Systematisk Testing av Software

En kort oversikt over et vanskelig fagfelt.

Av Hans Schaefer

hans.schaefer@istqb-norge.no
www.istqb-norge.no

Thomas Borchsenius, Monika Stöcklein-Olsen og Skule Johansen fra Norwegian Testing Board har gjennomgått dette dokument og kom med mange nyttige kommentarer.

Introduksjon

De fleste vet hva programmering er. Men hva er testing?

Generelt betyr testing at en sjekker produkter for å finne ut om de virker ”bra nok”. Uten test vet vi ingenting om produktet. Etter en test vet vi i hvert fall noe. Flere detaljer om dette kommer senere.

I forbindelse med utvikling av nye softwareprodukter koster testing fra en tredjedel til halvparten av arbeidet. Ser vi på vedlikehold, står testing for en enda høyere andel. Ved sikkerhetskritiske produkter ligger andelen også høyere, gjerne over halvparten av alt arbeid som gjøres.

Intuitivt vil de fleste ha en forståelse for at software burde være systematisk testet. Hvis en ligger på sykehus og er tilknyttet softwarestyrte medisinske apparater kan feil ha fatale konsekvenser. Ellers i livet er feil mer eller mindre irriterende, men mer og mer av våre omgivelser blir styrt av software, og dermed vil vi være interessert i at slike systemer er nøyaktig sjekket eller testet.

Dessverre finnes det mange prosjekter der testingen foregår alt for usystematisk og ufullstendig. Denne veiledningen skal gi en kort innføring i testingens grunnleggende prinsipper og metoder, slik at leseren får et inntrykk av hva testing kan bidra med og hva som eventuelt lønner seg å studere videre.

Her er noen spørsmål mange skal ha svar på vedrørende testing:

- Hvordan tester jeg? Hva slags teknikker og metoder finnes det og hvor er de brukbare?
- Hva betyr å teste ”nok”? Er der brukbare kriterier for å avslutte testingen?
- Hva slags oppgaver består testingen av, og når og i hvilken rekkefølge bør de utføres?
- Hvordan kan jeg evaluere og forbedre måten jeg tester på?
- Hvilke verktøy finnes for å hjelpe å utføre testoppgaver eller for å utføre de bedre, fortere og billigere enn før?
- Hvordan bli anerkjent som tester?

I denne innføringen får du noen svar på disse spørsmålene. Dette kan dog bare være en meget grov oversikt, men den gir deg forhåpentligvis en inngang til mer kunnskap om testfaget. Målet er at du som utvikler eller tester kan bidra til økt kvalitet i softwareprodukter, og at du kan gi sikrere utsagn om kvaliteten.

Denne innføringen baserer seg på en innføring på tysk skrevet av Prof. Andreas Spillner fra Faghøyskole Bremen. Tittel er ”Systematisches Testen von Software – Ein Einstieg” og er utgitt av dpunkt Verlag i 2008. Den er tilgjengelig for gratis nedlasting på http://www.dpunkt.de/download_softwaretesten.php.

Hva er testing?

En tester av to grunner:

- **Å finne feil**: Programvare blir utviklet av mennesker. Dermed vil det (nesten) alltid være feil i den. Testingen skal finne eventuelle feil før produktet blir brukt.
- **Å demonstrere kvalitet**: Testing skal vise at kvalitetskrav er oppfylt. Dvs. at programvaren ”fungerer” og at egenskaper som svartider, kapasitet, brukbarhet, sikkerhet etc. er oppfylt i tilstrekkelig grad.

Begge deler er i prinsipp like viktige. Før vi tester vet vi lite om programmet. Det har ukjent kvalitet.

I og med at mennesker er motivert av målsetningen ved arbeidet er det en fordel å definere testingens oppgave negativt, dvs. testingen skal finne eventuelle feil. Testerens oppgave er å vise at programvaren IKKE fungerer som den skal. Med en slik definisjon vil vi finne mer feil. Dette svarer til den vanlige vitenskapelige arbeidsmetoden: Noen framsetter en ny teori. Mange andre vil da prøve å motbevise teorien og slik finne eventuelle svakheter. Dersom de ikke klarer å finne svakheter, vil teorien etter hvert bli ansett som fornuftig eller ”sannsynliggjort”. Testingen kan dog aldri **bevise** at noe er rett. En demonstrerer kvaliteten indirekte ved å prøve å finne feil.

Det er klart at de mer alvorlige feil bør rettes allerede under utviklingen av programvare, slik at testingen av det endelige produktet ikke vil oppdage mange feil. Å finne få feil demonstrerer da kvalitet.

Det neste spørsmålet er hva **kvalitet** betyr. Her er det igjen to aspekter:

- At produktet oppfyller spesifiserte krav
- At produktet oppfyller kundens og brukerens (underforståtte) behov

Å teste mot spesifiserte krav er utgangspunktet, og det er relativt enkelt: Listen over krav er gjerne gitt og en prøver om hvert enkelt krav er oppfylt. Å teste mot underforståtte krav betyr først og fremst at en som tester må tenke ”noe lenger fram enn nesetuppen”, dvs. prøve å forstå hva som virkelig er viktig når en kunde eller bruker blir utsatt for et produkt. Dette selv om slike saker ikke står i den offisielle spesifikasjonen. Eksempel: Et salgssystem må være minst like enkelt og robust å bruke som de enkleste andre systemene som kundene er vant med. Et system bør ikke stoppe selv når det blir betjent feilaktig. Og feilmeldingene bør være riktige og forståelige!

Det er en stor fordel hvis feil blir funnet tidlig. I så fall kan de rettes tidlig under utviklingen, noe som er billigere enn å finne de under avsluttende test eller i drift. Dette betyr at testingen skal begynne så tidlig som mulig under et utviklingsprosjekt.

Et annet aspekt av testingen er hvor langt eller bredt arbeidet går. Generelt betyr testing å utføre programmet på den/de enhetene det er laget for, eller på en plattform som simulerer den/de enhetene de er laget for. Det kan dog være lurt å tenke litt lengre her: Å sjekke dokumenter som kontraktsutkast, planer, spesifikasjoner, opplæringsmateriell, brukermanualer etc. kan være del av testingen. Dette blir dog ikke fulgt videre i denne introduksjonen.

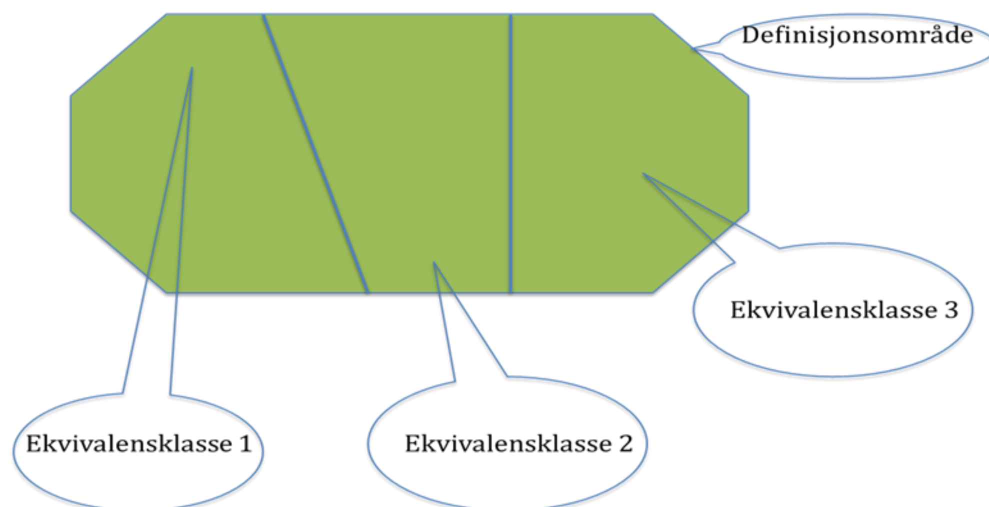
I praksis blir test ofte utført av utviklere. Dette skjer mer eller mindre usystematisk. En slik test gir ingen sikre resultater. Noen mener alternativet er å teste ”alt”, dvs. alle kombinasjoner av input og funksjoner. Dette er ikke praktisk mulig. Det en gjør er å bruke systematiske teknikker som så skal gi utsagn om systemets kvalitet med en viss, men ikke 100%, sikkerhet. Slike typiske teknikker blir vist i de neste avsnittene. Det finnes mange flere, men disse er beskrevet i videreførende faglitteratur.

Typiske testteknikker

Ekvivalensklasseanalyse

Teknikken har et skremmende navn, men tilsvarer det mange utviklere gjør intuitivt. Her er hvordan teknikken fungerer:

Man analyserer et dataelement, dvs. et felt eller en verdi for inndata eller utdata. En ekvivalensklasse (eller kort: klasse) er da en samling verdier der testeren tror at alle verdiene i denne klassen behandles prinsipielt likt av programmet. Samlingen av alle ekvivalensklasser for denne verdien skal omfatte alle mulige verdier som kan oppstå, altså liksom ”hele verden”.
De viktigste fordelene med denne metoden er at en ikke tester mange ting dobbelt og at en ikke glemmer ulovlige og feile verdier for data.



I prinsipp er det likegyldig hvilke konkrete verdier en benytter i testingen: Hvis én verdi finner en feil, vil alle andre verdier i samme klassen gjør det samme. Hvis én verdi fungerer rett, vil prinsipielt alle andre verdier i klassen gjøre det. Her er et eksempel:

Gitt er en regel for billettprisen ved et billettsystem:

- Ved en alder under 5 år skal billetten være gratis.
- Fra og med 5 år til opp under 16 år skal det selges barnebillett.
- Fra og med 16 til opp under 67 år skal det selges voksenbillett.
- Fra og med 67 år og oppover skal det selges honnørbillett.

Denne regelen fører til fire ekvivalensklasser:

- EK1: 0 <= alder < 5 år
- EK2: 5 <= alder < 16 år
- EK3: 16 <= alder < 67 år
- EK4: 67 <= alder

Dette betyr at vi burde bruke minst fire forskjellige verdier for å teste. Vi kan også kontrollere at samlingen av ekvivalensklassene inneholder alle mulige verdier, uten overlapp. Når en setter opp klassedefinisjonen like formelt som vist her, vil dette være enkelt. Det eneste foreløpig ubesvarte spørsmål er hva som er høyest mulig alder, men dette ignorerer vi foreløpig.

En kan så velge fire verdier for å teste, for eksempel 3, 7, 25 og 80. Det er viktig å forutsi resultatet, slik at vi kan sjekke det virkelige resultat mot det forutsagte. Ellers er det lett å bare godta det som systemet gir som output, selv om det ikke er rett. Altså burde testtilfellene defineres slik:

- T1: Input: 3, Forventet output: 0 kroner
- T2: Input: 7, Forventet output: 25 kroner
- T3: Input: 25, Forventet output: 50 kroner
- T4: Input: 80, Forventet output: 25 kroner

Her er antatt at billettprisene skal være hhv. 0, 25, 50 og 25 kroner.

Når en tester programmet med disse fire testtilfellene, vil en sjekke at programmet gjør det det skal. Her har en dog bare brukt "rett" input. En har aldri testet med feil input.. Testen benytter "gyldige" ekvivalensklasser. For å finne hva som skjer ved feil input, burde en teste "ugyldige" ekvivalensklasser i tillegg. Dette kan være at input ligger utenfor definisjonsområdet eller er rett og slett feil.

Her må en se hvordan input blir gitt. Hvis dette er en billettautomat og en skal oppgi alderen sin i år, kan følgende "ugyldige" klasser tenkes:

- EK5: Verdien er et negativt tall
- EK6: Verdien er over maksimal alder (for eksempel 120)
- EK7: Verdien er ikke oppgitt som tall (for eksempel * eller ?)

Avhengig av hvordan systemets grensesnitt er laget, vil det kanskje være umulig å bruke slike feile input, men en skal i hvert fall tenke på disse. Da vil en sjekke at programmet ikke gjør noe det ikke skal gjøre. Følgende tre ekstra testtilfeller kan være aktuelle:

- T5: Input: -4, Forventet output: Feilmelding
- T6: Input: 130, Forventet output: Feilmelding
- T7: Input: **, Forventet output: Feilmelding

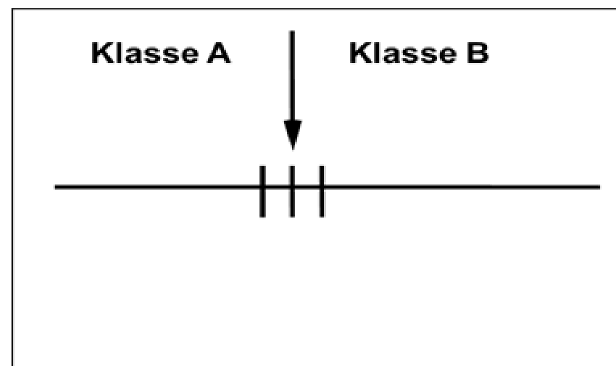
Slike ekstra testtilfelle hjelper til å finne flere feil og lage programmer med mer nøyaktig definisjon av grensesnitt, noe som bidrar til robusthet og dermed kvalitet. Veldig ofte er det feil i feilhåndteringen.

Grenseverdianalyse

Det har vist seg at feil ofte ligger ved eller rundt grensene, for eksempel grensene mellom ekvivalensklasser. Dette utnytter teknikken kalt ”grenseverdianalyse”.

Teknikken tar utgangspunkt i at folk er dårlige til å skille mellom grenseverdier som er inkludert og grenseverdier som ikke er det. For eksempel ”til” og ”til og med”. I tillegg er der en felle ved tabeller og lister der programmeringsspråk starter med indeks 0, mens vi vanligvis starter tellingen på 1. Dette resulterer i ”off-by-one”-feil.

Grenseverdianalyse lar seg illustrere slik:



For hver grense burde en altså teste tre verdier:

- like under grensen
- på grensen
- like over grensen

For billettsystemets grense mellom barne- og voksenbilletter kunne dette være følgende tre testtilfeller:

T11: Input: 15, Forventet output: 25 kroner
T12: Input: 16, Forventet output: 50 kroner
T13: Input: 17, Forventet output: 50 kroner

Strengt tatt er det siste testtilfellet unødvendig, fordi T11 og T12 til sammen tester begge ekvivalensklassene. T13 gir dog en ekstra sikkerhet mot eventuelle feil. I tillegg er det ikke sikkert alderen blir oppgitt i hele tall. Et alternativ kunne være å teste med en person med alder 16 år minus en dag, en person med alder 16 år eksakt, og en person som er en dag eldre enn 16 år.

For tabeller og lister bør en sjekke ut hva som skjer med første og siste element. Dette for å finne om det er ”off-by-one” feil i programmet. Slike feil kan for eksempel gi følgende utslag: I et røntgensystem blir alle bildene som er tatt for en pasient lagret tilhørende pasienten, bortsett fra siste bilde. Dette blir tilordnet neste person. Og så videre … Hvis bildene da er tatt av hhv. venstre og høyre hofteledd og bildene blir brukt til å avgjøre hva som skal opereres, kan dette ha mindre hyggelige konsekvenser.

Et problem med både ekvivalensklasseanalyse og grenseverdianalyse er at kombinasjoner av flere inputparametere ikke tas hensyn til. For å oppnå dette behøver vi andre, mer avanserte metoder.

Bruk av beslutningstabeller

Hvis vi skal teste om kombinasjoner av verdier av flere felt blir behandlet rett, kan dette gjøres ved hjelp av beslutningstabeller.

En slutningstabell består av fire deler:

- En liste over inputbetingelser eller input-ekvivalensklasser
- En liste over outputs eller output-ekvivalensklasser
- En liste over inputkombinasjoner
- En liste av tilhørende output

Dette kan illustreres slik:

Input-ekvivalensklasser	Inputkombinasjoner
Output-ekvivalensklasser	Outputs

Billettsystemet kan inneholde følgende betingelse:

Voksne kan reise med barnebillett dersom de reiser utenfor rushtiden (kl 10 til 14). Er de registrerte kunder, får de i tillegg en gratis barnebillett i denne tiden.

Tabellen vil da se slik ut:

Input/Output	Test 1	Test 2	Test 3	Test 4
Registrert kunde?	Ja	Nei	Ja	Nei
10-14	Ja	Ja	Nei	Nei
Voksenbillett	Nei	Nei	Ja	Ja
Barnebillett	Ja	Ja	Nei	Nei
Ekstra barnebillett	Ja	Nei	Nei	Nei

En ser at de fire mulige kombinasjonene av input gir tre forskjellige kombinasjoner av output. Kombinasjonene burde testes.
Istedenfor "Ja" og "Nei" kan en også bruke avkryssing ("x").

Hver kolonne på høyre siden av tabellen tilsvarer et testtilfelle. De forventede resultatene kan en lese direkte i tabellen. Hvis det er flere inputbetingelser, blir tabellen lengre. Vi må da legge inn alle kombinasjoner av alle input. Ofte går det dog å slette kolonner dersom de gir samme output. Ved omfangsrike tabeller blir dog metoden vanskelig å bruke. Typisk vil mer enn omtrent 30 kolonner kreve verktøystøtte.

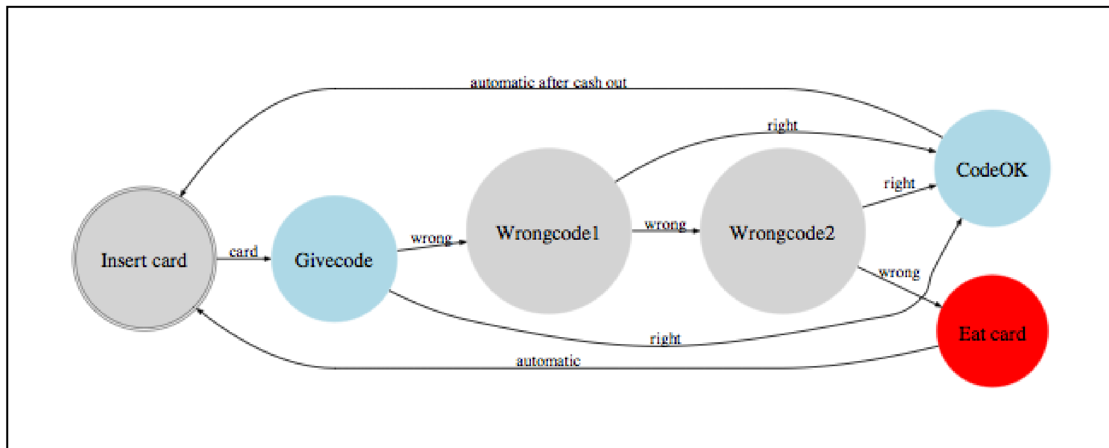
Teknikken tillater å sjekke om alle kombinasjoner av input blir håndtert på rett måte, uten redundans. Den egner seg godt til spesifikasjonsarbeid i tillegg til testarbeid. Det finnes også andre teknikker for å håndtere kombinasjoner. Disse er beskrevet i videreførende litteratur.

Test av tilstandsbaserte programmer

Mange programmer skal passe på en rekkefølge av handlinger, dvs. de skal "huske" ting som har skjedd før. Dette kan en modellere ved hjelp av tilstandsdiagrammer, der bokser er "tilstander" og piler er overganger. Et eksempel er dialogen ved minibanker.

Grovt forenklet er betingelsene slik:
Kunden skal møte et startskjermbilde ("Velkommen til minibanken, sett inn ditt kort"). Når kortet settes inn er det tre mulige forsøk på å angi PIN-koden. Det skal ikke være mulig å avbryte dialogen mens man prøver å oppgi koden. Er koden rett (første, andre eller tredje gang), så skal maskinen gå videre til dialogen vedr. utbetaling, hvoretter maskinen returnerer til å vise startskjermbildet. Er koden tre ganger feil, skal maskinen "spise kortet" og så returnere til startskjermbildet.

Dette kan en illustrere ved nedenstående tilstandsdiagram.



Typiske grunnleggende testkriterier for tilstandsdiagrammer er at hver boks og hver pil forekommer i testen. Det finnes mer omfattende kriterier beskrevet i litteraturen. Men for å komme til hver boks og hver pil, altså hver tilstand og hver overgang, kan en utføre følgende test:

- T1: Putt inn bankkort, gi rett PIN, få penger
- T2: Putt inn bankkort, gi feil PIN, gi rett PIN, få penger
- T3: Putt inn bankkort, gi feil PIN, gi feil PIN, gi rett PIN, få penger
- T4: Putt inn bankkort, gi feil PIN, gi feil PIN, gi feil PIN, kortet blir ”spist”

Denne testen vil utføre ”alt” i diagrammet. En må passe på at ikke bare sluttresultatet stemmer, men at programmet gir de rette resultatene ved hvert steg imellom også. Slik vil en finne mulig tilstandsfeil eller ”tilstandskorrupsjon”. Legger vi til flere betingelser og tegner de inn i diagrammet, vil vi måtte utføre flere tester. Det er en fordel hvis hvert testtilfelle utfører programmet slik at det kommer tilbake til utgangstilstanden. Dette muliggjør å ”henge på” andre testtilfelle uten særlige problemer. (En slags høflig oppførsel for en tester: Gjør noe – rydd opp etter deg!)

Kodedekning

De til nå beskrevne testteknikkene tar alle utgangspunkt i spesifikasjonen. Programmerere som tester koden kan også bruke kodedekning som et kriterium. Ideen er at kvaliteten til kode som aldri er utført i test er ukjent og dermed en risiko. Det eksisterer mange kriterier en kan bruke, men de to mest grunnleggende er at alle programinstruksjoner (grovt sagt alle kodelinjer) har blitt utført, og at alle beslutninger i har blitt utført i alle retninger (for eksempel ved en IF-instruksjon begge veier, true og false). Disse kriteriene vil ikke finne alle feil, men generell erfaring er at kvaliteten på koden øker når en har testet etter disse kriteriene.

Måling av slik testdekning er enkelt å utføre med moderne verktøy. De fleste programmeringsverktøy inkluderer muligheter til automatisk måling. Et ofte brukt kriterium er krav om 100% programinstruksjonsdekning og 85% beslutningsdekning. Kode som da ikke er utført blir kontrollert manuelt.

Ved høyrisikoprogramvare kan en bruke enda flere kriterier.

Det er dog viktig at en programmerer ikke ser seg blind på disse kriteriene og tror at alle feil er funnet. Typisk er det best å lage en spesifikasjonsbasert test først, og så å utvide den ved ekstra testtilfeller for å dekke koden. Slik dekning av kode blir også kalt "white box testing" (i motsetning til "black box testing" som er ulike teknikker for å dekke spesifikasjonen).

Erfaringsbasert test

De før behandlede testteknikkene er systematiske. I tillegg finnes usystematisk test som er basert på testerens intuisjon og erfaring. Den vil variere fra person til person, noe som betyr at en ikke alltid kan stole på den. Men erfarne personer er i stand til å finne mange feil relativt fort ved slike metoder.

Man baserer seg på viten om typiske feil og typiske plasser der det forekommer feil. Testeren utfører så testtilfeller som prøver å få programmet til å utføre slike feil.

Eksempler er å kjøre beregninger med input null, en og minus en samt meget store tall; prøving av æ, ø, å og andre spesialtegn ved tekstfelter; sortering av allerede sorterte eller motsatt sorterte tabeller; bruk av verdier en tror kan være misbrukt av programmereren i en spesiell betydning (populære er verdier som 99, 999, etc.) og mer. Et annet eksempel er å bruke "turer" gjennom et program, dvs. mange forskjellige funksjoner etter hverandre.

Disse metodene er ikke systematiske, dvs. de er vanskelige å styre. Men de er ofte brukbare i tillegg til de systematiske testteknikkene.

Testprioritering ved risikobasert testing

Å utføre testene kommer typisk sist i et utviklingsforløp, rett før et program skal leveres. I denne fasen av prosjektet kommer følgene av alle forsinkelser sammen, slik at testutføringen står under tidspress. For likevel å gjøre en best mulig testjobb finnes en teknikk kalt ”risikobasert testing”. Det en gjør er å identifisere områder med høyest risiko og teste disse først og mest omfattende, for så å teste andre ting etterpå, og kanskje mindre grundig. En sikrer med dette at en har noe informasjon om områder med høy risiko. Slik kan en få gode argumenter for enten å stoppe et prosjekt eller utsette leveransen (og teste mer).

Det en gjør er å analysere risiko og bruke det i testarbeidet.. Teknikken har følgende tre skritt, som utføres iterativt:

1. Identifisere risiko
2. Analysere risiko
3. Behandle risiko

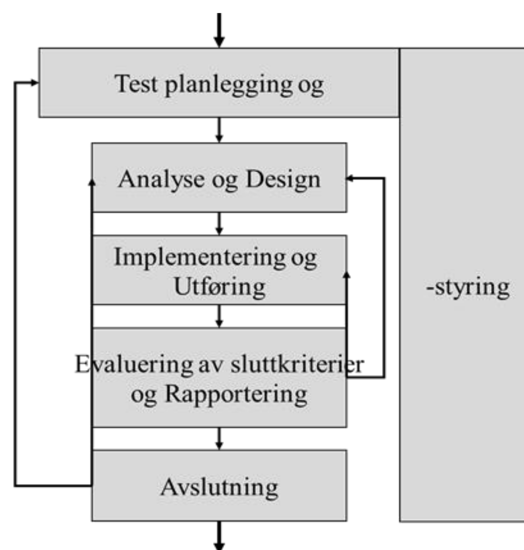
Risiko betyr det matematiske produkt av sannsynlighet for en negativ hendelse og dens konsekvenser eller kostnader.

Identifikasjon foregår ved brainstorming, diskusjon eller ved hjelp av sjekklister. Analyse finner ut hvor sannsynlig og skadelig hver risiko er. En uformell metode er å bruke en klassifisering som t-sjorte-størrelser, altså fra XS til XXL. Deretter bestemmer en tre risikonivåer og hva en gjør med dem. Behandlingen av dem er at en tester de høyeste risiki omfattende og først, de andre senere og mindre grundig.

Testprosessen og dens sammenheng med utviklingsarbeidet

Testingen består ikke bare av å utføre testtilfeller. Spesielt i større eller sikkerhetskritiske prosjekter er testingen et omfattende arbeid som krever systematikk. Her hjelper en systematisk testprosess. En slik testprosess er bl.a. beskrevet hos ISTQB (International Software Testing Qualifications Board, www.istqb.org) og i den internasjonale standarden ISO/IEC 29119.

Prosessen som ISTQB definerer er beskrevet nedenfor.



Prosessen består av fem hovedfaser med mulighet for iterasjon. Enkelte iterasjonsmuligheter er utelatt for å ikke gjøre figuren for komplisert.

I fasen ”Test planlegging” skal det tenkes gjennom hvordan testarbeidet skal gjøres, ressursbehov, hva slags risiko som forekommer og hvordan en skal håndtere den, og hva som er viktig eller mindre viktig å teste. Det er også viktig å definere hva testingen ikke skal dekke, for å holde kontroll med forventningene. Testplanleggingen dokumenteres i en testplan.

”Styring” er hengt på denne fasen, men foregår parallelt med testarbeidet gjennom hele prosjektet. Her følger vi opp hvor vidt testarbeidet foregår etter planen og tilpasser eventuelt planen eller modifiserer måten arbeidet blir gjort på. Typisk er både planlegging og styring aktiviteter for en testleder.

Neste fase, ”Analyse og design” vedrører den tekniske delen av arbeidet. I analysen finner vi ut hva som egentlig er innholdet i testingen. Vi definerer i detalj hva som skal testes, velger metoder og teknikker samt verktøy. Design betyr hovedsakelig å lage konkrete testtilfeller. De tidligere beskrevne testteknikkene hjelper til å utføre denne oppgaven. Resultatet er testtilfeller og testdata samt en spesifikasjon av nødvendige testmiljøer og verktøy.

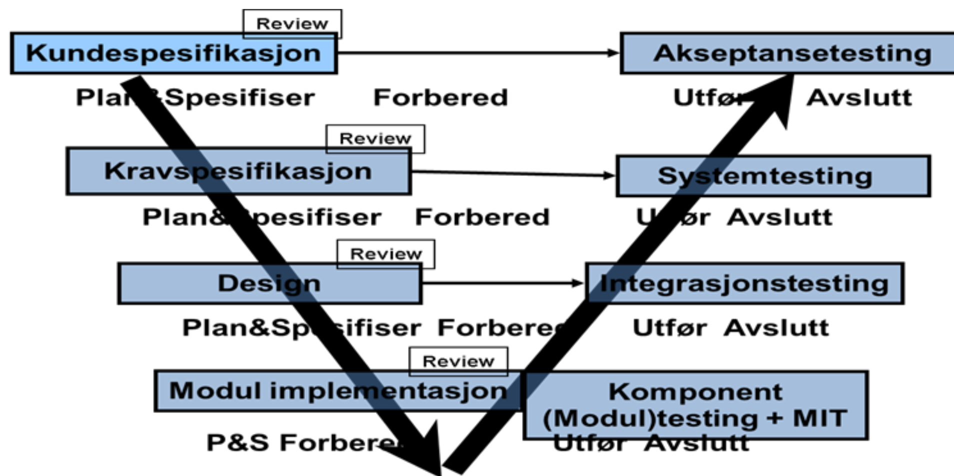
Tredje fase, ”implementering og utføring” er sentral. Implementering betyr å gjøre alt ferdig til å utføre testene. Dette kan være å beskrive testene detaljert nok til manuell kjøring, men det kan også bety å kode testene for kjøring med verktøy. I tillegg må en sette opp testmiljøet og gjøre alt klart. ”Utføring” er best kjent i praksis, fordi det alltid gjøres, mer eller mindre systematisk. Her kjører en de testtilfelle som er definert og sjekker resultatene mot forventede resultater.
Et stort problem kan være det såkalte ”testorakel”, måten en bestemmer om et resultat er rett eller feil. Hvis en ikke har tenkt på dette før, kan testen blir ganske unøyaktig: En sjekker bare om systemet oppfører seg ”fornuftig”, sett ut fra testerens synspunkt. Mange feil blir da ofte oversett.

Fjerde fase, ”sjekking av sluttkriterier og rapportering”, er i prinsipp liten. Sluttkriteriene, dvs. kriteriene for når en kan avslutte testingen, burde være beskrevet i testplanen og godkjent av prosjektledelsen. Eksempler på slike kriterier er en gitt dekningsgrad er oppnådd, testing er utført i henhold til testplan, testingen avdekker få eller ingen feil.
En må bare kontrollere at sluttkriteriene er oppnådd. Disse sikrer forhåpentligvis at målene med testingen er oppnådd også. Basert på analysen kan en kjøre en ekstra test eller en diskuterer endring av sluttkriteriene.

Testrapporten skrives typisk av testlederen. Den skal være en kort oppsummering. Det viktigste er en beskrivelse av risikoen som er funnet med systemet under test, som munner ut i en anbefaling til prosjektledelsen om produktet bør frigis eller ikke.

Den siste fasen, ”avslutning” er stort sett en opprydding. Testmaterialet arkiveres, fordi det kan være aktuelt å benytte om igjen i vedlikehold av produktet. Det bør også kjøres en oppsummering, en retrospektive eller liknende, for å trekke lærdom av den testjobben som er gjort. Til slutt bør avslutningen feires.

Denne testprosessen er anvendbar i alle testnivåer, fra utviklertesting (modultesting) til akseptansetesting ved kunden. For å beskrive hvordan test og utvikling bør koordineres, har en V-modellen. V-modellen beskriver hvordan test og utvikingsarbeid henger sammen. Den har fått sitt navn etter hvordan den ser ut, nemlig som en V (se figuren lenger nede). Egentlig finnes det mage varianter av V-modeller, alt ettersom hva som er de konkrete behovene. Men prinsippene er forklart ved hjelp av figuren nedenfor. Modellen fungerer like bra med smidige utviklingsmetoder som med tradisjonelle.



Vanligvis er det fire nivåer (antall nivå kan variere). Kundens spesifikasjon er grunnlag for akseptansetestingen. Denne testen skal sjekke om systemet oppfyller kundens behov og krav og om systemet er brukbart i praksis. IT-spesialister utvikler så en kravspesifikasjon som er mer teknisk i sin natur. Den er grunnlag for en systemtest, der hele systemet blir sjekket opp mot kravene definert i kravspesifikasjonen. Hvis systemet er av mer enn triviell størrelse, behøves en designbeskrivelse, eller en beskrivelse av systemets arkitektur. I den tilhørende integrasjonstesten blir det sjekket hvor vidt enkeltbitene, delsystemene, passer sammen og dermed oppfyller de krav som er satt i design. Til slutt blir modulene utviklet, eventuelt med en detaljdesign først, så programkode. Disse detaljene blir testet i en modultest, modulene blir eventuelt integrert i en modulintegrasjonstest (vanligvis uten at dette blir nevnt som en offisiell test). Vi har altså flere forskjellige testnivåer med hver sin egne oppgave. I tillegg skal hvert testnivå være sikkerhetsnett for feil som har overlevd i testene på et lavere nivå.

Det er en stor fordel at testingens planlegging, spesifikasjon og design starter tidligst mulig[1]. Dette muliggjør tidlig feedback til de som utvikler. Testforberedelse fører til at en gjennomtenker det som skal utvikles. Feilene blir dermed i stor grad funnet når en beveger seg nedover nivåene i modellen, ikke når en beveger seg oppover igjen og utfører test. Et tilleggsmoment i nedenstående tegning er bruk av inspeksjoner/reviews: Testere kan med fordel brukes for å gjennomgå og sjekke utviklingsdokumenter. Det er bevist, spesielt i kritiske systemer, at modellen fungerer, men den krever disiplin og spesielt god endringskontroll. Modellen kan i modifisert form brukes i smidige prosjekter. En utfører da en ”V” i hver iterasjon, og testmaterialet oppbevares for omkjøring (regresjonstest) i de neste iterasjonene.

[1] Modellen bruker en annen betegnelse enn fasene i ISTQB-modellen. Dette er gjort for å holde teksten kort nok, slik at den fortsatt er lesbar.

Startkriterier for testutføringen

Et stort problem ved utføringen av testene er ofte at programvaren egentlig ikke er klar til testing. Eksempler kan være at den inneholder alt for mange feil eller at tidligere test ikke er utført. Men det kan også være problemer med testmiljø, verktøy etc. Derfor er det en stor fordel at testeren definerer hva som skal til for å utføre testen og at ansvar for disse kriteriene blir delt ut til de rette personene.

Eksempelvis kan start for utføring av systemtest kreve følgende:

- Koden bør ha vært utsatt for verktøystøttet statisk analyse og gjennomgang, være godkjent og de vesentlige feil utbedret.
- Koden har vært testet i modultest og integrasjonstest med tilstrekkelig testdekning og de alvorligste feil er reparert.
- Testmiljø er satt opp og prøvd for funksjon.
- Testverktøy er tilgjengelige og utprøvd.
- Det finnes et fungerende system for behandling av feil som avdekkes.
- Konfigurasjonsstyring og endringskontroll fungerer.

Testlederen bør i hvert fall sette opp de startkriteriene som erfaringsmessig kan føre til trøbbel i organisasjonen.

Hvordan bedømme testingens kvalitet i en organisasjon?

I og med at testingen er en ganske vesentlig del av arbeidet med å utvikle eller vedlikeholde programsystemer, er det interessant å vite hvor bra testingen utføres, og eventuelt forbedre den. For å gjøre dette kan en bruke forskjellige referanser.

En populær modell er beskrevet av firmaet Sogeti og heter TPI-Next. TPI står for ”test process improvement” og ”next” betyr at dette ikke er første versjon av modellen. Modellen er beskrevet på tilgjengelige dokumenter på Internett og i tillegg i en lærebok. 16 såkalte ”key process areas” blir bedømt ved hjelp av sjekkspørsmål. En finner så ut på hvilket nivå i modellen en ligger. Modellen gir da informasjon om hva som lønner seg å forbedre i neste omgang.

En annen modell heter TMMI (Test Maturity Model Integration). Den er laget så mye som mulig i overensstemmelse med den kjente prosessforbedringsmodellen CMMI (Capability Maturity Model Integration). Her blir også ulike nøkkelområder bedømt og veiet, og modellen oppgir hva som burde forbedres for å komme opp i høyere modenhet. Høyere modenhet betyr i prinsipp bedre forutsigbarhet, kvalitet og produktivitet på arbeidet.

Flere modeller finnes, og innen ISTQB er det definert et pensum for testprosessforbedring.

Testverktøy og automatisering av testarbeid

Testarbeidet er omfattende, og flere sider av den kan støttes med verktøy. Verktøy er dog først interessant dersom en har en rimelig ordnet testprosess. Så lenge den er kaotisk, dvs. så lenge en kjemper for å teste i det hele tatt, gir automatisering bare hurtigere kaos (ifølge Dorothy Graham, en internasjonalt kjent "testguru"). Men har en litt orden, kan følgende verktøy anbefales.

1. Forvaltning av feil
2. Forvaltning av testmateriale / testledelsesverktøy
3. Verktøy til støtte for testutvikling
4. Verktøy for statisk analyse
5. Verktøy for testutføring
6. Verktøy for automatisering av testutføring
7. Verktøy for ytelsestest og andre egenskapstester

Litt detaljer i det følgende.

Forvaltning av feil

Slike verktøy holder rede på alle feil som er funnet. Feilene blir fulgt opp til de eventuelt er rettet og testet, godkjent og ny programversjon distribuert. Slike verktøy er ikke bare aktuelle under testing. De er i aller høyeste grad viktige for oppfølging av programmer i drift. En testleder får med slike verktøy kontroll over feilhåndteringen.

Forvaltning av testmateriale / testledelsesverktøy

Testmaterialet er omfattende. Det kan være snakk om tusenvis av testtilfelle, grunnlagsdata, filer, testmiljøer og verktøy samt medarbeidere som skal ta seg av alt dette. På en måte er slike verktøy en testleders "dashboard".

Verktøy til støtte for testutvikling

Så snart en bruker mer avanserte metoder enn de som er beskrevet lenger oppe i dette dokument, vil en måtte støtte testutviklingen med verktøy. Disse er spesielt anvendelige når en skal teste kombinasjoner. Slike verktøy kan delvis automatisk generere testmateriale og grunnlagsdata.

Verktøy for statisk analyse

Statisk analyse betyr at verktøy kontrollerer koden mot et sett med regler for å unngå en del feil. Eksempler er variable som aldri fikk en initialisering, kode som ikke er flyttbar, kode som gir sikkerhetshull eller kode med åpenbare feil i kontrollflyten. Statisk analyse arbeider på prinsipielt samme måte som stave- og grammatikkontroll i verktøy for tekstbehandling.

Verktøy for testutføringen

Viktige verktøy her er sjekking av kodedekningen under test, men også alle slags "spioner" og loggeverktøy som kan se på viktige egenskaper av et program under test. Et eksempel er kontroll av minne- og ressurslekkasjer.

Verktøy for automatisering av testutføring

Testutføringen og resultatsjekkingen, når den blir gjort manuelt, er arbeidskrevende og upålitelig. Det er fristende å automatisere slikt arbeid. Dette lover spesielt bra hvis systemet skal vedlikeholdes og det er et stort behov for regresjonstest (omkjøring av eksisterende tester for å finne eventuelle bivirkninger av endringer). I tillegg er dette aktuelt ved plattformtest, dvs. hvis et system skal leveres på mange plattformer. Typisk ved kommersiell programvare. Automatisering er dog vanskelig og krever både erfaring og en bra arkitektur. Det er mange organisasjoner som kaster bort store ressurser med naiv testautomatisering her.

Verktøy for ytelsestest og andre egenskapstester

De fleste programvare-egenskapene er vanskelig å teste manuelt. For eksempel finnes det gode verktøy for å simulere store antall parallelle brukere ved en ytelsestest. Det samme gjelder verktøy for sikkerhetstest. Slike verktøy er dog meget avhengige av krav og plattform.

Mange verktøy er nå tilgjengelig med åpen kildekode eller som tjenester på nett.

Hvordan finner du ut mer om software testing?

Det er mange kilder til mer kunnskap. Noen fagbøker er klassikere. Et bra eksempel er boken ”The Art of Software Testing” av Glenford Myers. Boken kom i 1979 og en nyere oppdatert utgave er tilgjengelig. Andre forfattere som har skrevet mye om å lage gode testtilfelle er Boris Beizer og Lee Copeland. På Testledelsesområdet finnes en bra bok av Kaner, Falk og Nguyen. Rex Black har også skrevet flere gode bøker på området testledelse. James Whittaker beskriver intuitivt og erfaringsbasert test i flere bøker med titler som alle begynner med ”How to break…”.

Det finnes et internasjonalt sertifiseringsprogram for testere i regi av ISTQB. På www.istqb.org finnes mange gode referanser. En bra lærebok for det grunnleggende viten er Spillner, Linz, Schaefer, ”Software Testing Foundations”, Rocky Nook 2011 og 2014. Norsk pensum for ISTQB Foundation nivå finner du på www.istqb-norge.no sine sider for dokumentasjon.

Du kan også se om det finnes ressurser i ditt lokalmiljø. Den Norske Dataforening har flere lokale faggrupper som arrangerer foredrag og kurs i software testing.

Men det viktigste er at du har den rette instillingen: En testers første bud er skepsis: Du skal ikke tro på noe før du har testet det!

Lykke til med arbeidet!