer att finnas
- i den miljö det är tänkt att köras
- med de kopplingar som finns till andra delar
- tillsammans med andra system som körs i samma miljö

Använd en miljö som är så produktionslik som möjligt, helst den slutliga miljön om det finns möjlighet. Är det möjligt använder du produktionsdata under kontrollerade former, annars skapar du fejkat data av samma storlek som systemet ska klara av att hantera. Om du ska sälja applikationen på den öppna marknaden bör du ha med uppgifter om vilken prestanda det har och i vilka miljöer det fungerar. All dokumentation om installation och handhavande ingår.

Om du använder produktionsdata i någon fas, tänk då på att denna typ av test ställer stora krav på säkerhet och integritet och måste utföras under kontrollerade former tillsammans med driftpersonal och kanske även specialister på belastningstest. Generellt sätt är det förbjudet att använda riktiga data i testsituationer men ett flertal företag hamnar ofrivilligt i rampljuset varje år på grund av att de inte bara använt data på ett otillåtet sätt utan att de även råkat skicka ut resultatet av testerna till riktiga kunder. Jag avstår av juridiska skäl från att lämna några referenser.

Ibland kör du något som kallas pilotproduktion för en begränsad mängd slutanvändare. Detta är ett sätt att kunna köra på riktig produktionsmiljö och riktig data och ha slutanvändare som utför arbetet.

Områden där du ofta hittar fel är:

- svarstiderna mot slutanvändaren är för långa
- batchprocesser som är dagliga, månatliga eller årliga tar för lång tid
- minne eller databaskapacitet är otillräckligt
- parametrar för kopplingar utanför systemet skiljer sig mellan test och produktion
- datanätverket klarar inte av belastningen
- systemet störs av, eller stör själv, andra system

För de delar som har med belastning att göra krävs ofta ett verktyg. I dessa fall tar du in specialister på området som utför dessa tester enligt de specifikationer på användning som testgruppen tagit fram. Tänk på att den som ska utföra testet måste veta vilken typ av användargrupper som finns och hur många av varje grupp som beräknas vara samtidiga och vilka funktioner de kommer att använda. Detta kallas för att ta fram användarprofiler. Inte minst måste de veta vad de ska mäta och helst vilka krav på svarstider som finns! I annat fall blir det mer ett konstaterande av att ”under dessa förhållanden har vi de här svarstiderna, duger det?”.

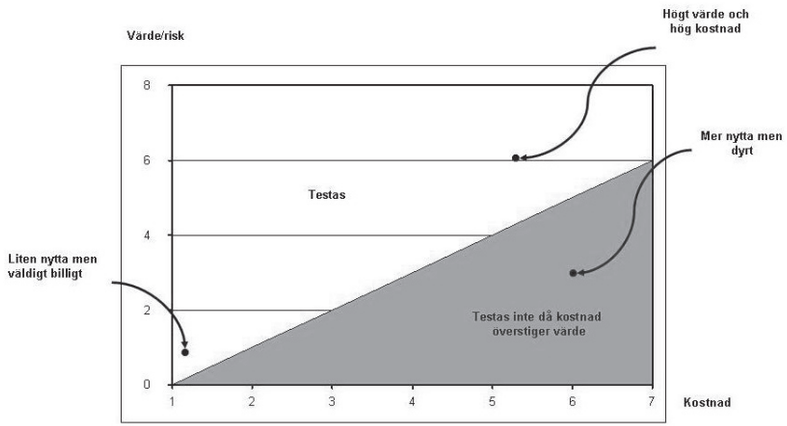
Det finns ett antal andra aktiviteter som av vissa benämns tester. En av dem är att en applikation körs i en *pilot*. Detta innebär att vi arbetar på riktigt men med ett begränsat antal användare som vet att de är de första som använder systemet. Syftet är att verifiera verklig användning i produktionsmiljön.

En annan variant är det som kallas betatester. Om vi säljer en produkt kommersiellt kan vi välja att sprida en tidig version för att få slutanvändarnas synpunkter och hinna åtgärda eventuella tankefel och teknikmissar som visar sig först i verklig användning.

5.2 Vilka tester ska vi göra?

Ett förslag till angreppssätt är att utgå från vad en test är värd i förhållande till dess kostnad där värdet står i direkt relation till risken. Detta betyder att ett testfall med relativt liten nytta för oss är värt att köra om det är tillräckligt billigt. [27]

Samma princip används då vi ska besluta vilka krav som ska ingå i en viss release/iteration av ett projekt. Detta gäller oavsett enligt vilken metod vi valt att utveckla. Tankesättet är ett komplement till ett helt riskbaserat angreppssätt.



Figur 5.2: Kostnad ställs mot värde när vi väljer vilka tester vi ska göra. Tester av begränsat värde kan utföras om kostnaden för dem är väldigt liten.

5.3 Allt skrivet och planerat i detalj i förväg

Det traditionella angreppssättet som är rotat både i vattenfalls- och V-modellen innebär att ett projekt är linjärt och att en aktivitet avslutas helt innan nästa börjar. Detta innebär att samtliga krav är helt klara innan kodningen börjar och att testerna börjar då kodningen är klar. Detta betyder också att alla testfall ska vara helt klara inklusive förväntat resultat innan testutförandet börjar. Detta traditionella sätt kallas ofta för *skriptad testning* för att skilja det från *utforskande testning (*eng *exploratory testing)*. De uppenbara problemen som finns med skriptad testning i verklig tillämpning har lett till stor debatt i testarvärlden mellan de som anser att alla testfall inklusive förväntat resultat kan och ska vara helt färdiga innan testerna körs och de som menar att vi aldrig kan veta allt vi behöver i förväg. Utifrån detta föddes begreppet utforskande testning som innebär att testdesignen pågår parallellt med utförandet. Detta beskrivs ingående i nästa kapitel.

[27] Bach, James [2005]: Rapid Testing, kursmaterial

Ett antal anledningar till att du ska skriva ner all information i förväg är:

1. Legala eller regulatoriska krav exempelvis inom flygindustri, medicinteknik eller annan säkerhetskritisk verksamhet
2. Komplexa samband som gör att du måste förbereda testdata i detalj
3. Den som ska utföra testfallen måste ha informationen i detalj
4. Du bedömer det som mest effektivt att skriva ner mycket i förväg
5. Du vet tillräckligt mycket om hur det färdiga systemet kommer att se ut för att kunna designa effektiva testfall
6. Du behöver testfallen som historik eller för regressionstest

Alla verksamheter som hanterar ”liv- och dödrelaterade” tjänster och produkter, där effekten blir stor om ett dödsfall inträffar på grund av kvalitetsbrister, har kvalitet som det viktigaste styrande kravet på produkt- och tjänsteutvecklingen. Det gäller branscher som flyg, rymdindustri och läkemedel. Inom läkemedelsbranschen är hotet av en kontroll av amerikanska FDA stort. Risken att få sin försäljningslicens indragen på den största marknaden (USA) vid anmärkning innebär att alla kvalitetshöjande insatser måste dokumenteras för att bevisa vad man gjort. Många jag pratat med menar att den enorma mängden administration betyder att en väldigt stor del av resurserna fokuseras på administration och dokumentation och därmed blir testerna i realiteten sämre än med ett mindre formellt angreppssätt. Självklart kan de väl dokumenterade testerna även kompletteras med utforskande tester för att minimera riskerna ytterligare vilket också sker.

5.4 The Context-Driven School

Ett intressant synsätt på test har den sammanslutning som kallar sig *The Context-Driven School*.[28] Grundtanken är att hur du testar beror på sammanhanget och det finns stora likheter med *Agile*-utveckling. Det finns sju grundläggande regler:

1. Värdet av varje praxis beror på dess sammanhang.
2. Det finns bra praxis i sitt sammanhang men det finns inga bästa praxis som alltid gäller.
3. Människor som arbetar tillsammans är den viktigaste delen av varje projekts sammanhang.
4. Projekt utvecklar sig över tiden på sätt som ofta inte går att förutsäga.
5. Produkten är en lösning, om problemet inte är löst så fungerar inte produkten.
6. Bra mjukvarutestning är en utmanande intellektuell process.
7. Bara genom omdöme och skicklighet, utfört tillsammans genom hela projektet, kan vi göra rätt saker vid rätt tidpunkt för att effektivt testa våra produkter.

Påföljder av ovanstående:

Det är lönt att producera olika resultat inom test i den mån det tillfredsställer intressenternas relevanta krav. Det vill säga att varken dokument eller annat tas fram för sakens egen skull om det inte ger ett mervärde.

Olika typer av tester hittar olika sorters fel. Om ett program verkar stabilt under en uppsättning tekniker är det dags att byta tekniker.

Snarare än att se automatisering som ett sätt att kapa kostnader genom att en maskin på ett imperfekt sätt gör vad en människa gör, se det som ett sätt att utföra saker som vore svårt att åstadkomma manuellt.

Anhängare av denna sammanslutning använder sig ofta av utforskande testning.

[28] www.context-driven-testing.com

5.5 Den gyllene medelvägen

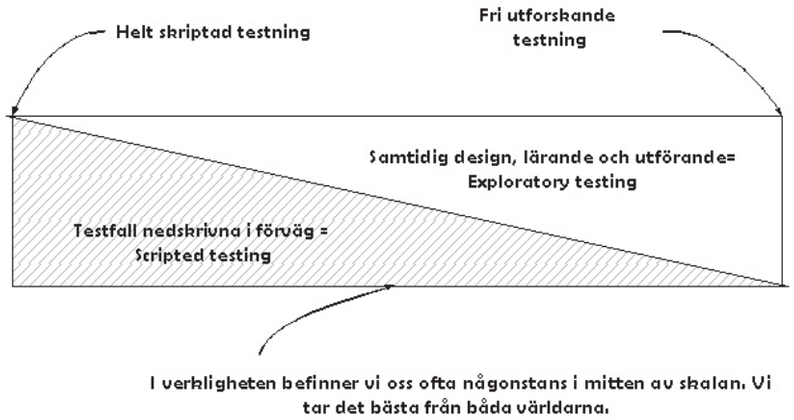
Det kan finnas lagar och förordningar som kräver en formell dokumentation, exempelvis inom medicinteknikindustrin. Det kan finnas praktiska krav på detaljplanering, som väldigt många gränssnitt och sammanhängande testdata utspritt över flera system. I dessa fall planerar och dokumenterar vi noggrant i förväg.

Andra situationer innebär att vi inte får levererat det vi förväntat oss, ändrad eller ny funktionalitet kan ha smugit sig in. Det kan uppstå situationer som vi inte riktigt har förberett oss för - då tar vi till den utforskande testningen. Eftersom fel är sällskapliga av sig finns det ofta fler i närheten då du hittar ett. Det arbete du gör med att isolera felet och undersöka om det finns fel i närheten är inte nedskrivet i testfallet utan du improviserar. Det kan ses som ett exempel på utforskande testning.

Jag har haft diskussioner med mycket erfarna testare av kritiska system inom medicinteknik som utför mängder av noggrant dokumenterade tester på grund av legala krav. Med mycket gott resultat kompletterar de med utforskande tester. Hur bra de än designar de förberedda testfallen finns det ändå avvikelser som hittas först i den mer fria delen.

Vi har tidigare kommit fram till att variation är bra både vad det gäller testdesign och vilka som deltar. Det verkar även gälla angreppssättet till test där vi med fördel kombinerar grundligt förbereda testfall med en mer utforskande approach. Så i verkligheten genomför vi nog alla både nedskriven och utforskande testning i varje projekt. Vi befinner oss någonstans på skalan i grafen nedan.[29]

[29] Bach, James [2005]: Rapid Testing, kursmaterial



Figur 5.3: Oftast består testerna av en blandning av skriptad och utforskande testning. Vissa delar är nedskrivna och andra kompletterar man med efterhand.

6 Utforskande testning

Hela detta kapitel är en översättning av artikeln Exploratory Testing Explained 2.0 1/6/06 av James Bach och publiceras med författarens godkännande.

Copyright © 2002-2006, James Bach

Utforskande testning är ett kraftfullt men ofta missförstått angreppssätt. Min erfarenhet är att det kan vara många gånger mer produktivt än så kallad skriptad testning där testfallen är detaljerat nedskrivna i förväg. Alla som skapar tester gör någon form av utforskande testning även om många inte inser det. Få av oss studerar detta angreppssätt och det får inte speciellt mycket respekt inom vårt område. Denna attityd håller på att förändras då företag letar efter mer lättrörliga och kostnadseffektiva metoder att utveckla programvara.

Det är svårt att förklara något för människor som tror de redan vet hur det ligger till. Vi kan alla lyssna, läsa, tänka och berätta anekdoter om händelserna i vårt liv, som vuxna gör vi dessa saker dagligen. Men *nivån* på dessa kunskaper hos den genomsnittliga personen kanske inte är tillräcklig för vissa situationer. Psykoterapeuter måste vara *experter* på att lyssna, advokater *experter* på att läsa; forskande vetenskapsmän måste noggrant fjärma sitt tänkande från fel och journalister rapportera berättelser som överträffar sällskapsanekdoter.

Samma sak gäller för utforskande testning. Detta är ett enkelt koncept, men det faktum att det kan beskrivas med en mening kan få det att verka vara något som inte är värt att förklara. Den mycket situationsberoende strukturen kan för den tillfällige betraktaren få det att verka som om det inte finns någon struktur alls. Det är med anledning av detta som böcker om programvarutestning antingen inte beskriver utforskande testning alls eller diskuterar det som en ovärdig utövning.

6.1 Definitionen av utforskande testning

Jag har provat diverse olika definitioner av utforskande testning. Den som framträtt som favorit hos mina kollegor är denna:

Utforskande testning är samtidigt lärande, testdesign och testutförande.

Med andra ord all testning där testaren aktivt kontrollerar designen av tester under tiden de utförs och använder information som framkommer under utförandet för att skapa nya bättre tester. Att definiera utforskande testning är lätt, precis som att jonglera är lätt att definiera. Att själv göra det på ett bra sätt är svårare.

Har du någonsin lagt ett pussel? I så fall har du gjort en form av utforskande test. Tänk på vad som händer när du lägger pusslet. Du plockar upp en bit och försöker passa ihop den med någon av de delar som ännu har lediga sidor. Varje titt på en ny bit är ett testfall. ("Passar den här biten med den där? Inte? Om jag vänder på den? Ja, den passar nästan formen men bilden blir fel…") Istället för att leta efter problem letar du efter kopplingar men annars är det samma sätt att tänka. Det är relativt simpelt att testa om två bitar passar ihop men valet och följden av vilka bitar du plockar upp är inte lika lätt. Det finns inget säkert sätt att veta när du börjar pusslet vilka "tester" du ska göra och i vilken ordning.

Men det är inte helt slumpvisa val heller. Det är här skickligheten kommer in. Du siktar in dig på form, färg eller mönster. Du kanske först sorterar upp bitarna i små högar. Du kan vända alla med rätt sida upp. Om du kommer på att du har en tillräckligt stor mängd bitar som hänger ihop kan du flytta dessa till rätt plats inom ramen för att se var de passar in. Du kan känna att det blir rörigt ibland och då det inträffar kan du ta ett steg tillbaka för att analysera situationen och skapa en mer specifik plan. Om du arbetar med en typ av "testning" ett tag (att exempelvis försöka passa ihop kantbitarna), kan du byta till en annan typ för att hålla intellektet fräscht. Även om du inte anvisas en stegvis procedur för exakt hur du ska göra är processen att lägga ett pussel ändå systematisk – precis som designen av programvarutester.

Notera hur processen *flödar* och hur den *hela tiden* är under kontroll av den som utför arbetet. Lägg märke till att idéerna som styr den inte kommer från någon som berättar för dig hur du ska göra. Du följer inga skrivna instruktioner. Nya idéer och strategier kommer till dig då mönstret i pusslet kommer fram och dessa idéer kommer från dig själv. Varje pusselbit du lyckas placera betyder att det finns en bit mindre kvar i högen vilket gör resten av processen lite lättare. Pusslet ändrar sig under tiden du löser det.

Om du lägger samma pussel flera gånger kommer din bekantskap med bilden och bitarna låta dig lägga det fortare. Du kommer att ha en bättre idé om hur du ska sortera bitarna, du kan till och med beskriva processen i detalj genom att märka varje bit med en identitet (åtminstone ett pussel jag sett har nummer på baksidan av varje bit). Men första gången lär du dig under tiden du arbetar. Du kan inte undvika att lära dig om du vill lösa pusslet.

Om du redan i förväg skulle veta exakt var varje bit passar in vore det inte längre ett spel utan en tillverkningsprocess. Du kunde då ha en metod för att sätta ihop det mer effektivt men då skulle det inte längre likna en testprocess. I test vet jag inte när jag börjar var felen finns, för det mesta vet jag inte heller var riskerna finns. När jag utför en test försöker jag lista ut vilken nästa test ska vara. De specifika delarna av det pusslet som framkom-

mer under utförandeprocessen påverkar mitt val av kommande tester. Denna återkopplingsdynamik - tänka lite, testa lite, tänka lite mer, testa lite mer - är hjärtat av alla utforskande undersökningar oavsett om det gäller test, utveckling, forskning eller detektivarbete.

Vi gör väldigt många saker på ett utforskande sätt. Att köra en bil är utforskande, du kör inte genom att läsa exakta instruktioner för varje moment. Att ha en konversation är utforskande, du går inte omkring med ett förberett samtal för varje tillfälle, eller hur? Mycket av matlagning är utforskande liksom de flesta spel. Hur lång vill du att listan ska bli? Om du är människa är du en varelse av utforskande natur. Släpp loss din inre utforskare och låt den jobba med test.

6.2 Andra benämningar på utforskande testning

Du kanske redan känner till utforskande testning under ett annat namn. Processen att samtidigt lära sig, designa och utföra tester kallas ibland gerillatestning, gorillatestning, aptestning eller betatestning. Jag har också hört det kallas ostrukturerad eller slumpvis testning vilket är väldigt missvisande då den varken är ostrukturerad eller slumpvis! En mer vanlig term i litteraturen är *ad hoc- testning.* Ad hoc är en godtagbar benämning. Från orden *"till detta"* på latin, betyder det "formad, arrangerad eller gjord enbart för ett särskilt syfte" (till skillnad från en pågående rutin). Olyckligtvis är ad hoc ofta kopplat till att betyda ett slarvigt utfört arbete. Bara genom att yttra orden *ad hoc* verkar konversationen avstanna.

Andra populära termer dyker upp i organisationer som tror de utför enbart skriptad testning. I dessa organisationer är oplanerade tester inkluderade i aktiviteter som kallas felanalyser, testutveckling, testdesign eller produktutbildning.

Tidigt på 1990-talet började en grupp metodare (som nu kallar sig *the Context-Driven School*, *context* = sammanhang) använda termen utforskande som benämning på de tester som inte var nedskrivna i förväg. Med denna terminologi, först publicerad på sent 1980-tal av Cem Kaner i boken *Testing Computer Software*, försökte vi betona den tankeprocess som dominerar den testning som inte är nedskriven i förväg och börja utveckla den som en disciplin möjlig att lära ut.

Utforskande testning kan sannerligen vara lika disciplinerad som vilken annan intellektuell aktivitet som helst. Microsoft använder en formaliserad process för utforskande testning för att certifiera tredjepartsprodukter för Windows-kompatibilitet [30], och sessionsbaserad teststyrning [31] är en metod specifikt designad för att göra utforskande testning reviderbar och mätbar på en större skala.

Även om vår ursprungliga motivation var att undvika bagaget att ha en *ad hoc*-etikett är utforskande inte bara en skönskrivning. Vi använder ordet som det beskrivs i ordboken, applicerad på test. Vår användning av termen stämmer också med koncepten hos utforskande dataanalys, utforskande forskningsstudier och utforskande försök. Vår användning speglar särskilt användningen av termen i geografin, då upptäcktsresanden i Royal Geographic Society under 1700- och 1800-talen brottades med samma metodfrågor.

> *Så för att kvalificera som utforskande skulle en resa vara trovärdig, innehålla umbäranden och risk och måste innehålla nyheten av en upptäckt. Därefter var den, precis som cricket, svår att förklara för de som inte var insatta. Men ett element var absolut nödvändigt; sannerligen var det just vad som skilde utforskandets era från tidigare upptäckandes eror och gjorde det nödvändigt att adoptera beteckningen utforskande. Det var, enkelt sagt, en vördnad för vetenskapen.* [32]

[30] www.satisfice.com/tools/procedure.pdf. [31] www.satisfice.com/sbtm
[32] Keay, John [1994]: The Permanent Book of Exploration

6.3 Se på det som ett angreppssätt - inte en teknik

Det vi kallar testtekniker är specifika heuristiker för att designa och utföra tester. Utforskande testning är inte en testteknik utan mer ett angreppssätt. Detta är en viktig praktisk åtskillnad eftersom utforskande testning inte lever för sig själv ensam eller skild från testtekniker. Alla testtekniker jag känner till kan användas på ett utforskande sätt. Att sätta det på en lista över tekniker vore förvirrande, som om alla andra tekniker, som ekvivalensgruppering eller gränsvärdesanalys, inte skulle kunna utföras på ett utforskande sätt. Ett annat angreppssätt är det vi kallar *skriptad* testning vilket är motsatsen till utforskande testning.

6.4 Utforskande kontra skriptad testning

När jag säger skriptad test menar jag en uppsättning instruktioner för att utföra ett testfall. I ren skriptad testning är testfallen helt specificerade innan de körs. Testerna ändras inte under själva utförandet, inte heller ändras de rutinmässigt baserat på nytt lärande som kan ske under utförandet av testerna.

Det kan vara knepigt att prata om skriptad testning, speciellt då man jämför det med utforskande testning, eftersom termen skriptad testning ofta används för att beskriva testutförande utan hänsyn till testdesign. Men när vi jämför dessa tillvägagångssätt måste vi inkludera hela processen att lära sig om produkten och bygga och avlusa testprocedurerna. Testdesignprocessen måste ske på något sätt, annars kommer det inte fram några testprocedurer. Även om skriptad testning är motsatsen till utforskande testning utesluter tillvägagångssätten inte helt varandra. Det är mer användbart att se dem som motsatta ändar på en skala. De flesta tester i verkligheten är en blandning av ett skriptat och ett utforskande sätt att arbeta.

De båda tillvägagångssätten kan faktiskt förbättra varandra. Samma test kan vara utforskande på ett sätt och skriptad på ett annat. Ett helt skriptat testfall informerar testaren vad denne ska göra och observera. Men om ett testfall säger till dig exakt vilken indata du ska ge till programvaran utan att berätta för dig hur du ska veta om en avvikelse inträffat måste testen lita på att du själv har den informationen. Du måste improvisera. Många av de skriptade tester jag granskar innehåller inte så mycket detaljer. Somliga består bara av en kort mening som exempelvis *"prova långa inmatningar"* vilket kräver betydande bedömningsförmåga och bakgrundskunskap för att utföra dem som verkliga tester.

Eftersom tillvägagångssätten ligger på var sin ände av en skala kan frågan "Gör du utforskande testning?" bättre omformuleras som "På vilka sätt och till hur stor del är din testning utforskande eller skriptad?" Den kunniga testaren vet hur han ställer och besvarar denna fråga.

6.5 Hur känner man igen utforskande testning?

Om lärande, testdesign och testutförande överlappar varandra är testningen utforskande i den mån de överlappar varandra. En komplett överlappning innebär ren, utforskande testning som är helt fri (eng *freestyle exploratory*). I praktiken är ett bra sätt att upptäcka utforskande testning att fråga var testdesignen kommer från. Om den kommer från en annan person än den som utför testen är det inte en utforskande process ur den personens synvinkel. Om testaren kontrollerar vissa men inte alla delar av testdesignen är det en delvis utforskande process.

Det är viktigt att förstå vilka sätt man *inte* kan identifiera en utforskande process genom: frånvaron av testdokumentation (testaren kanske följer någon annan persons muntliga anvisningar, eller följer en vana eller ett minne av en gammal test som aldrig skrevs ner), frånvaron av stränghet i testdesignen (skriptade tester kan vara baserade på idéer som är vaga, grunda och tvetydiga medan utforskande testning kan drivas av största möjliga sträng-

het) eller att testerna ej kan återupprepas (bara för att någon säger åt dig att göra något betyder detta inte att du kan eller kommer att upprepa samma sak om de ber dig igen). Brist på dokumentation, stränghet och möjlighet att återupprepa kan karakterisera såväl skriptad som utforskande testning.

Enligt definitionen är det relationen mellan tankeprocesserna i lärande, testdesign och testutförande som karakteriserar utforskande testning. Eftersom det är en oavbruten skala menar jag då jag använder termen utforskande testning utan att karakterisera det, testning som ligger närmare helt utforskande än helt skriptad testning. När jag vill prata om testning som är helt eller väldigt nära helt utforskande testning, utan något skrivet eller externa instruktioner över huvud taget, kallar jag detta fri utforskande testning.

6.6 Varför prata om utforskande testning?

Jag säger ofta att *alla* testare gör utforskande testning på något sätt. Detta får vissa att undra varför jag förespråkar detta tillvägagångssätt. Varför förespråka ett sätt att testa som alla redan gör? Sanningen är att jag inte förespråkar att man ska göra utforskande testning utan hur vi kan lära oss att bli bra på att göra det. Alla gör det, ja, men de flesta gör det illa. Få kan förklara eller försvara kvaliteten på sitt arbete. Få kan lära ut utforskande tekniker. Detta är inte ett svårt problem att lösa men ett som vår bransch har ignorerat alltför länge. Istället finns det en populär fantasi att all testning ska beskrivas som tankar på burk nedskrivna på papper. Skriptade tester är till stor hjälp när de används inom rimliga gränser. Så är typiskt inte fallet utan det har istället blivit en farlig besatthet inom programvaruindustrin.

Vår näringsgren är märkligt förtegen när det gäller att acceptera att testning är en produkt från vårt intellekt och att det är möjligt att träna våra intellekt till att testa bättre och prata tydligt om hur våra hjärnor fungerar. Jag tror inte att det bara är möjligt utan att det också är nödvändigt. Utmärkt intellektuellt arbete kräver att vi tittar på detaljer hos våra tankeprocesser. Vi kan inte systematiskt uppnå förträfflighet i vår testning om vi inte gör det.

För att bli bra på utforskande testning måste vi först erkänna att det finns och lära oss hur vi känner igen det. Men varför ska vi bli bra på utforskande testning alls? *Eftersom det är ett fantastiskt produktivt tillvägagångssätt till test*. Prova det, jämför det noga med skriptade tester och jag är övertygad om att du håller med.

6.7 Utforskande testning från utsidan

Den synbara strukturen hos utforskande testning är lätt nog att beskriva. Under en *tidsperiod* interagerar en *testare* med en *produkt* med avsikt att slutföra ett *testuppdrag* och sen rapportera *testresultatet*. Där har du de grundläggande synbara delarna av utforskande testning: tid, testare, produkt, uppdrag och rapportering. Uppdraget fullförs genom en kontinuerlig cykel av att anpassa oss till uppdraget, tänka ut frågor om produkten som om de besvaras också låter oss fullfölja vårt uppdrag, designa tester för att besvara dessa frågor och utföra tester för att få fram svaren. Ofta besvarar inte våra tester helt frågorna så vi justerar dem och fortsätter våra försök - med andra ord utforskar vi. Vi måste vara beredda att rapportera vår status och våra resultat när som helst.

En utforskande testomgång börjar ofta med en övergripande beskrivning (eng *charter)* av ett uppdrag och kanske några av de taktiker som ska användas. Beskrivningen kan vara vald av testaren själv eller vara tilldelad av testledaren. Ibland är beskrivningarna nedskrivna. I vissa organisationer är testfallen så löst beskrivna att de i verkligheten har samma funktion som en övergripande beskrivning för utforskande testning.

Här är några charters för DecideRight, en beslutsanalysprodukt:

- Utforska och analysera delarna i DecideRight. Skapa en testtäckningskarta.
- Identifiera och testa alla anspråk/krav i manualen. Antingen genom att kryssa för direkt i manualen eller genom att lista varje testat krav i dina anteckningar.
- Definiera arbetsflöden genom applikationen och testa vart och ett av dem. Flödena ska representera realistisk användning och de ska tillsammans omfatta samtliga primära funktioner som finns.
- Vi behöver veta prestanda och pålitlighet hos DecideRight då komplexiteten i besluten ökar. Börja med ett normalt scenario och skala sen upp det i termer av antal valmöjligheter och faktorer tills dess att applikationen verkar hänga sig, krascha eller på ett snyggt sätt hindrar användaren från att ytterligare utöka komplexiteten.
- Testa alla fält som tillåter att du skriver in data (du vet tillvägagångssättet: funktion, stress och gränsvärden).
- Analysera filformatet hos ett scenario i DecideRight och kontrollera hur applikationen uppför sig då dess delar manipuleras. Testa felhantering och prestanda då korrupta filer används.
- Kontrollera användargränssnittet mot Windows-standarden.
- Finns det något sätt att göra en fil korrupt? Hur märker vi om den är korrupt? Undersök möjligheten att skriva en automatisk filkontrollant. Undersök om utvecklarna redan har byggt en sådan.
- Testa integration med externa applikationer, speciellt MS Word.
- Ta fram hur beslutsalgoritmen är uppbyggd genom att experimentera och reproducera den i Excel. Använd sedan bladet i Excel för att testa DecideRight med komplexa beslutsscenarier.
- Kör DecideRight under AppVerifiyer och rapportera eventuella fel.

Om du tycker att vissa av ovanstående charters är överlappande är jag inte förvånad. Syftet med dem är att kommunicera uppdraget med en testsession klart och kortfattat för en testare som redan är utbildad i de förutsättningar, vokabulär, tekniker och verktyg som används av organisationen. Kom ihåg att vi i utforskande testning maximerar användningen av egna färdigheter och inte försöker ha varje detalj i utförandet nedskriven.

I helt fri utforskande testning är det enda resultat från en session ett antal avvikelserapporter. I sessionsbaserad testledning resulterar varje session i en mängd anteckningar som granskas av testledaren. Det kan också resultera i uppdaterat testmaterial eller ny testdata. Om du tänker efter skapades troligen de flesta skriptade tester genom någon form av utforskande testning.

6.8 Utforskande testning inifrån: en aktivitet som kräver skicklighet

Precis som att en stjärna kan verka svag i ett spektrum synligt ljus men brinna starkt i infrarött blir den enkla iden om utforskande testning intressant och komplex när vi ser på den i ett spektrum av *skicklighet*. Ta schack till exempel. Reglerna för hur man spelar schack är betydligt mindre intressant än med vilken skicklighet man spelar. Ingen pratar om hur fantastiskt Emanuel Lasker följde reglerna för schack när han besegrade Steinitz i guldmatchen om världsmästerskapet 1894. Regler i schack förblir som de är, men valet av drag liksom spelarnas skicklighet växlar. Det som gör utforskande testning intressant, och enligt mitt synsätt mycket viktigt, är när en testare är skicklig på att lyssna, läsa, tänka och rapportera grundligt och effektivt utan i förhand nedskrivna instruktioner. Då kan det utforskande angreppssättet vara flera gånger så effektivt (i termer av att hitta vital information) som det skriptade. Och när det är ordentligt handlett och instruerat på lagom nivå kan även testare utan särskilda färdigheter producera användbara resultat som inte kunde ha förutsagts av ett skript. Här drar jag återigen paralleller till historien: Den spektakulärt lyckade expeditionen med Lewis och Clark är ett utmärkt exempel på vilken roll skickligheten spelade i utforskningen.

"Lewis var diplomaten och den kommersiella tänkaren, Clark förhandlaren. Lewis som åkte till Philadelphia för att utbilda sig i botanik, zoologi och navigering efter stjärnorna var den vetenskapliga specialisten, Clark var ingenjör och geograf samt mästare i forskningens hantverk. Båda var mycket intelligenta män, utmärkande intelligenta. Man finner ingen som kan kallas deras intellektuella jämlike i hela den sammanlagda tidigare historien av Nordamerikas utforskning." [33]

Den yttre utrustningen, indata och utdata i utforskande testning är värda att studera, men det är den inre strukturen som betyder mest - den del som sker i huvudet på testaren. Det är det utforskande testning står och faller med, när den utmärkta utforskaren skiljer sig från amatören. Låt oss titta på åtta grundläggande faktorer som skiljer experten från amatören och hur du kan bli skickligare på dessa delar.

6.8.1 Testdesign

En utforskande testare är först och främst en testdesigner. Vem som helst kan designa en test av en händelse. Den utforskande testaren kan bygga tester för en systematisk utforskning av produkten. Testdesign är förstås ett stort ämne men ett sätt att angripa det är att se det som en utfrågning. Att designa ett testfall är att ta fram en fråga till en produkt som ger ett svar med viktig information.

För att bli bättre på detta: Välj en funktion (något rimligt komplext, som tabellformateringen i ditt favoritprogram för texthantering) och ställ 30 frågor om denna som du kan besvara, helt eller delvis, genom att utföra någon form av testaktivitet. Med detta menar jag ett testfall, en samling tester eller något som skapar testfall. Identifiera aktiviteter för alla frågor. Om du inte kan hitta 30 frågor som alla skiljer sig åt, kör då några tester och försök igen. Lägg märke till hur antalet frågor ökar ju mer du ökar din erfarenhet av produkten. En annan aspekt av testdesign är att skapa modeller. Varje modell föreslår olika tester. Det finns många böcker om modellering, t ex UML (Unified Modeling Language). Välj någon typ av modell som ett flödesschema, en beslutstabell eller en tillståndsgraf och skapa en modell över den funktion du testar. När du kan skapa dessa typer av modeller på en servett eller en whiteboard på under två minuter, säkert och utan tvekan, kommer du att inse att du också är säkrare på att designa tester utan tvekan.

[33] De Voto, Bernard [1997]: The Journals of Lewis and Clark

6.8.2 Noggrann observation

Utmärkta utforskande testare är mer noggranna observatörer än nybörjare eller, för den delen, erfarna skriptade testare. Den som arbetar med skriptade tester behöver bara observera vad instruktionerna säger åt honom att observera. Den som håller på med utforskande testning måste titta efter *allt* som verkar ovanligt, mystiskt eller på annat sätt relevant för testningen. Utforskande testare måste också vara noga med att skilja observation från egna slutsatser. Även under press måste de undvika att förutfattade meningar gör dem blinda för viktiga tester eller produktens uppförande.

För att bli bättre på det här: Prova att studera någon annan testa det du redan själv har testat och lägg märke till vad den personen ser som du inte först såg och vice versa. En annan sak du kan göra är att filma skärmen när du testar eller använda en produkt som *Spector* som tar en bild av skärmen varje sekund. Granska periodiskt den sista kvarten av ditt testande och se om du upptäcker något nytt.

Eller prova att skriftligt beskriva ett fönster för någon annan och be personen återskapa fönstret enligt din beskrivning. Fortsätt till dess ni kan rita upp varandras fönster. Idealet är att upprepa detta med flera olika personer så att du inte bara blir bättre på att prata med en enskild person.

För att skilja mellan observation och egna slutsatser kan du observera en produkt, skriva ner dina iakttagelser och sen fråga dig själv för var och en av dessa om du verkligen såg detta eller om du bara drog dina egna slutsatser. Om jag exempelvis laddar in en fil i MS Word är jag frestad att säga att jag såg filen laddas men det gjorde jag inte egentligen. Sanningen är att jag såg vissa ord på skärmen som jag minns finns i just den filen och det tar jag som bevis på att filen laddades korrekt. Faktum är att filen kanske inte alls har laddats korrekt utan är korrupt på något sätt som jag ännu inte upptäckt.

Ett annat sätt att utforska observation och slutsatser är att studera trollerifram trädanden eller ännu bättre, själv lära sig att trolla. Alla magiska trick bygger på utnyttjandet av de misstag vi gör när vi drar slutsatser från observationer. Genom att luras av ett magiskt trick och sen lära mig hur det fungerar, får jag insikt i hur jag kan bli lurad av programvara.

6.8.3 Kritiskt tänkande

Utmärkta utforskande testare kan granska och förklara sin logik, leta efter fel i sitt eget tänkande. Detta är speciellt viktigt vid rapportering av status för en session av utforskande tester eller vid undersökandet av en avvikelse.

För att bli bättre på det här: Välj ett test du nyligen utfört. Undersök vilken fråga som låg till grund för just det testfallet. Vad var det du försökte komma underfund med? Försök sen komma på ett sätt du kan få ett testresultat på som pekar i en riktning (t ex att programmet är fel på något sätt) när verkligheten är det motsatta (t ex att programmet inte är fel utan vad du ser är sidoeffekterna av en parametersättning på ett annat ställe eller ett konfigurationsproblem). Är det möjligt att få testen att misslyckas trots att produkten fungerar perfekt? Är det möjligt att produkten är grundligt felaktig trots att testen verkade ge ett godkänt resultat? Jag kan komma på tre sätt detta kan ske på: otillräcklig täckning, otillräckligt orakel eller testarens eget fel.

Otillräcklig täckning betyder att ditt test inte når tillräckligt mycket av produkten för att uppfylla ditt mål. Kanske har du testat utskrifter men inte tillräckligt många utskriftssituationer för att rättfärdiga tron att utskriftsfunktionen fungerar. Orakel är mekanismer eller principer för att upptäcka ett problem om det inträffar. *Otillräckligt orakel* betyder att du använt en svag metod för att komma fram till att en avvikelse inträffat, och detta ledde till att du rapporterat något som inte är ett problem eller missat att rapportera något som verkligen är det. Kanske skrev du något till en fil och du kontrollerade att filen skapades men inte att innehållet i filen var korrekt. *Testarens eget fel* betyder att testdesignen var rätt men att du inte la märke till något som skedde eller använde fel data, misslyckades att sätta upp systemet rätt etc. Kanske såg du att resultatet av utskriften var korrekt men kom senare fram till att du egentligen tittade på resultatet från en annan test.

Eftersom testning i grunden är en oändlig process innehåller alla verkliga testsituationer kompromisser. Så du borde kunna hitta många sätt på vilket dina tester kunde luras. Tanken är att behålla medvetenheten om begränsningarna hos din testning. För en typiskt komplex produkt krävs det många tester för att besvara en given fråga med stor säkerhet.

6.8.4 Diversifierade idéer

Utmärkta utforskande testare producerar fler och bättre idéer än nybörjare. De använder heuristiker för att åstadkomma detta. Heuristiker är mentala verktyg som riktlinjer, generiska checklistor, minnesramsor eller tumregler. Satisfice Heuristic Test Strategy Model[34] är ett exempel på en grupp heuristiker för att snabbt generera diversifierade idéer. James Whittaker och Alan Jorgensens 17 attacker är en annan. [35]

För att bli bättre på detta: Öva på att använda *Heuristic Test Strategy Model*. Prova den på en funktion hos en produkt du vill testa. Gå igenom listan av

[34] www.satisfice.com/tools/satisfice-tsm-4p.pdf
[35] Whittaker, James [2003]: How to Break Software

idéer i modellen och fundera på ett sätt att testa funktionen för varje idé. Nybörjare har ofta problem med detta. Jag tror det beror på att listan fungerar främst för koppling till mönster för tidigare erfarenheter. Testare hittar något i modellen som retar deras minne att komma ihåg ett tidigare sätt att testa eller en typ av avvikelse och sen applicerar de detta minne på den sak de testar just nu. Idéerna i modellen är överlappande men var och en av dem innehåller också något som är unikt. En annan övning som jag rekommenderar är att skriva ner, ur minnet, 20 olika sätt att testa en produkt. Du måste kunna förklara hur varje idé är unik gentemot de andra. Eftersom jag har memorerat den heuristiska teststrategimodellen kan jag lista 33 olika sätt att testa när jag får frågan. Jag säger minnesregeln CITESTDSFDPOCRUSPICSTMPLFSDFSCURR till mig själv och sen utvecklar jag varje bokstav. Bokstav nummer två står till exempel för *information* vilket representerar idén ”hitta varje källa till information om denna funktion som jag kan och jämför källorna med varandra och med produkten och leta efter inkonsistenser”. O står för *operation* vilket representerar idén ”undersök miljön som produkten kommer att användas i och efterlikna den miljön så nära det är möjligt för mina tester”.

6.8.5 Rikliga resurser

Utmärkta utforskande testare skaffar sig en stor verktygslåda, informationskällor, testdata och vänner att använda som stöd. Under tiden de testar håller de ögonen öppna för tillfällen att använda dessa resurser som hjälp.

För att bli bättre på detta: Gå in på en webbplats med shareware som exempelvis Download.com och undersök vad som finns för varje område. Fundera på hur du kan använda de olika applikationerna som testverktyg. Besök webbplatser som har koppling till varje teknologi du testar och leta efter informationsblad eller utredningar. Skaffa mycket vänner som du kan kontakta för hjälp när du behöver en färdighet som de har.

6.8.6 Styrning av ditt eget arbete

Utmärkta utforskande testare styr användningen av sin egen tid. De måste kunna veta skillnaden mellan en återvändsgränd och ett lovande spår. De måste kunna relatera sitt arbete till sitt uppdrag och välja mellan de många olika saker som är möjliga att göra.

För att bli bättre på detta: Sätt upp en övergripande beskrivning för att testa något under en timmes tid. Det kan vara en enda mening som ”testa felhanteringen i rapportgeneratorn”. Sätt en larmsignal som låter varje hel kvart. Varje gång larmet går ska du säga högt varför du gör det du håller på med just då. Motivera det och berätta hur det du gör är kopplat till beskrivningen. Om det ligger utanför den övergripande beskrivningen, berätta varför du valde den vägen och om det visade sig vara ett bra beslut.

6.8.7 Snabb inlärning

Utmärkta utforskande testare har en snabbare inlärningskurva än de flesta andra. Intelligens hjälper så klart men även detta är en fråga om färdighet och övning. Det är också en fråga om att vara säker på sin sak – att, oavsett hur komplex och svår tekniken verkar till en början, tro på att du kommer att lära dig det du behöver veta för att testa det.

För att bli bättre på detta: Gå till en bokaffär och välj en slumpvis databok. Bläddra genom den under maximalt fem minuter. Stäng igen boken och besvara dessa frågor: vad gör den här teknologin, varför ska någon bry sig om den, hur fungerar den och vad är ett exempel på hur den fungerar i verkligheten? Om du inte kan besvara dessa frågor, slå upp boken igen och leta reda på svaren.

6.8.8 Statusrapportering

Knacka en utmärkt utforskande testare på axeln när som helst och fråga "Vad är din status?". Testaren kommer att kunna berätta vad som testades, vilka testtekniker och data som användes, vilka mekanismer som användes för att hitta eventuella problem om de inträffar, vilka risker testerna syftade till att undersöka och hur detta är relaterat till testuppdraget.

För att bli bättre på detta: Gör en 30 minuters testövning. Välj en funktion och testa den. Sluta efter exakt 30 minuter. Berätta sen högt utan hjälp av anteckningar vad du har testat, hur du skulle ha upptäckt ett problem om det inträffade, vilka problem du hittade och vilka hinder du påträffade. Med andra ord, gör en testrapport. En variant är att på tio minuter göra en skriftlig rapport.

6.9 Utforskande testning beror på situationen

Utforskande testning utvecklas under projektets gång. Hur det utvecklas beror på vad som inträffar. Det är ett pussel vi lägger och detaljerna för hur vi löser testpusslet, vilka framkommer under själva lösningsprocessen, påverkar vår taktik för hur vi löser det. Denna sanning ligger i hjärtat av alla utforskande undersökningar oavsett om det gäller test, utveckling eller till och med vetenskaplig forskning eller detektivarbete. Vilka detaljer påverkar då utforskande testning? Här är några av dem:

- ▶ målet med testprojektet
- ▶ målet med just denna testsession
- ▶ testarens roll
- ▶ testarens färdigheter, talanger och inställning
- ▶ tillgängliga verktyg och inrättningar
- ▶ tillgänglig tid
- ▶ tillgänglig testdata och material
- ▶ tillgänglig hjälp från andra personer
- ▶ krav på hur mycket vi behöver förklara vad vi gjort
- ▶ vad kunden bryr sig om
- ▶ aktuell teststrategi
- ▶ status för andra tester av samma produkt
- ▶ produkten själv
  - - användargränssnitt
  - - uppförande
  - - aktuellt läge för körbar kod
  - - kända avvikelser
  - - testbarhet
  - - syfte
- ▶ vad testaren vet om produkten
  - - vad som hände i föregående tester
  - - kända problem
  - - tidigare problem
  - - styrkor och svagheter
  - - riskområden och hur stora riskerna är
  - - nyligen genomförda ändringar
  - - egna observationer
  - - rykten
  - - vilka användarna är och hur de beter sig
  - - hur den ska fungera
  - - hur den är uppbyggd
  - - hur den liknar eller skiljer sig från andra produkter
- ▶ vad testaren skulle vilja veta om produkten

Istället för att fråga vilka tester som ska köras frågar en utforskande testare: ”Vilken är den bästa testen jag kan göra just nu?”. Varje fråga i listan ovan kan påverka vilka tester som behövs. Dessa faktorer ändras kontinuerligt under projektets gång och ibland även under en specifik testomgång. Styrkan hos de utforskande testerna kan optimeras under testernas fortskridande medan de nedskrivna testerna tenderar att bli mindre kraftfulla med tiden. Orsakerna till detta är flera men det främsta skälet är att när du har utfört ett skript en gång utan att hitta något fel så är chansen att du hittar fel nästa gång du utför samma skript betydligt mindre än om du skulle ha kört ett nytt testfall.

6.10 Att handleda utforskande testning

I många organisationer är det viktigt att skilja mellan en testledare som även har personalansvar och andra administrativa ansvarsområden och testledaren som har fokus endast på strategi och taktik. För denna diskussion använder jag beteckningen testledare för den som styr det dagliga arbetet och arbetar med strategin.

Fri utforskande testning kan ledas på två sätt, genom delegering eller deltagande. Vid delegering bestämmer testledaren de övergripande riktlinjerna. Sen går testarna iväg och utför arbetet med att designa och utföra tester enligt de övergripande instruktionerna och rapporterar sen tillbaka till testledaren. I praktiken får en specifik testare ofta ansvar för en grupp komponenter så att projektet gynnas av en obruten inlärningskurva. Testrapporterna som kommer tillbaka kan vara antingen skriftliga eller muntliga. Cem Kaner föreslår regelbundna möten med testarna, minst en gång i veckan. Han brukar öppna mötet med den mycket användbara frågan: ”Vad är den mest intressanta avvikelsen ni hittat nyligen? Visa mig.” I det sessionsbaserade angreppssättet görs skriftliga rapporter och testarna intervjuas minst en gång per dag.

Handledning genom delegering handlar i grunden om att behandla varje testare som man gör med en chef som styr användningen av sin egen tid. Precis som med en chef får en testare som ännu inte fått trovärdighet som produktiv och ansvarsfull inte några stora uppgifter att lösa utan får hålla sig till kortare utforskande sessioner och får se sitt arbete granskas mer noggrant. Att leda utforskande testare innebär att vara coach för delvis oberoende kreativa agerande personer.

Ledning genom deltagande betyder att testledaren själv aktivt utför tester tillsammans med resten av testarna. I praktiken fungerar detta bäst för de testledare som delegerat det administrativa arbetet och deltagandet i möten till andra personer. Deltagande ger ledaren möjlighet att styra teststrategin i realtid och löpande demonstrera vilket arbete han förväntar sig av resten av gruppen. Då testledaren ansvarar för gruppens prestationer är eget deltagande ett utmärkt sätt att uppfylla det kravet. Många orosmoment gällande förvirring eller ineffektiv testning tenderar att försvinna då testledaren själv är direkt inblandad i utförandet.

De flesta testledare använder någon form av guide för testtäckning som hjälp för att organisera arbetet. Guiden kan vara en testtäckningsöversikt eller matris, en risklista eller en vanlig gammal att göra-lista.

Utforskande testning i grupp kan vara extremt kraftfull. Erfarenheten hos många testledare som provat det säger att den sociala energin hos människor som arbetar tillsammans, letar efter avvikelser på samma utrustning och vid samma tillfälle, ofta leder till fler och bättre idéer än om samma människor hade arbetat enskilt. Ett sätt att organisera utforskande testning i grupp är att para ihop två testare som samsas om en dator. Ett annat sätt jag har gjort det på är att låta en testare utföra testerna samtidigt som flera andra ser på och kommenterar. Om den som utför testerna upptäcker ett problem eller har en fråga som behöver undersökas kan en av åskådarna gå ifrån och undersöka frågan på en annan dator. Detta låter den utförande testaren följa

sin huvudidé med färre störningar. Denna metod är speciellt användbar som ett kraftfullt verktyg för att få bort testningen ur hjulspåren, eller som hjälp att lära upp testare i teknologin runt produkten eller i metoder för testdesign.

6.11 Var passar utforskande testning in?

Det utforskande angreppssättet genomsyrar inte bara utveckling av programvara utan all intellektuell sysselsättning. Tänk på att utforskande och skriptad testning ligger i var sin ände på en kontinuerlig skala. De flesta tester har någon beståndsdel av utforskande typ. Titta noga och se själv.

Låt oss prata om *väldigt* utforskande testning. Fri utforskande testning passar i följande situationer:

- Du måste lämna snabb återkoppling på en helt ny produkt eller funktion
- Du måste lära dig en ny produkt snabbt
- Du har redan testat med hjälp av skript och behöver större spridning på testerna
- Du vill hitta det enskilt viktigaste felet på kortast möjliga tid
- Du vill kontrollera en annan testares arbete genom att göra en kort oberoende undersökning
- Du vill undersöka och isolera en specifik avvikelse
- Du vill undersöka status för en specifik risk för att bedöma behovet för skriptade tester på området

Förutom den fria variant som beskrivits passar utforskande testning överallt där testerna inte är helt dikterade i förhand. Detta inkluderar alla ovanstående situationer plus åtminstone dessa ytterligare:

- Improvisera vidare från nedskrivna tester
- Tolkning av vaga testbeskrivningar
- Produktanalys och testplanering
- Förbättring av existerande testskript
- För att skriva nya testfall
- Regressionstester baserade på gamla avvikelserapporter
- Testning som baseras på att kontrollera alla påståenden i användarmanualer

Att utforskande testning är så kraftfullt beror på att information kommer tillbaka från utförande av tester till att designa om dem. När vägarna för återkoppling är svaga, långsamma eller dyra tappar utforskande testning sin kraft. I detta fall måste vi gå tillbaka till att utföra noggrant nedskrivna testfall. Andra fall där det passar bra att använda sig av skriptade tester är kontroversiella delar av testerna, eller delar som kontrolleras extra noga av chefer och kunder. Men nöj dig inte med dåliga tester bara för att det behagar granskarna. Fundera istället på att använda en kombination av utforskande och skriptade tester och få det bästa av båda världar.

6.12 Utforskande testning i verklig användning

För att ge ett exempel på hur en utforskande testare tänker under utförande vill jag dela med mig av en av mina erfarenheter.

Jag hade en gång ett uppdrag att testa ett populärt fotoredigeringsprogram på fyra timmar. Mitt uppdrag var att bedöma hur det uppfyllde standarden för certifiering enligt kompatibilitet med Windows. Proceduren för detta är upplagd som en formaliserad utforskande testprocess. Mitt mål var att hitta alla brott mot kompatibilitetskraven som var tydligt nedskrivna för mig.

Med detta övergripande mål i sinnet började jag testa. Genom att använda en av de enklaste heuristikerna för utforskande testning började jag med att systematiskt använda alla funktioner under alla menyer i applikationen. Under tiden jag höll på skapade jag en skiss av de primära funktionerna i produkten. Detta skulle sedan vara underlag för min testrapport av vad jag testat och inte.

Jag noterade att *Spara som*-funktionen tog mig till en avancerad uppsättning kontroller som lät användaren ställa in parametrar för bildkvaliteten. Eftersom jag inte kände till de tekniska detaljerna av bildkvalitet kände jag mig oförberedd att testa dessa funktioner. Istället påbörjade jag en lista med frågor och noterade att jag skulle fråga om kunden var villig att låta mig utöka testtiden så att jag fick tid att studera dokumentationen för inställningar av bildkvalitet och skapa en strategi för att testa dessa. Efter att ha noterat detta fortsatte jag att gå igenom menyerna.

En grundläggande strategi för utforskande testning är att ha en generell angreppsplan och sen låta sig själv avvika från den i kortare omgångar. Det är som att åka på en sightseeingtur med buss. Även om bussen tar mig till olika ställen vid olika tidpunkter kan jag ändå stiga av ibland och gå omkring. Detsamma gäller för utforskande testning. Det finns ett värde att se det som visas på rundturen men det är också viktigt att ibland titta närmare på något eller att undersöka något som inte fanns på turlistan. Vad du än gör, somna inte på bussen – detta händer då du skaffar en plan för att besöka olika delar av produkten och sen besöker dem utan att fundera på hur bra produkten egentligen fungerar. En utforskande testare ifrågasätter alltid det som händer.

Min första instinkt att lämna turen av menyerna kom då jag hittade en dialogruta som lät mig kontrollera hur mycket minne som användes av applikationen. Detta gav mig en plötslig idé (plötsliga idéer välkomnas i utforskande testning). Eftersom stabilitet är ett av kraven för Windows-kompatibilitet kom jag på att det skulle vara intressant att försöka göra produkten instabil genom att först ställa in minnesanvändningen till ett minimum och sen be den utföra minneskrävande funktioner. Så jag ställde in skjutreglaget på fem procent av systemets minne och gick sen till bildinställningarna. Jag satte bildstorleken till en kvadrat med sidor på 2,5 meter vilket är en rejält stor tavla. Jag fyllde tavlan med lila prickar och gick sen till menyn för att välja olika grafiska effekter. Nu kommer den viktiga delen: jag valde *krusning* (eng *ripple*) från menyn och då visade produkten omedelbart ett felmeddelande att det inte fanns tillräckligt med minne för åtgärden. Det är ett intressant uppförande eftersom det sätter en standard. Jag har från och med nu uppfattningen att en funktion själv ska kunna hindra att den utförs om det inte finns tillräckligt med minne för åtgärden. Det här är ett utmärkt exempel på hur resultatet från en test påverkar de kommande testerna eftersom jag nu fortsatte med övriga effekter för att se om dessa uppförde sig på samma sätt. Vad upptäckte jag då? Ingen av de andra effekterna betedde sig på samma sätt som den första. Istället försökte de utföra instruktionen i fem minuter, utan att göra något som var synligt för mig annat än att provocera hårddisken så den skrek. Till slut dök felmeddelandet "*Error -32: Sorry this Error is Fatal.*" upp och applikationen kraschade. Det här är ett bra resultat men testerna vore inte kompletta (utforskande testare strävar efter att förutsäga alla frågor deras kunder kommer att ställa vid ett senare tillfälle) om jag inte ställde in minnesreglaget på max. Till min förvåning fick jag inte samma felmeddelande utan istället låste sig hela operativsystemet. Så ska inte Windows 2000 uppföra sig. Det här var ett betydligt allvarligare problem än endast en krasch.

Så här långt hade jag använt 30 minuter av en fyra timmars process och redan hittat ett problem som diskvalificerade produkten från certifiering. Det var de goda nyheterna. De dåliga var att jag tappade mina noteringar när systemet låste sig. Efter att ha bootat om beslutade jag att jag hade lärt mig tillräckligt från stresstestning och återvände till menyturen.

Jag hävdar att denna berättelse är ett exempel på disciplinerad, målinriktad testning. Jag kan rapportera vad jag testat och vad jag kommit fram till. Jag kan koppla det till det uppdrag jag fick. Det gick dessutom att upprepa, åtminstone lika bra som de flesta nedskrivna tester, eftersom jag hela tiden hade en sammanhängande idé om vad jag skulle testa och hur jag skul-

le testa det. Det faktum att dessa idéer dök upp under tiden jag testade, istället för att komma från ett i förväg nedskrivet dokument, är oväsentlig. Jag hoppas att du ser att detta är långt ifrån någon osystematisk testning.

6.13 Produktiviteten hos utforskande testning

Det finns inga rimliga siffror, inga giltiga studier, som jämför produktiviteten mellan utforskande och skriptad testning. Allt vi har är anekdoter. Här är några av mina.

Jag har undervisat klasser där utforskande testare, på ett system som automatiskt loggar deras arbete, har designat och utfört nästan 100 tester var. Sedan, när de fick uppgiften att skriva en repeterbar testprocedur på samma tid, lyckades varje person skapa endast ett enda test. Man kan argumentera att den repeterbara testen var bättre på vissa sätt än de utforskande testerna. Kanske var det så eller också inte. Personligen kände jag att de tester som skrevs ner var betydligt svagare än de utforskande.

Jag har lett grupper i deras tester av applikationer som redan testats med hjälp av skriptade testfall och hittat allvarliga problem i dem. I ett fall lyckades min grupp utan några förberedelser eller tillgängligt testmaterial krascha en nätverkskomponent så illa att den interna hårddisken fick formateras om. Det är sant att det tog mer än tio minuter att undersöka och isolera problemet men så hade också varit fallet för en skriptad testprocedur, förutsatt att denna alls hade hittat problemet.

Jag hjälpte ett stort företag att starta en utforskande testgrupp på en avdelningen omringad av testare som jobbade med skriptade tester. Det kom fram till att den utforskande gruppen, ohindrade av förvaltningen kopplad till testunderlagen, kunde följa mer ledtrådar och rykten om risker. De hittade fler och viktigare avvikelser. Tre år senare fanns gruppen fortfarande på plats.

En man kom fram till mig på en konferens och sa att han ledde en utforskande testgrupp för ett stort telekomföretag. Han berättade att hans grupp flyttar från projekt till projekt och anledningen att de tillåts fortsätta är att mätningarna visar att hans grupp hittar fyra gånger så många problem under en given tidsperiod jämfört med dem som arbetar med skriptade tester.

Dessa anekdoter bevisar ingenting, jag nämner dem för att få dig intresserad. Sanningen är att produktiviteten beror på många olika faktorer. Arbeta med det och samla dina egna erfarenheter.

Utforskande testning kan beskrivas som en kampkonst för tänkandet. Det handlar om hur du hanterar en produkt som hoppar ut från buskarna och utmanar dig till en duell i test. Du får inte svart bälte genom att läsa böcker, du måste arbeta aktivt. Lycka till!

6.14 Tack till

Denna artikel har dragit nytta av ifrågasättande och idéer från följande kollegor: Cem Kaner, Lee Copeland, Brian Marick och Dorothy Graham. Delar av denna artikel publicerades först i "Inside the Mind of the Exploratory Tester", STQE Magazine, volume 5, issue 6, November/December 2003.

7 Testdesigntekniker - en översikt

7.1 Statiska och dynamiska tekniker

Vi brukar skilja på statiska och dynamiska testdesigntekniker. En statisk teknik innebär att du granskar dokument, design eller kod medan en dynamisk innebär att du studerar programmet under körning. Nedan visas en bild av den vanligaste uppdelningen. [36]

Figur 7.1: Översikt av testdesigntekniker. Dynamiska tekniker kan fokusera på funktioner eller på kvalitetsfaktorer. Statiska tekniker som innebär granskning av kod eller dokument brukar även de räknas till test trots att ingen kod exekveras.

7.2 Statiska tekniker

En statisk testdesignteknik innebär att ingen programkod exekveras. Tekni-

[36] ISTQB [2005]: Kursplan, kap 4

ken innebär istället att olika typer av dokument i form av text, modeller eller kod analyseras, vanligen manuellt. I en del fall, som t ex vid kompilering av programkod, använder man sig av verktyg. De avvikelser man hittar har ofta med krav och design att göra, antingen saknas det delar eller så stämmer dokumenten inte överens inbördes eller med den standard som finns. Fördelarna med granskning är att vi kommer in tidigt och att vi kompletterar den dynamiska testningen genom att vi hittar andra sorters fel. Hur formell granskningen är brukar styras av hur kritiskt materialet man granskar är. En kravspecifikation som ligger till grund för allt annat arbete granskas noggrannare än en programspecifikation på en liten del av systemet. Det gäller att välja rätt granskningstyp för att få ut mesta möjliga nytta. Granskningar anses av de flesta organisationer idag som värdefulla och är vanligt förekommande i många olika branscher. Jag har jobbat med olika former av granskningar inom telekom, bank, försäkring och statliga verk. I de allra flesta fall är granskningarna informella och inte på något ställe utanför den akademiska världen har jag varit med om inspektioner på det sätt som beskrivs nedan.

7.2.1 Inspektion

Den mest formella granskningstekniken kallas *inspektion* och är hårt styrd. Metoden utvecklades av Michael E. Fagan på IBM.[37] Deltagarna förbereder sig noga genom att utifrån rollbeskrivningar och checklistor granska utvalda områden. Under granskningsmötet loggas alla synpunkter och hantering av dem kan diskuteras. Däremot löses problemen i efterhand. Då arbetssättet är formellt är det endast ett begränsat material som granskas vid varje tillfälle, oftast inte mer än ett tiotal sidor beroende på underlaget. För att denna typ av granskning ska vara effektiv krävs att deltagarna utbildas, att checklistor tas fram och att själva granskningsmötet leds av en erfaren moderator. För den som önskar lära sig mer finns en detaljerad beskrivning i boken *Software Inspection*[38].

[37] Fagan, Michael [1976]: Design and Code Inspections to Reduce Errors in Program Development. IBS Systems Journal, 15(3), s 182-211. [38] Gilb, Tom, Dorothy Graham [1993]: Software Inspection

7.2.2 Genomgång

En av de mer informella och enklare teknikerna kallas genomgång (eng *walkthrough*). Den innebär att författaren presenterar sitt material för en utvald grupp deltagare för att på så sätt sätta in dem snabbare i underlaget. Frågor och förklaringar välkomnas under mötet. Syftet är mer att skaffa en gemensam bild än att identifiera felaktigheter, men tekniken räknas ändå till statisk testning.

7.2.3 Teknisk granskning

En teknisk granskning fokuserar som namnet säger på de tekniska delarna i projektet. Här ska de tekniska experterna och arkitekterna delta tillsammans med övriga utvecklare. Syftet är både att utvärdera val av lösning och överensstämmelse mot standarder och övriga dokument. Resultatet dokumenteras ofta i en logg.

7.2.4 Informell granskning

Den informella granskningen innebär att två eller flera personer tittar igenom ett dokument eller kod som någon av dem skrivit. Syftet är fortfarande att hitta fel men vanligen används inga checklistor och resultatet behöver inte dokumenteras. [39]

7.2.5 Alternativ till granskning

Den metod jag föredrar passar inte in under de tekniker som räknats upp ovan. Det handlar mer om att analysera det underlag som finns och utifrån detta skapa modeller av hur jag tror att det ska fungera. Genom att ta fram modellerna och diskutera dem med inblandade personer hittar jag ofta avvikelser i form av information som saknas eller inte är konsistent, och ibland hittar jag även rena felaktigheter. Vi går igenom modellering i detalj senare i boken.

[39] Weinberg, Gerald [1971]: The Psychology of Computer Programming

7.3 Dynamisk testdesignteknik

Dynamisk test innebär att vi testar koden genom att exekvera den. Vi brukar dela in de dynamiska teknikerna i *beteendebaserade*, ofta benämnda black-box, och *strukturella*, vanligen kallade white- eller glass-box. Oavsett om vi jobbar med beteendebaserade eller strukturella metoder finns det en mängd olika tekniker som kan användas för att få fram bra testfall.

7.3.1 Data

För att hantera testdata delar vi in värdemängden för en parameter i *ekvivalensgrupper*. För framför allt numeriska intervall finns också *gränsvärdesanalys* för de grupper vi tagit fram. Hantering av testdata på det här sättet återkommer som en viktig del i de flesta andra tekniker och kallas därför i den här boken för *de grundläggande teknikerna*.

7.3.2 Flöden

Det finns flöden på många olika nivåer från övergripande verksamhetsprocesser till programkod som har både kontrollflöde och dataflöde. Principerna för flödestester liknar varandra oavsett nivå. Först ritar vi upp flödesschemat i en grafisk modell, sedan täcker vi in de varianter på flöden som finns.

Hur bra testtäckningen behöver vara beror på hur komplicerat flödet är. Det kanske något otillfredsställande svaret är att det beror på. Den lägsta nivån för täckning som jag använder mig av är att alla grenar ska vara täckta. Det fungerar för det mesta bra vad gäller mer övergripande flöden som exempelvis verksamhetsprocesser. Mer detaljerade flöden som exempelvis användningsfallsflöde i form av aktivitetsdiagram kräver ofta att du har fler kombinationer av olika flöden. Det finns inte någon generell regel för vilken nivå du bör eller måste täcka för att ha en bra samling tester.

7.3.3 Logik: regelverk, formler

För att hantera regelverk och matematiska formler är *beslutstabeller* kraftfulla hjälpmedel för att skapa testfall. För att kontrollera om en beslutstabell är korrekt – komplett och inte överlappande - kan du för vissa typer av regelverk rita ett *beslutsträd*. Grafen är dessutom ett utmärkt hjälpmedel även då du kör dina tester och utvärderar eventuella avvikelser.

7.3.4 Kombinatorik

Ett ofta förekommande problem är att du har ett stort antal variabler som kan kombineras med varandra i en nästan oändlig variation. Att testa alla kombinationer för ett antal variabler som beror på varandra är ofta inte möjligt. Vi får snabbt vad vi kallar en kombinatorisk explosion vilket är vanligt inom test.

För att hantera denna typ av problem finns ett antal användbara sätt att förenkla utan att vi missar för mycket. *Grundläggande jämförelser*[40] (eng *modified condition decision coverage*) innebär att alla ingående variabler i ett villkor ska få bestämma resultatet av villkoret minst en gång och alla resultat av villkoret ska uppnås. Ett annat sätt att kombinera variabler är att testa variabler parvis med varandra så att *alla par* täcks.

I detta första steg väntar vi med att analysera data i detalj även om vissa data har en direkt koppling till vissa grundtestfall.

7.3.5 Riskbaserad testning

Det finns en mängd olika tekniker som alla bygger på tidigare erfarenheter om vad som brukar gå fel. Till den här gruppen hör de varianter som kallas *risklistor, felgissning, taxonomier, felklassificeringar, attackmönster.*

Felgissning beskrivs i viss litteratur som en egen teknik.[41] Syftet är att utifrån erfarenhet, teknikkunskap, gamla felrapporter etc peka ut områden vi misstänker har fel. Ett alternativt synsätt är att se *felgissning* som en naturlig del av alla tekniker.[42] I denna bok räknas den som en riskbaserad teknik.

[40] Pol, Martin, Ruud Teunissen, Erik van Veenendaal [2002]: Software Testing – A Guide to the TMAP Approach, s 219-223. [41] Myers, Glenford [1979]: The Art of Software Testing, s 37.
[42] Beizer, Boris [1995]: Black Box Testing, s XIV

7.3.6 Avancerad testning

Då de enskilda funktionerna i programvaran har testats gäller det att kontrollera att även det stora sammanhanget fungerar. Vi har redan tagit upp *test av verksamhetsprocesser* under flöden ovan. Andra användbara tekniker är test av *tidscykler*, *datacykler*, *såpoperatestning*[43], *livscykler* och *syntaxtestning*. Vi kan också arbeta med användarprofiler för att ta reda på hur de verkliga användarna kommer att använda vår applikation och utifrån det testa lämpliga kombinationer.

7.4 Svart, vitt och grått

7.4.1 Beteendebaserad testteknik – Black Box

Vi kallar ofta beteendebaserad testning för indata-, utdatadriven testning. Den är inriktad på att verifiera att systemet uppför sig rätt utåt utan att vi bryr oss om vad som händer inuti – därav det ofta använda uttrycket black-box-tester. Som utgångspunkt har du ofta en specifikation i form av användningsfall, arbetsprocess eller annat kravdokument. Vi ställer en fråga till systemet och får ett svar, stämmer svaret med vad vi förväntar oss är testfallet utfört med godkänt resultat. Att vi inte bryr oss om vad som händer inuti systemet betyder inte nödvändigtvis att vi inte vill veta hur det fungerar – en sån definition tycker jag är felaktig. Ofta kan vi utifrån systemdesignen få värdefull information som hjälper oss att göra bättre testfall. Ett exempel är om en utvecklare gjort en uppdatering i ett program vill vi veta vilka delar av systemet som kan påverkas av ändringen och vilka regressionstester vi därför bör köra. Mer om detta i avsnittet Grey Box.

7.4.2 Strukturell testteknik – White Box

Strukturella tester kallas även white eller glass box-test. Syftet med testerna är att verifiera den interna logiken i testobjektet. Tekniken används vid program- och programintegrationstest och kräver kunskap om hur systemet är uppbyggt i detalj. De strukturella testerna omfattar *kontrollflöde* (programflöde) och *dataflöde*. Kontrollflödestester kontrollerar att den interna programlogiken fungerar. Dataflödestester verifierar att hanteringen för hur en variabel deklareras, används och förstörs är korrekt. [44]

De underlag som används är program- och designspecifikationer där detaljerna om programmets interna struktur är tydligt beskrivna. Det är speciellt viktigt att noga testa alla gränsvärden och felsituationer. Du kan utföra strukturella tester utan verktyg men om du ska mäta täckningsgraden är detta mycket svårt utan automatiserade hjälpmedel. Strukturella tester påverkas av programspråk och det stöd testverktygen ger.

7.4.3 Kombinerat - Grey Box

Ofta använder vi oss av en kombination av beteendebaserade och strukturella tekniker för att få bättre testfall. De kallas ofta grey box-test. De innebär att vi tittar inuti lådan på de strukturer som finns för att göra bättre beteendebaserade tester. Kunskap om programmering och databaser är därför en styrka för att hitta fler fel. [45]

7.5 Icke-funktionella tester

Förutom rena funktionskrav finns även egenskapskrav. Kategorin egenskapskrav rymmer så många aspekter att det är lämpligt att göra en ytterligare kategorisering för att göra kravbilden mer tillgänglig. En bra grund för en sådan indelning ges i standarden ISO 9126 Software Product Evaluation som beskriver hur man kan strukturera framför allt egenskapskraven. ISO 9126 [1991] anger att en produkts kvalitet ska bedömas efter följande sex huvudrubriker. Ett appendix i standarden innehåller en modell för hur dessa kan tolkas genom att komplettera med ett antal underavdelningar per rubrik:

[43] Såpoperatestning handlar kort sagt om att tänka ut alla udda och extrema fall som kan förekomma

[44] Copeland, Lee [2003]: A Practitioner´s Guide to Software Test Design, s 139-142

[45] Copeland, Lee [2003]: A Practitioner´s Guide to Software Test Design, s 8, Whittaker, James [2003]: How to Break Software, s 57-58

1. **Funktionalitet:** Finns de önskade funktionerna med? lämplighet, korrekthet, samverkan, överensstämmelse med standard, säkerhet
2. **Tillförlitlighet:** Är systemet robust och fungerar i olika situationer? mognad, feltolerans, återgång (efter fel), tillgänglighet
3. **Användbarhet:** Är systemet intuitivt, förståeligt och enkelt att använda? begriplighet, inlärningskrav, handhavande
4. **Effektivitet:** Använder systemet resurser på ett bra sätt? tidsaspekter (t ex prestanda), resurskrav (t ex skalbarhet)
5. **Underhållbarhet:** Kan drift, utvecklare och användare uppgradera systemet då det behövs? analyserbarhet, modifierbarhet, konfigurerbarhet, testbarhet
6. **Flyttbarhet:** Kan systemet fungera på olika plattformar, med olika databaser etc? anpassningsbarhet, installationskrav, överensstämmelse med standard, ersättningsbart

Det är viktigt att vi inte glömmer bort ovanstående delar. Frågan är vems ansvar det är att se till att de fungerar? Funktionaliteten är den del som vi testare arbetar mest med, tillförlitlighet och effektivitet är mer tekniska delar som oftast kräver någon form av verktyg och specialistkunskaper. Användbarhet är i fokus redan i designen av gränssnittet mot användaren, detta bör göras av experter på användbarhet [46] men glöms alltför ofta bort. Det är inte en bra idé att överlåta denna typ av tester till testare utan specialkunskaper om dessa frågor!

7.6 Hur många designtekniker finns det?

Egentligen är frågan hur många designtekniker som finns i sig ointressant. Det som är intressant är vilka tekniker du behöver kunna för att göra ett bra arbete och hur du för varje situation anpassar teknikerna så att de passar just där. Bara för att du varierar en teknik efter situationen betyder ju inte det att det är en helt annan teknik. Det är inte att rekommendera att ha en favoritteknik som du alltid använder, utan ha en uppsättning grundtekniker som du behärskar väl och kan applicera på olika situationer.

Jag har under flera år tränat kampkonsten Aikido som bygger på samurajernas självförsvar. Min mästare pratar om tolv grundtekniker som kan varieras utifrån vilken situation du befinner dig i. Det är mycket viktigt att behärska grundteknikerna väl, nästa steg är att kunna anpassa sitt försvar efter motståndare och attacksituation. Det är just känslan och timingen som kännetecknar en mästare. Nedan finns ett urval av olika organisationers syn på vilka testtekniker som är viktiga. Som du kommer att märka skiljer sig sättet att dela in dem på och synen på vad som är viktigt beroende på vem du pratar med.

7.7 Bokens urval

Teknikerna i boken är det urval jag själv anser varje testare bör ha kunskap om och kunna applicera. Samtliga tekniker är vedertagna och väl kända i testsammanhang både i Sverige och internationellt. Vill du läsa mer om varje teknik finns det gott om referenser i litteraturlistan. Urvalet gör på inget sätt anspråk på att vara heltäckande, eller vara det enda du ska kunna, men det är en bra grund att börja med.

- Data: *ekvivalensgrupper, gränsvärdesanalys, domäntestning*
- Flöden: *verksamhetsprocesser, användningsfall till testfall, tillståndsgrafer*
- Logik: *beslutsträd, beslutstabeller*
- Kombinatorik: *alla par, grundläggande jämförelser*
- Riskbaserad testning: *risk, felgissning, taxonomier, heuristiker, attackmönster*
- Avancerad testning: *scenariobaserat, såpopera, tidscykler, datacykler*
- För utvecklaren: *kontrollflöde, dataflöde*

[46] Cooper, Alan [2004]: The Inmates are Running the Asylum

7.8 Internationella standarden ISTQB

Det finns en internationell sammanslutning vid namn International Software Testing Qualifications Board, ISTQB[47]. Den tar i sin kursplan för certifiering i grundkursen upp de testdesigntekniker den tycker är viktigast för funktionella tester. Från början bygger detta urval på den brittiska standarden BS 7925:2 för programtest. I den senaste officiella versionen listas teknikerna: [48]

1. Ekvivalensgrupper
2. Gränsvärdesanalys
3. Tillståndsdiagram
4. Beslutstabeller
5. Test av användningsfall

Den tar dessutom upp angreppssätten:

6. Felgissning
7. Utforskande testing

För strukturell testning tar den upp kodtäckning i form av ett antal olika varianter där du täcker in alla:

1. Kodsatser och kodtäckning (statement)
2. Kodgrenar (branch)
3. Looptäckning
4. En kombination av kodgrenar och villkor (multiple combination decision coverage)

7.9 Den akademiska världen

Enligt den välkända auktoriteten inom test, Cem Kaner, som är lärare på Florida Institute of Technology och författare till flera böcker, är de viktigaste testteknikerna de 10 grundläggande tekniker han lär ut i sin black box-testkurs: [49] [50]

[47] www.istqb.org. [48] ISTQB kursplan [2005]: version 0.2. [49] Kaner, Cem [2005]: groups.yahoo.com/group/software-testing/message/2880. [50] www.testingeducation.org/BBST/BBST--IntroductiontoTestDesign.html

Han lär ut :

1. Funktionstest
2. Specifikationsbaserad test
3. Domäntest
4. Riskbaserad test
5. Scenariotest
6. Regressionstest
7. Stresstest
8. Användartester
9. Tillståndsgrafer
10. Volymtester

Cem Kaner och James Bach har valt dessa tio eftersom de exemplifierar de viktigaste områdena inom blackbox-testning:

”Det finns en separat kurs för programtest. Men syftet med att lära ut dessa tekniker är inte att göra folk bra på tio tekniker. Jag vill att eleverna ska förstå att det finns en mängd olika varianter på tillvägagångssätt, att de har fundamentalt olika styrkor och att den skicklige testaren väljer teknik eller hittar på en teknik som passar de behov som finns just nu.

Exempel på variation: några av våra tekniker fokuserar på indata och utdata, andra på kontrollflöde, vissa på interaktion med andra program eller enheter (detta kan modelleras som indata/utdata, men på grund av att interaktionerna ofta innehåller sekvenser av meddelanden kan det ofta bättre modelleras som en tillståndsgraf med tillhörande dataanalys). Vissa tekniker fokuserar på överensstämmelse med legala krav, specifikationer, annonser etc. Andra fokuserar på verklig användning av produkten etc. Alla dessa för testaren in i en ny undersökning på jakt efter olika informationskällor som hjälp i design och utvärdering av testfall.”

7.10 Företag som är långt framme inom test

Peter Zimmerer på Siemens i Tyskland presenterade på EuroSTAR i december 2005 det han kallar Test Design Poster[51]. Att jag har valt att publicera hela listan här beror på att det är en av de mest kompletta förteckningar jag sett. Han berättar så här:

”Test Design Poster innehåller en kategoriserad överblick av testdesignmetoder, paradigmer, tekniker, stilar och idéer för hur du skapar testfall. Vi delar ut den till alla som arbetar med utveckling och test för att ge dem inspiration. Sen kommer vi ut i organisationen och visar hur de ska använda de olika metoderna.

För en tillräckligt bra testning krävs att flera olika metoder används i kombination. Val av metoder beror på många faktorer, bland annat:

1. Krav på systemet och önskad kvalitet
2. Krav på testerna, styrka och djup
3. Teststrategi: vilka tester görs i vilken del av utvecklingskedjan
4. Existerande testunderlag
5. Systemet som ska testas
6. Teknik för mjuk- och hårdvara
7. Passande verktygsstöd

Arbetsinsats/svårighet för metoderna eller egentligen intensiteten delas in i fem nivåer från ett som är mycket liten/enkel till fem som är mycket hög/svår.”

[51] Zimmerer, Peter [2005]: EuroSTAR 2005 presentation

Kategori	Metod, paradigm, teknik, stil och idéer för testfall	Svårighetsgrad/ intensitet
Black Box Gränssnitt, data, modeller	Standarder, normer, formella specifikationer, krav	3
	Kännetecken, funktioner, gränssnitt	1
	Kravbaserat med spårbarhetsmatris	3
	Användningsfallsbaserat (aktivitestdiagram, sekvensdiagram)	3
	CRUD, Create, Read, Update, Delete (databas-operationer)	3
	Scenariotester, såpoperatester	4
	Användarprofiler: frekvens och prioritet/kritiskt (tillförlitlighet)	4
	Statistisk testning (Markovkedjor)	4
	Random (aptestning)	4
	Design med kontrakt (inbyggd självtest)	3
	Ekvivalensgrupper	2
	Klassificeringsträd	3
	Domäntester, kategoriuppdelning	4
	Gränsvärdesanalys	2
	Speciella värden	1
	Testkatalog för indatavärden, indatafält	5
	Tillståndsgrafer	3
	Orsak-verkangrafer	5
	Beslutstabeller	5
	Syntaxtester (grammatiker)	4
	Kombinatorisk testing (parvisa tester)	3
	Evolutionstester	5
Grey-box	Beroende/relationer mellan klasser, objekt, metoder, funktioner	2
	Beroende/relationer mellan komponenter, services, applikationer, system	3
	Kommunikationsbeteende (beroendeanalys)	3
	Spårning (passiv testning)	3
	Protokollbaserat (sekvensdiagram)	4

Tabell 7.2a Testdesign poster del 1

White-box	**Kontrollföden**	**Täckning**	**Påståenden**	**2**
Intern struktur, vägar		(kodbaserad, modellbaserad)	Grenar	3
			Villkor	4
			Gränssnitt	4
		Statisk mätning	Cyklomatisk komplexitet (McCabe)	4
			Mätning (ex Halstead)	4
	Dataflöden		Läsa/skriva	3
			Definiera/använda	5

Tabell 7.2b Testdesign poster del 2

Positiva, giltiga fall	**Normalt, förväntat uppförande**	**1**
Negativa, ogiltiga fall	Otillåtet uppförande	3
	Felhantering	3
	Undantagshantering	5
Felbaserat	Riskbaserat	2
	Systematisk felanalys (FMEA: Failure Mode and Effect Analysis)	4
	Felkataloger, bug taxonomies (Biezer, Kaner)	4
	Attackmönster (Whittaker)	3
	Felmodeller som beror på teknologin och naturen hos systemet	2
	Felmönster: standardmönster eller utifrån rotorsaksanalys	3
	Felrapporter (tidigare)	2
	Felgissning	2
	Testmönster (Binder), Frågemönster (Q-patterns: Vipul Kochar)	3
	Ad Hoc, intuitivt, baserade på erfarenhet	1
	Exploratory testing, heuristiker	2
	Mutationstester	5

Regressionstest	Delar som har förändrats	1
	Delar som påverkas av ändringarna	2
	Riskabla, högprioriterade, kritiska delar	3
	Delar som används ofta	3
	Allt	5

Tabell 7.2c Testdesign poster del 3

7.11 Övriga källor

Det finns många ytterligare källor där du kan hitta listor på tekniker. Några av de mest kända är TMAP: Test Management Approach [52], som är väldigt stor i bland annat Nederländerna och Belgien. Boris Beizer har skrivit ett par klassiska verk om testtekniker: Software Testing Techniques[53] och Black Box Testing [54]. Slutligen finns Glenford Myers bok The Art of Software Testing[55] från 1979 som var det första som skrevs inom området. Många av hans tankar håller bra än idag och det finns förmodligen inte en enda nyare testbok på marknaden som inte innehåller en referens till denna bok. Boken har nyligen kommit ut i en uppdaterad version, mer än 25 år efter att originalet skrevs.

[52] Pol, Martin, Ruud Teunissen, Erik van Veenendaal [2002]: Software Testing – A Guide to the TMAP Approach
[53] Beizer Boris [1990]: Software Testing Techniques. [54] Beizer, Boris [1995]: Black Box Testing.
[55] Myers, Glenford [1979]: The Art of Software Testing