Amna Dastgir

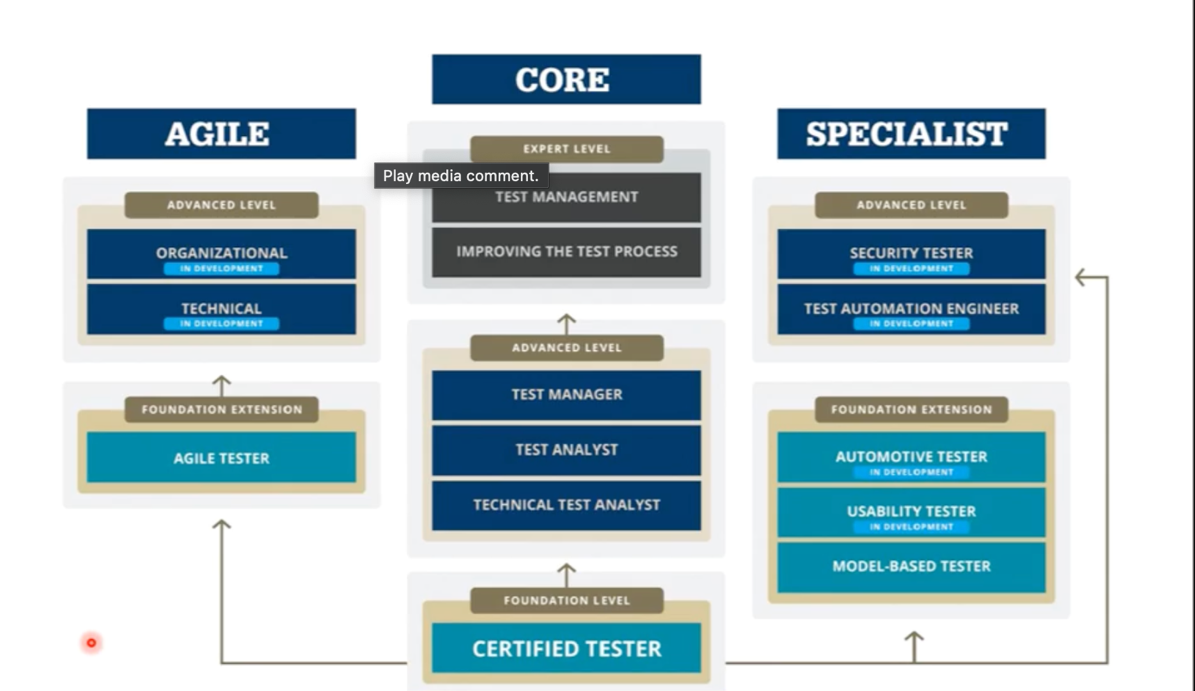
Testing av programvare

Eksamen 24 mai

2.år Bachelorstudium IT

**GRUNNLEGGENDE TESTPRINSIPPER**

ISTQB (International software testing qualification board) 🡪 samling av profesjonelle testere fra over 80 land som har satt et teoretisk rammeverk om hvordan man bør teste. Man kan gå opp til testeksamener for å bli sertifisert tester (da finnes det ulike nivåer). Vi skal se foundational level.



**HVORFOR TESTER VI?**

* Fordi mennesker gjør feil hele tiden
* Vi må finne feil som er gjort
* Vi må sikre at brukeren er fornøyd
* Sikre kvaliteten i systemet
* Sikre god ytelse på løsningen

**DEFINISJON PÅ TESTING**

Testing er en prosess for å verifisere og validere at et produkt, system eller en enhet oppfyller spesifiserte krav og fungerer som forventet. Det kan inkludere manuelle tester utført av en tester, eller automatiserte tester som kjører ved hjelp av en programvare.

**TESTINGS ROLLE I PROGRAMVAREUTVIKLING**

Testing er viktig i de aller fleste stegene i livet til et programvareprodukt

1. utvikling
2. produksjon
3. vedlikehold

Rollen testing har:

1. redusere risikoen for at feil oppstår under produksjon
2. sjekke at programvare oppfyller kravene til
   1. brukerne
   2. lover og regler (f.eks universell utforming)

**MÅL FOR TESTING**

Ulike formål 🡪 finne feil, få tillitt til kvalitetsnivå, gi informasjon for å danne et beslutningsgrunnlag, forebygge feil.

Ulike mål for ulike typer testing:

1. Enhetstest 🡪 finne så mange feil så tidlig som mulig
2. akseptansetest 🡪 for å bekrefte at systemet fungerer som det skal, når vi aksepterer systemet, f eks når en kunde får prøvd det ut
3. vedlikeholdstest 🡪 sjekke at ingen nye feil er introdusert.

**TESTING OG KVALITET**

Man kan delvis øke kvalitet inn i løsningen ved å teste den, men det er viktigst å bygge kvalitet når den utvikles. Vi kan måle kvalitet ved å referere feilene som er funnet og sette disse opp mot de krav som er satt til løsningen. Dette er både for funksjonelle (lagre kunde) og ikke-funksjonelle krav (ytelse skal være å håndtere mange samtlige brukere) som gir trygget på kvaliteten når det ikke gjenstår mange feil.

**NÅR STOPPE Å TESTE?**

Da man har nok informasjon til å foreta beslutning om systemet kan settes i produksjon, eller starte ny sprint. Det avhenger av hvilke risiko som ønskes å ta teknisk og funksjonelt.

Når feil-raten i enkelte områder går ned så kan det bety at vi ikke tester dette området bra nok.

Hvis vi får null feil ved systemet, ved å rette og teste, rette og teste. Feil finnes alltid.

**7 PRINSIPPER FOR TESTING**

Utforske et system/systemkomponenter for å avdekke feil + det som mangler

P1 peil = testing kan bare vise at feil eksisterer, men ikke bevise at et program er feilfritt.

P2 Fullstendig testing er umulig = det er umulig å teste alt. Testing burde prioriteres ut ifra hva man skal teste og hvor stor risiko det er. Da måler man risikoen mellom hvor stor sannsynligheten er for at en feil oppstår (faktor 1) med konsekvensen feilen får (faktor 2).

P3 Tidlig testing = still kritiske spørsmål allerede i begynnelsen: skal vi virkelig utvikle det her? Er det noe poeng? Kravspesifikasjonen bør være godt forstått. Ved gransking av kravspesifikasjon, så er det spesialt mangler og inkonsistens man må være bevist på.

Det er mye som skjer før man begynner å implementere et program (alt fra planlegging, kravspesifikasjon, funksjonelle krav og ikke funksjonelle krav).

* Allerede i planleggingsfasen kan man kartlegge testteknikker som skal benyttes for å kvalitetssikre og fjerne mest mulig feil i startfasen.
* Feil utvikles raskt og i hverandre, og må derfor oppdages tidlig.
* Økonomisk gevinst: avdekke feil under utvikling er ofte rimeligere å håndtere enn når systemet er tatt i bruk.

P4 Feilene i et program er ikke spredt jevnt utover, men har en tendens til å samle seg i enkelte moduler = Fordi deler av koden har fellestrekk.

Hvordan håndtere dette?

* Statisk analyse verktøy (kan avdekke kvalitet i koden):
  + Identifisere steder som det kan være fare for mye feil
  + Avdekke steder med høy kompleksitet i koden

P5 Planteverdmiddelparadoks = om vi tester og bruker de samme testmetodene, fjerner vi bare de type feilene vi har konsentrert oss om.

* Kan finnes andre feil som vi har oversett
* For å overvinne dette paradokset må vi stadig fornye testene våre og finne nye områder som vi tester på. Testene burde varieres og forbedres.

P6 testing er kontekst avhengig = noen programmer må være mer eller mindre feilfri i den grad det er mulig. F.eks.: vil ikke sikkerhetskritiske systemer virke på samme måte som handelssystemer.

P7 «Fravær av feil» - fellen = et feilfritt system er ikke nødvendigvis et brukbart system. Det må valideres og verifiseres.

* Validering: bygger vi det riktige produktet?
* Verifisering: bygger vi produktet riktig?

**Å FINNE FEIL KREVER:**

* Nysgjerrighet
* Profesjonell pessimisme
* Fokus på detaljer
* Gode kommunikasjons ferdigheter
* Erfaring i feilgjetting

**UAVHENGIGE TESTNIVÅER**

Det kan være effektivt at forskjellige personer gjør ulike testaktiviteter, ikke programmereren som f eks tester alt.

**TESTERENS ETIKK**

Ofte kommer man borti «live» data. Det vil si produksjonsdata som oftest er et system, som skal erstatte et annet system. Da konverterer vi dataene fra det gamle til nye systemet. Da kjører man produksjonstester som vil si at da systemet er klart, og man har akseptert systemer, så skal man sette produksjon, altså at det skal gå live, da kjører man noen produksjonstester før brukerne kommer på. De dataene i databasen kan være sensitive f eks innenfor sensitiv data (helse) og bank (elektroniske penger).

**GRUNNLEGGENDE TESTPROSESSER**

1. Testplanlegging- og styring

* Man skal planlegge testing og hvordan den skal styres.
* Plan 🡪 hva, hvem, når?
* Scope: hva skal dette omfatte, hvilket system?
* Mål og risiko (hva slags feil kan vi få eller er folk syke)
* Testnivåer og typer som vil bli brukt: enhetstester, ytelsestester
* Dokumentasjon som skal produseres
* Tildeling av ressurser for ulike testaktiviteter
* Tidsplanlegging: test-implementering, gjennomføring og rapportering
* Kontroll og justering: hvordan forbedre

1. Testanalyse- og design

* Planlegge å designe testene
* Man tar utgangspunkt i:
  + Kravspesifikasjoner – brukerhistoriene
  + Grensesnittspesifikasjoner – for å teste imot kravspesifikasjonene
  + Risiko – er det enkelt av deler osm er kritiske? Skal vi teste mer der
  + Design og arkitektur – hvordan er systemet er laget, hvilke lag er i applikasjonen og teste ut ifra det
* Evaluering av testbarheten til de ulike testobjektene – hvor enkelt er det å teste testobjektene? Metoder, grensesnitt osv
* Design og valg av testmiljø og verktøy
* Design av sporbarhet mellom testbasis og testtilfeller

1. Testimplementering- og utførelse

* **Implementere** planene som er satt
  + Implementere og prioritere testtilfelle / test-suits / cases inkludert testdata
  + Skrive automatiserte testscenarioer
* **Utføre**
  + Kjøre testene
* **Rapportere**
  + Dokumentere kjøringer og avvik
* Gjenta alle stegene for korrigerte avvik

1. Evaluering og rapportering

* Evaluere resultater mot kriterier for avslutning av test og rapportere til beslutningstakere
  + Kan systemet gå i drift?
  + Kan man gå til neste sprint?

1. Avslutningsaktiviterer

* Gå gjennom hva som gikk bra og dårlig, dokumentasjon om hva som bør endres/forbedres

**TESTING GJENNOM LIVSSYKLUSEN**

**Testteknikker – 2 hovedgrupper**

Teknikker vi kan bruke for å lage testene.

1. Statiske: tester kode uten å kompilere den
   1. reviews: granskinger
   2. statisk analyse (av kilde og generert output)
      1. ubrukte variabler, referanser til variabler med udefinerte verdi, brudd med kodestandarder, deadlocks (vranglås). Sårbarhet mtp sikkerhet.
2. Dynamiske: krever at vi kjører programmet (kompilerer koden)
   1. Blackbox: spesifikasjonsbaserte
      1. Tester funksjonaliteten til en applikasjon. Vi ser på hva vi sender inn til en modul også måler vi det som kommer ut. Vi ser ikke på koden som ligger bak.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

* 1. Whitebox: strukturbaserte
     1. Vi ser inn i boksen, hvordan noe er bygd opp. Vi tester den interne strukturen til et system.

Et bilde som inneholder diagram

Automatisk generert beskrivelse

**V-MODELLEN OG ALLE «NODER» -** benyttes enten i en iterasjon eller for hele systemet.

BRUKERKRAV: her setter man opp brukerkravene så fullstendig som mulig – hva skal systemet gjøre og hvordan?

SYSTEMKRAV: settes etter brukerkrav. Hvordan skal brukerkravene settes i bruk og implementeres. Krav til systemet og hva systemet skal gjøres settes. Fra brukerens og systemets synspunkt.

GLOBAL DESIGN: sette opp selve arkitekturen til programmet. Hvilke moduler: frontend, database, programlogikk osv. Hvordan er grensesnittet mellom de ulike modulene og hvordan er de satt sammen?

DETALJERT DESIGN: beskriver mer hvordan koden skal implementeres (ift klasser, klassehierarkier, datastrukturer osv.)

IMPLEMENTASJON: vanligvis benytter man enhetstesting for å teste implementasjonen hele veien ned venstresiden av V-modellen. Man benytter statiske testteknikker for å sikre at alt det som ligger i grunn er i orden før man begynner implementering. Testbasisen for systemet blir all den informasjonen man har og generer. Deretter så implementerer man, men man tester de med komponenttester/enhetstester.

ENHETSTESTING: tester alle delmodulene separat i programmet. Dette baserer seg på detaljert design og beskrivelse av kode.

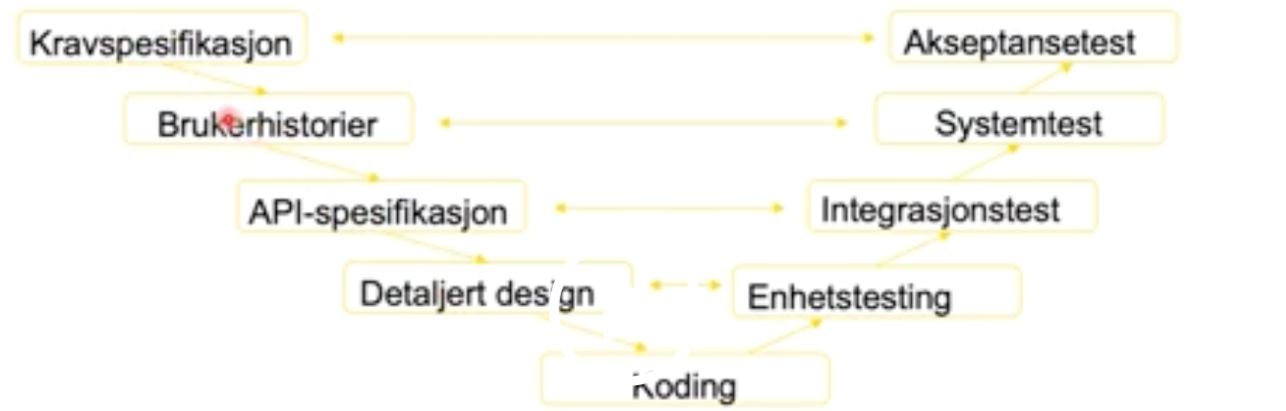
INTEGRASJONSTESTING: mellom hver modul vil det være et grensesnitt som må testes (eks: hvordan API henger sammen med frontend). Integrasjonstesing baserer seg på global design og man tester grensesnitt mellom ulike moduler.

SYSTEMTESTING: verifiserer hva systemet gjør ift beskrivelse i systemkrav. Tester funksjonaliteten til programmet og hvordan hele systemet oppfører seg.

AKSEPTANSETESTING: utføres av sluttbruker eller representanter for det kan hente at bruker har noen krav til brukervennlighet, responstid osv. Målet et å få tillit til systemet, spesielt når det kommer til de ikke-funksjonelle karasterikkene ved det.

**Pilene betyr at når vi har:**

1. laget brukerkrav kan vi forbedre akseptansetesting
2. satt opp systemkravene kan vi forbedre systemutesting
3. global design kan vi forbedre integrasjonstestingen
4. detaljert design på plass kan vi forbedre enhetstesting



**Inkrementell utvikling**

Systemet utvikler gradvis i form av nye inkrementer som legges til. Hvert inkrement evalueres før utviklingen av neste inkrement starter. Det er i smidige metoder. Selve evalueringen gjøres av en bruker eller kunderepresentant.

Et bilde som inneholder diagram

Automatisk generert beskrivelse

**KARAKTERISTIKKER FOR GOD TESTING**

* for hver utviklingsaktivitet er det en tilsvarende testaktivitet.
* Hvert testnivå har bestemte testmål.
* Analyse og design av testene for et gitt testnivå bør starte når den tilsvarende utviklingsaktiviteten utføres. Testere bør delta i gjennomgang av dokumenter så snart utkast er tilgjengelig.

**Testnivåer – 4 forskjellige**

1. komponenttesting/enhetstesting: når vi har testet de ulike modulene separat så
2. integrasjonstesting: tester vi grensesnitt mellom de ulike moduler/interaksjonen med andre systemer. Når alt fungerer så..
3. systemtesting: tester og verifiserer at programmet gjør det som er systemkravet
4. akseptansetesting: vi tester og bruker fra bruker perspektiv. Sjekker om mer må tillegges. Oppfyller systemet det systemkravet etterspør.

**Enhetstesting:** sjekker om hver enkelt komponent eller enhet av programvaren fungerer som forventet. Enheten kan være en funksjon, en metode eller et modul, og testen fokuserer på å sjekke at enheten oppfyller spesifikasjonene og gir forventet resultat for gitte inndata. Enhetstester utføres vanligvis av utviklere som en del av utviklingsprosessen for å sikre kvaliteten på koden og unngå feil og problemer når programvaren integreres.

**Integrasjonstest:** tester at forskjellige systemer eller komponenter fungerer sammen som forventet. Det kan være flere komponenter eller systemer som trenger å kommunisere og samhandle med hverandre, og integrasjonstesting hjelper til med å oppdage eventuelle feil eller problemer i denne samhandlingen. Integrasjonstester kan utføres på forskjellige nivåer av programvareutvikling, for eksempel enhetstesting, systemtesting eller akseptansetesting, og kan utføres manuelt eller automatisk ved hjelp av verktøy. Integrasjonstesting kan være hvit- og svartboks.

**Systemtest**: testing som fokuserer på å evaluere hele systemets funksjonalitet i en integrert måte. Målet er å teste systemet som en helhet for å avdekke eventuelle feil, mangler eller avvik fra kravene som er satt. Dette kan omfatte testing av systemets grensesnitt, funksjonalitet, ytelse, sikkerhet og pålitelighet. System testing utføres vanligvis etter at enhetstesting og integrasjonstesting er fullført, og det kan inkludere både manuelle og automatiserte tester.

**Akseptansetest**: testing som utføres for å verifisere at et system eller en applikasjon oppfyller kravene og forventningene til kundene og brukerne. Testene blir utført på slutten av utviklingsprosessen for å sikre at systemet eller applikasjonen er klar for utrulling og bruk i produksjonsmiljøet. Resultatene fra akseptansetesting blir brukt for å avgjøre om systemet eller applikasjonen kan godkjennes og tas i bruk eller om det må gjøres endringer og forbedringer før det kan tas i bruk.

Selv om en liten feil oppdages på hele systemet testes!

**STATISK TESTING**

Statiske testtekniker er for eksempel manuell kontroll eller automatisk analyse av koden eller annen prosjektdokumentasjon uten at koden blir utført. Dette er imotsetning til dynamiske tester, slik som enhetstester, integrasjonstester og systemtester Hvor kodene kjøres under testingen. Med statiske tester kan vi finne feil før vi gjør det dynamiske testene. Dette vil si at vi kontrollerer alle dokumenter som for eksempel kravspesifikasjoner, designspesifikasjoner, koder, testplaner, testspesifikasjoner, test tilfeller, tests skripter, brukermanualer og websider.

**FEILENE SOM KAN FINNES:** Det er avvik fra standarder, feilaktige krav, feil i design, utilstrekkelig vedlikeholdbarhet, feilaktige grensesnittspesifikasjoner.

**GRANSNKNINGSPROSESSEN:**

Disse kan bestå av uformelle eller formelle granskinger. Det er selvfølgelig best med de formelle, og de er som oftest planlagte og dokumenterte gjennomganger. Disse består av:

1. Planlegging:

Her definerer man kriterier, man velger personer til, deler roller, man definerer start og slutt kriterier. Og så velger man hvilke deler av dokumentet en skal se på.

1. Oppstart:

Man forklarer målsetningene, prosessen og dokumentene til deltakerne.

1. Individuell forberedelse:

Før ulike møter bør man gå gjennom dokumentene og forberede seg selv.

1. Granskningsmøte:

Under møtene burde man diskutere stille spørsmål og notere feil.

1. Retting:

Og så burde man rette feilene som er funnet og alltid loggføre.

1. Oppfølging:

Og så burde man kontrollere at feilene er gjort noe med.

**ROLLER I FORMELL GRANSKNING**

* Man har en ***testleder*** som planlegger når og hvor, hvilke granskninger skal gjøres.
* Man har også ***granskingsleder***, altså en moderator, som styrer møtene og passer på oppfølging.
* Man har ***forfattere*** de som skriver dokumentet.
* Vi har ***revisorer*** personer med spesifikt Teknisk eller spesiell bakgrunn som kan foreta gjennomganger.
* Det finnes ***sekretærer*** som dokumenterer anmerkninger, problemer og åpne punkter.

Og selvfølgelig er det som mulig at flere roller i team.

**TYPE GJENNOMGANGER**

1. Uformell gransking som ikke er en formell prosess. Da kan vi for eksempel be en annen programmerer se på din kode eller en annen tester se på dine testcase.
2. Strukturert gjennomgang kan ledes av en forfatter, og det er et møte for å få innspill. Det kan gjøres formelt eller uformelt.
3. Inspeksjon ledes av en moderator der en formell prosess som baserer seg på regler og sjekklister.
4. Tekniske gjennomganger kan gjøres uformelt eller formelt med moderator.

Formålet er som de andre gjennomgangene å finne problemer og feil så tydelig som mulig.

**SUKSESSFAKTORER FOR GRANSKNINGER**

Det burde være klare definerte formål. Man burde ha riktige mennesker med nok kunnskap. File som oppstår burde rapporteres. Gransking burde bli en positiv opplevelse, ikke kritikk. Sjekklister bør brukes effektivt. Teknikker er vel bekjente. Ledelsen setter av tid og ressurser. Det legges også vekt på læring og prosessforbedring.

**STATISK ANALYSE MED VERKTØY**

Statisk analyse med verktøy kan gjøres av

* utviklere for å sjekke programmerings standarder
* designere under modelleringen av programvaren.

Statiske analyseverktøy kan produsere et stort antall advarsler.

**VERDIEN AV STATISK ANALYSE**

Tidlig oppdagelse av feil for test utføring. Tidlig varsel om mistenkelige egenskaper ved koden eller designet ved å beregne målverdier, som høy kompleksitet. Identifisering av feil som dynamisk testing ikke finner så lett. Forebygging av feil hvis en lærer av feilene under utvikling er dette veldig bra.

**TYPISKE FEIL MED STATISK ANALYSEVERKTØY**

1. Å referere en variabel med en udefinert verdi.
2. Avvik i grensesnitt mellom komponenter.
3. Variabel som aldri brukes eller er feil deklarert.
4. Kode som ikke kan utføres. Altså død kode.
5. Manglende og feil logikk, for eksempel uendelige løkker.
6. For kompliserte konstruksjoner.
7. For store metoder.
8. Avvik fra programmerings standarder.

**TESTTEKNIKKER**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**Testtyper – 4 forskjellige**

1. Funksjonell testing: tester hva systemet gjør/skal gjøre, slik at vi har beskrevet det i systemkravene. Verifiserer koden og passer på der. Bekrefter forventet input.
2. Ikke-funksjonell testing: går mer på hvordan systemet gjør det. Har mer med kvalitet og utformingen å gjøre.
3. Strukturell testing: handler om å bruke strukturer i koden eller i testbasis til å beregne hvor mye man har testet.
   1. Man understreker hvor stor testdekning man har (test coverage).
   2. Testing av hvor mange krav i kravspesifikasjonen som er testet
   3. Code coverage – hvor mye kode har vi dekket?
      1. Vi må ha noe tellbart – teller opp hvor mye vi kan teste med henblikk på et eller annet også ser vi hvor stor andel vi virkelig har testet
      2. Har vi bare testet halvparten har vi bare 50% dekning. Selv om vi tester alt betyr det ikke at programmet er feilfritt.
4. Endringsbaserte: disse skjer mer med forandringer i programmet.
   1. Confirmation testing: sjekker om det oppstår feil og fikse
   2. Regression testing: går ut på å teste hele eller deler av systemet

**Statisk og dynamisk testing – hva er forskjellen?**

|  |  |
| --- | --- |
| Statisk | Dynamisk |
| Testing uten at vi kjører programmet under test. Nedgående venstresiden av V-modellen | Krever at vi kjører programmet |
| Eks:   * gransking av testbasis eller statisk analyse som input til et analyseprogram og sjekker kvaliteten | Over i en annen type testing som foregår etter at vi har begynt å implementere, altså på høyresiden av V-modellen. |

**----------------------------------------- DEL 1 ----------------------------------------------**

**SPESIFIKASJONSBASERTE (SVART BOKS) TEKNIKKER:**

* 1. Ekvivalensklasseinndeling
  2. Grenseverdianalyse
  3. Beslutningstabelltesting
  4. Tilstandsbasert testing
  5. Brukstilfelle (use case) testing

**1.1 EKVIVALENSKLASSEINNDELING**

- Går ut på at inputen, som skal gå inn i systemet, i områder som forventes å ha lik oppførsel.

-Defineres fra både gyldige og ugyldig data, det vil si verdier som forventes godtatt hhv å bli forkastet

-Kan benyttes for mennesker i input, input via grensesnitt mot systemet, eller grensesnitt parametere under integrasjonstesting.

-For eksempel ved regulære uttrykk.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

**1.2 GRENSEVERDIANALYSE**

Vi burde ikke bare teste grenseklassene, men vi burde også teste grenseområdene i de klassene. Maksimum og minimum verdier av en klasse er den klassens grenseverdier.

**Et bilde som inneholder tekst

Automatisk generert beskrivelse**

**1.3 BESLUTNINGSTABELLTESTING**

En beslutnings tabell er en måte å beskrive komplekse forretningsregler. Handlingene er enten sanne eller usanne. Hver kolonne av tabellen tilsvarer en forretningsregel. Resultatet er en hendelse for hver regel. Minst en test per kolonne for å dekke alle muligheter.

**Et bilde som inneholder bord

Automatisk generert beskrivelse**

**1.4 TILSTANDSBASERT TESTING**

Systemet illustreres som et tilstandsdiagram. Det viser forholdet mellom tilstand, input og mange ugyldige overganger. Det lages tester ut ifra tilstand, overgang mellom tilstander, input, verdier eller hendelser. Et eksempel på dette er å modellere flyten i en skjerm dialog.

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

* 1. **BRUKSTILFELLE / BRUKERHISTORIE TESTING**

Brukstilfelle eller med andre ord et use case beskriver forhåndsbetingelser, avslutningstilstander, hovedflyt og alternativ flyt. Brukerhistorier beskriver bare hovedflyt. Brukstilfeller brukes mest ved akseptansetesting og brukerhistorier ved systemtest (agile).

**----------------------------------------- DEL 1 ----------------------------------------------**

**2. STRUKTURBASERTE (HVIT BOKS) TEKNIKKER**

-Komponentnivå: Er strukturen i koden selv for eksempel programinstruksjoner, beslutninger eller forgreininger eller til og med bestemte stier gjennom programmet.

-Integrasjonsnivå: Strukturen kan være kaldt et kallhierarki (Et diagram der moduler kaller opp andre moduler.).

-Systemnivå: Strukturen kan være en menystruktur, forretningsprosess eller nettsidestruktur.

**2.1 PROGRAMINSTRUKSJONSTESTING OG DEKNING (LINJEDEKNING)**

Måling av prosenten av kjørbare programinstruksjoner som har blitt utført av en enhetstest-testsuite. Code coverage.

**2.2 GRENDEKNINGS- /BESLUTNINGSTESTING OG DEKNING**

Måling av prosenten av beslutningsresultat, det vil si sant og usant mulighetene i en if setning som har blitt utført av en enhets-testsuite. Grense dekning er bedre enn programinstruksjonsdekning. 100% greningsdekning garanterer 100% programinstruksjonsdekning, men ikke motsatt.

Dekningsgrad kan også benyttes på andre testnivået som på integrasjons nivå. Altså prosenten av moduler komponenter eller klasser som har blitt utført.

**Et bilde som inneholder tekst

Automatisk generert beskrivelse**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**Et bilde som inneholder tekst

Automatisk generert beskrivelse**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**Et bilde som inneholder diagram

Automatisk generert beskrivelse**

**ERFARINGSBASERTE TEKNIKKER**

* Testene utledes basert på testernes kunnskap og deres erfaringer med liknende applikasjoner og teknologier
* Erfaringsbasert testing blir anvendt i senere tilfeller
  + Testere kan forutse feil basert på erfaring
* Eksplorativ testing
  + Ved få eller manglende spesifikasjoner
  + «ad hoc» testing, nå skal jeg prøve å kræsje systemet
  + Ved stort tidspress
  + For å supplere mer formell testing
    - Brukes faktisk også i formelle miljøer!

**TESTLEDELSE**

**1. Testorganisasjon**

- Bruk av uavhengige testere kan øke effektiviteten med å finne feil.

Det finnes ulike grader av uavhengighet:

1. Ingen uavhengige testere. Utviklere tester sin egen kode

2. Uavhengige testere innenfor utviklingsgrupper

3. Uavhengig testgruppe i organisasjonen

4. Uavhengige testere fra forretningssiden

5. Uavhengige testspesialister for spesifikke testmål (UU, sikkerhet etc,)

Fordeler med bruk av uavhengige testere kan være uavhengige testere ser andre og ulike feil. Og de kan verifisere antakelser som folk gjorde under spesifikasjonen og implementasjon av systemet.

Ulemper kan være isolasjon fra utviklingsgruppen. Uavhengige testere kan bli ansett som de forsinker leveransen. Utviklere kan miste ansvarsfølelsen for kvaliteten. Uavhengige testere må lære seg systemet og/eller fagfeltet.

Store prosjekter har som regel uavhengige testere.

|  |  |
| --- | --- |
| **TESTLEDERENS OPPGAVER** | **TESTERENS OPPGAVER** |
| Koordinere og utarbeide teststrategier og planer, planlegge testene (velge teststrategier, estimere tider, ressurser, planlegging), organisere verktøy, ta beslutninger, overvåke og skrive sluttrapporter. | Gjennomgå og bidra til testplaner, analysere, lage testspesifikasjoner, sette opp miljø, automatisere tester, gjennomgå tester som er utviklet av andre, forberede og fremskaffe test-data, implementere tester på alle nivå, utføre og logge disse, evaluere resultater og dokumentere avvik fra forventede resultater. |

**2. Testplanlegging og estimering**

Bestemme omfang, risiko og mål. Definere teststrategier, inklusive testnivåer, og start- og sluttkriterier. Koordinere testaktivitetene med livssyklusaktivitetene, definere tidspunkter og testanalyse og designaktiviteter, tildele ressurser, definere detaljnivået for testprosedyrer.

**START- OG SLUTTKRITERIER**

Når kan testingen starte:

-testmiljøet er klart og tilgjengelig

-testverktøy er klare i testmiljøene

-testbar kode er tilgjengelig

-testdata er tilgjengelig

Når skal testingen stoppe:

-oppfylt grundighetsmål, som kodedekning og funksjonsdekning

-når feiltetthet eller pålitelighetsmål er oppnådd

-ikke rettede feil eller mangel på testdekning er vurdert innenfor tilfredsstillende risiko

-tidsplaner eller kostnader tilsier avslutning

**TESTESTIMERING**

Dette er noe som kan være vanskelig og krever en del erfaring.

1. Basert på måldata: estimere testarbeidet på grunnlag av måledata fra tidligere eller liknende prosjekter eller basert på typiske verdier.

2. Ekspertbasert: estimere oppgavene ved at den som skal ha ansvaret for oppgavene eller eksperter estimerer

**TESTSTRATEGIER OG TESTMETODER**

Testmetoden er realiseringen av teststrategien. Testmetoden blir definert og detaljert i testplanene og testdesignet.

**METODISKE TESTMETODER:** spesifikasjonsbasert, erfaringsbasert, feilgjetting

**ANALYTISKE TESTMETODE:** risikobasert testing der testing blir rettet mot områdene med størst risiko

**DYNAMISKE TESTMETODE:** utforskende testing

**KONSULTATIVE TESTMETODER:** man spør eksperter ‘hvor mye skal vi teste her?’

**REGRESJONSMETODER:** omfattende automatisering av funksjonelle regresjonstester, og standard testsuiter

**PROSESS- ELLER STANDARDKONFORME METODER**: som de som er spesifisert i bransjestandarder.

**3. Overvåking og kontroll av testfremdrift**

Prosent arbeid utført i forberedelse av testtilfelle og testmiljø.

**Utføring av testtilfelle:** antall testtilfelle utført eller ikke, antall testtilfelle kjørt med rett eller feil

**Informasjon om feil:** feiltetthet, feil funnet og rettet, feilrate og resultatene av fornyet test.

**Testdekning**

**Testernes subjektive tillitt til produktet**

**Oppnåelse av testmilepæler**

**Testkostnader, inklusive kostnad sammenlikner med nytteverdien av å finne den neste feilen eller å kjøre den neste testen**

**TESTRAPPORTERING**

Oppsummere informasjon om testingen: hva skjedde under testing, info om dagene, bedømme feil som er igjen, økonomiske nytte, om testmålene til testnivået var passende, om tilnærmingen til testen var passende og om effektiviteten av testing mht. dens mål.

**TESTSTYRING**

Betyr at man skal endre på planer om det er nødvendig. Om det er store feil må man kanskje øke tester, kanskje testene er nede, og annet kan gjøres i mellomtiden? Omdefinere ressurser eller skaffe flere etter behov.

**4. Konfigurasjonsstyring**

Ha styr på alle test case, testdokumentasjon, feil, retting av feil og mer. Alle endringer og alle deler av dokumentasjon må være konfigurasjonsstyrt. Det vil se alt må være dokumentert 🡪sikre sporbarhet.

**5. Risiko og testing**

**Organisatoriske faktorer:** mangel på kunnskap, interne problemer i team, dårlig holdning eller feil forventning til testing.

**Tekniske faktorer:** feildefinering av krav, testomgivelsen er ikke ferdig på tide, kvaliteten av design, kode, testdata og testtilfeller.

**Leverandørfaktorer:** problemer hos en tredje part (underleverandør), problemer med kontrakter.

**Produktrisikoer:** for dårlig programvare levert til test, slik som funksjonalitet, pålitelighet, brukbarhet og ytelse, dårlig dataintegritet eller at ønskede funksjoner ikke oppfylles.

**RISIKOBASERT TILNÆRMING TIL TESTING**

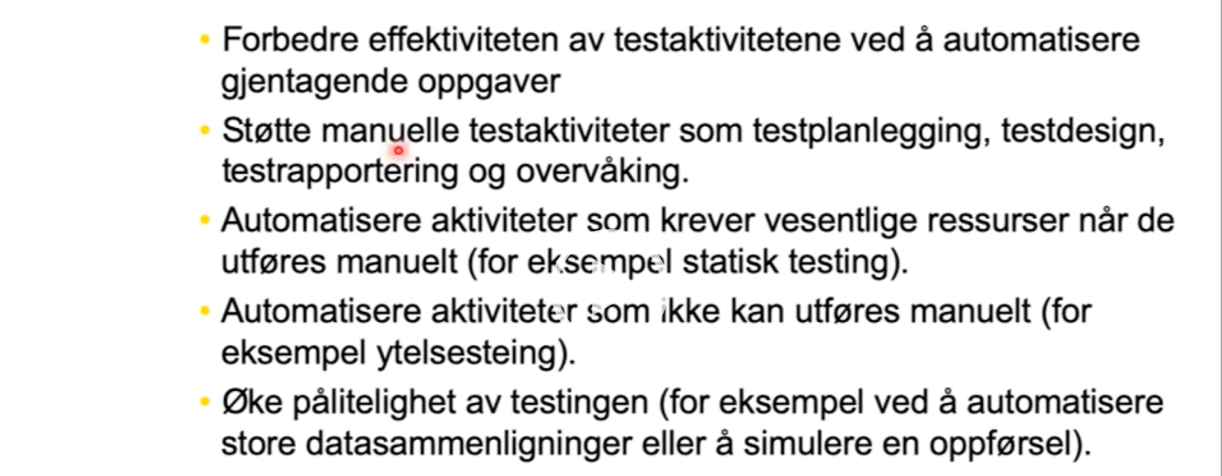
En litt annen måte å tilnærme seg testing på. Man vurderer risiko av prosjektet og bestemmer testteknikker og metoder. Bestemmer hvor mye test som skal utføres og hvilke deler som skal prioriteres. Definere aktiviteter utenom testingen for å redusere risikoen, for eks tilstrekkelig opplæring.

**6. Hendelses- og feilstyring**

En hendelse er avvik mellom virkelig og forventede utdata og indikerer feil. Det er viktig å definere og følge prosesser for hendelseshåndtering og regler for klassifisering av disse. Hendelsesrapportering: hvilke testobjekt går dette på, forventet resultat og dokumentering.

**VERKTØYSTØTTE**

**FORSTÅ MÅL OG MENING MED VERKTØYSTØTTE FOR TESTING**

****

**VERKTØYSTØTTE FOR ADMINISTRASJON AV TESTING OG TESTER**

1. Testadministrasjonsverktøy (f.eks. Excel, MTM)
   1. utføre tester, spore feil og rapportering av testobjektene
   2. spore testobjektene til kravspesifikasjon med evt. Versjonskontroll
2. Kravhåndteringsverktøy (f.eks. Excel, MTM)
   1. lagrer kravbeskrivelser, brukerhistorier etc. Prioriter disse
3. Feil-/Hendelseshåndteringsverktøy
   1. lagrer og håndterer avviks/hendelsesrapporter, dvs. defekter, feil, endringsønsker eller antatte problemer og anomalier/avvik
4. Konfigurasjonsstyringsverktøy
   1. nødvendig for å lagre og styre versjonering av testdokumenter og relatert programvare

**VERKTØYSTØTTE FOR TESTUTFØRING OG LOGGING**

Et bilde som inneholder tekst

Automatisk generert beskrivelse

**MULIGE FORDELER OG RISIKO VED VERKTØYSTØTTE TIL TESTING**

Et bilde som inneholder tekst, brev

Automatisk generert beskrivelse

**SPESIELLE HENSYN FOR NOEN TYPER VERKTØY**

1. Testutføringsverktøy (F.eks. Selenium)
   1. å lage tester ved å gjøre opptak av manuelle tester
   2. vanskelig å vedlikeholde, blanding av data og funksjonalitet
   3. kan lett bli ustabile når endringer skjer i produktet (GUI)
2. Statiske analyseverktøy
   1. sjekking av kodestandarder kan generere masse «feil» altså warnings. Man bør slå av alle warnings og slå de av på senere.
3. Testadministrasjonsverktøy
   1. rapporteringen er som oftest ikke på det formatet man ønsker
   2. Ref MTM