Pensumoversikt - boka:

1. Fundamentals of Testing

Hvorfor testing?:

- Unngå tap av penger
- Unngå tap av omdømmer
- Unngå skade og død

Årsaker til programvarefeil:

- Menneskelige
  - Interne - kunnskap og motivasjon, mm.
  - Eksterne - Stress, uhåndterbar kompleksitet
- Ukontrollerbare
  - Naturkatastrofer ol. Stråling, forurensing osv.

Feil = Mangle/bus

- Kan føre til svikt
- Fra milde til alvorlige konsekvenser

Testing i alle faser av livssyklus:

- Planlegging
- Utvikling
- Vedlikehold
- Drift

Testingens rolle:

- Reduserer risiko
- Kontroll - lovstandarder ol.
- Lære om systemet

Fordeler med testing:

- Måler kvalitet - loggføring av statistikk

- Skaper tillit til kvalitet
- Gir lærdommer som kan brukes videre
- Forebygger defekter og feil, samt konsekvensgrad

Hvor mye testing?

- Risikonivå - tekniske og businessrelaterte
- Prosjektbegrensninger - tid og budsjett

Hva er testing?

- Prosessen av å teste software og tilhørende prosesser for å forsikre som om at satte krav og funksjonalitetsmål oppfylles, samt for å oppdage eventuelle defekter.

Testaktiviteter (de som er utehevet er de "fundamentale" aktivitetene):

- **Planlegging** (Hva, hvordan, når, av hvem?)
- **Kontroll (og monitorering)** (kontroller og juster planlegging i tilfelle ny informasjon/utfordringer)
- **Analyse** (gjennomgå testgrunnlaget og omformulere testobjektiver til testforhold)
- **Design** (spesifisering av test cases, testmiljø, testdata, dekning og sporbarhet)
- **Implementasjon** (samle tester til scripts, prioriter, forbered orakler, skriv automatiserte tester)
- **Utførelse** (Kjør og sammenlign tester med orakler, rapporter hendelser, gjenta for alle korreksjoner, loggfør)
- Resultatsjekk
- Evaluering av exit kriterier (Sammenlign utførelse med de definerte målene. Trengs flere tester? Må exit kriterier endres?)
- Rapportering av testresultater (oppsummering for stakeholders)
- **Test avslutning** (tilgjengeligjør test assets for senere bruk)

Testing kan fokusere på forskjellige objektiver:

- Kravoppnåelse
- Fuzztesting - framprovosering av feil
- Kontroll av endringer
- Kvalitetskontroll (uten intensjon om å finne defekter - f.eks. ytelse)

7 prinsipper for testing:

1. **Testing viser at feil finnes, ikke fraværet av dem.** Reduserer sjangsen for uoppdagede feil, men kan aldri bevise det.
2. **Fullstendig testing er umulig.** Bruk risiko og prioritering for å fokusere testingen.
3. **Tidlig testing.** Testing bør starte så tidlig som mulig i utviklingssyklusen, med fokuserte mål.
4. **Defekt opphopning (clustering).** Et lite antall moduler inneholder mesteparten av defektene som blir oppdaget i løpet a pre-release testing.
5. **Sprøytemiddelsparadokset.** Hvis det samme settet av tester blir gjentatt, vil det ikke lenger finne nye feil.
6. **Testing er kontekstavhengig.** Kritisk software testes forskjellig fra en shopping nettside.
7. **Fravær-av-feil feilslutningen.** At man ikke finner feil betyr ikke at de ikke finnes, eller at systemet ikke kan ha andre problemer.

Gode egenskaper for en tester:

- Nysjerrighet
- Profesjonell pessismisme
- Øye for detaljer
- God kommunikasjonsevne
- Erfaring med feil-gjetting

Uavhengighet i testing

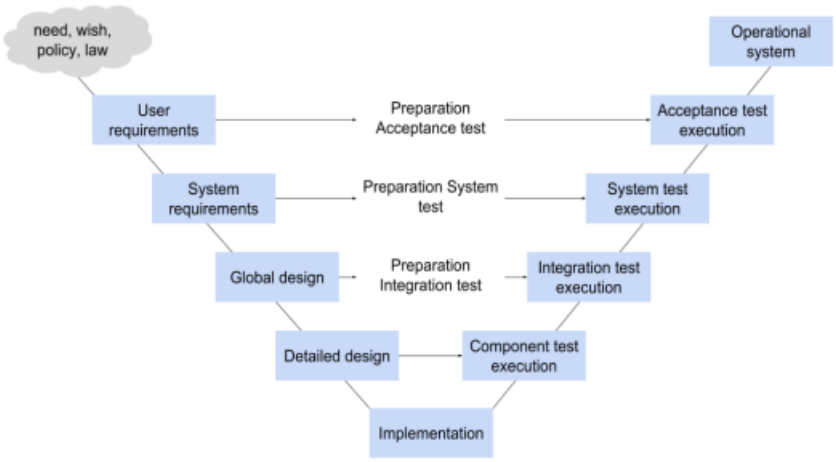
- Beskrivelse av hvor nært den som skriver testen er den som skriver koden
- Utvikler tester egen kode - lite uavhenig.
- Noen utenfor selskaper tester koden - veldig uavhengig.

2. Testing and Lifecycle

Testing fokuserer på:

- Bekreftelse - I henhold til spesifikasjoner?
- Validering - Riktig funksjonalitet?

V-modellen:



- Testing må begynne så tidlig som mulig
- Testing kan integreres i alle faser
- I V-modellen skjer validerings-testing spesielt i
  - de tidlige stadiene, f.eks. ved gjennomgang av bruker-krav
  - de sene stadiene, f.eks. under akseptansetestingen

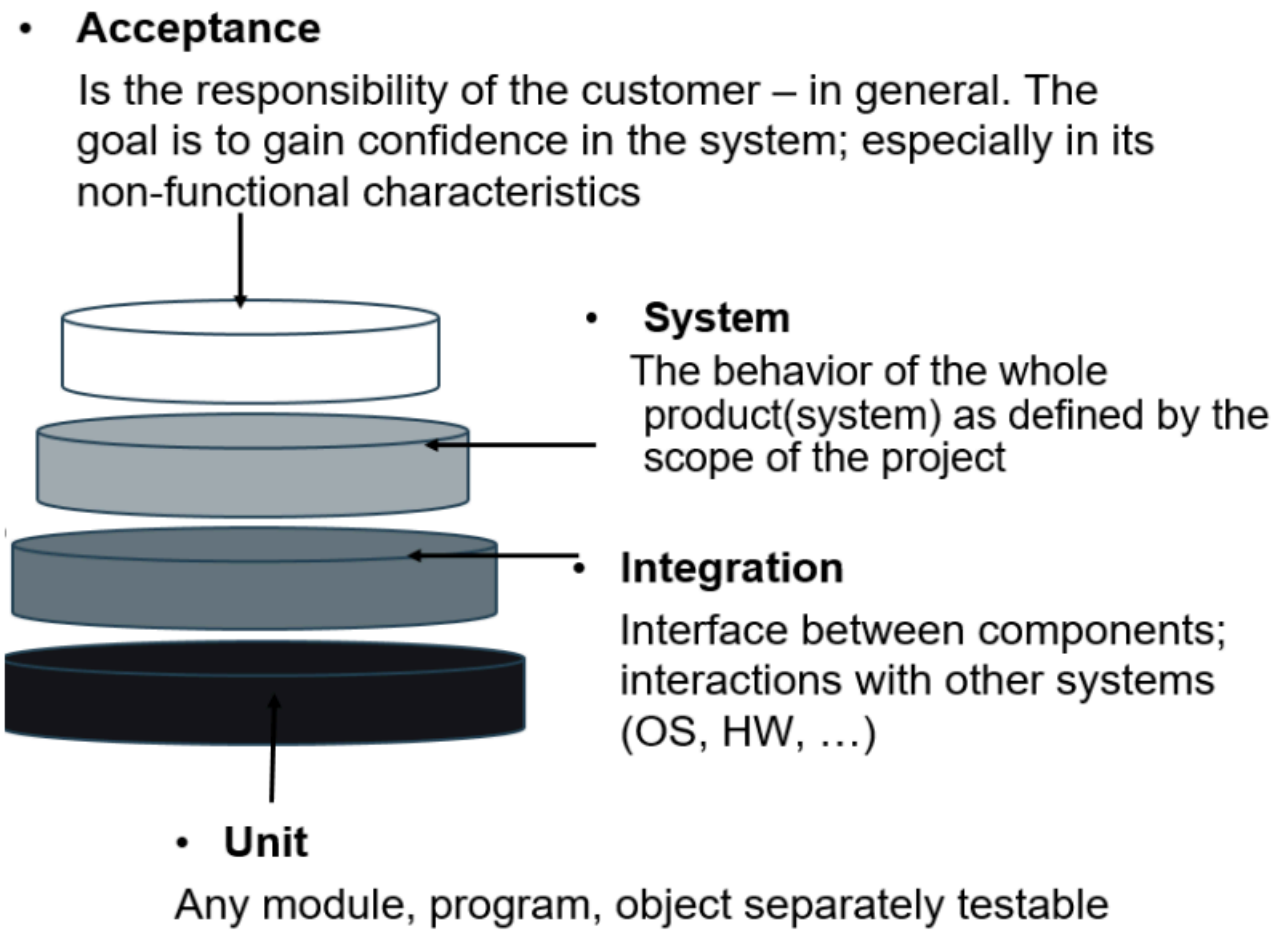
Software development models

- Iterativ-inkrementell utvikling - korte utviklingssykluser
  - Ethvert inkrement bør også testes
  - Regresjonstesting blir viktigere jo flere iterasjoner det har vært
  - Testing er ofte mindre formelt under denne modellen
  - Vanlig utfordringer:
    - Mer regresjonstesting
    - Defekter utenfor scopet til iterasjonen kan glemmes
    - Mindre grunding testing

Karakteristikker av god testing:

- Enhver utviklingsaktivitet har en samsvarende testaktivitet
- Hvert test nivå har testobjektiver spesifikke til det nivået
- Analysen og designet av tester for et gitt testnivå burde begynne under den korresponderende utviklingsaktiviteten.
- Testere burde involveres i gjennomgang av dokumenter så fort utkast av dem er klare.
  Testnivåer kan bli kombinert eller reorganisert avhengig av prosjektets natur eller systemarkitektur.

Testnivåer:



Komponent (Unit) testing:

- Objektiver - Bekrefte at komponenter som er *separat testbare* fungerer som de skal
- Komponenttesting tester:
  - Funksjonalitet
  - Ikke-funksjonelle karakteristikker
- Testbasis - Alt materiale som er anvendelig for komponenten under test
  - Spesifikasjonen
  - Softwaredesign
  - Datamodellen
  - Koden
- Skrive kode som tester kode
- Stubs, drivers og simulatorer kan brukes
  - Drivers er kode som erstatter en annen softwarekomponent for å kunne kalle på den under test
- Laveste grad av uavhengighet - som oftest laget av forfatter
- Defekter fikses umiddelbart, uten formell dokumentasjon
- TDD kan brukes

Integrasjonstesting:

- Tester interfaces mellom komponenter
- Interaksjoner mellom forskjellige deler av et system
- Testbasis
  - Software- og systemdesign
  - Systemarkitektur
  - Data flow
  - Workflows/use cases

- Typer:
  - Komponentintegrasjon - etter komponenttesting
  - Systemintegrasjon - etter systemtesting
- Lurt å fokusere på antatte problemområder først
- Bør skje inkrementelt, for å unngå sen defektoppdagelse
- Både funksjonelle og strukturelle fremgangsmåter kan brukes
- Noe grad av uavhengighet - kan gjøres av separat team, men gjøres ofte av utviklere

Systemtesting:

- Tester oppførselen til hele systemet
- Testbasis:
  - Systemkrav (funksjonelle/ikke-funksjonelle)
  - Businessprosesser
  - Risikoanalyser
  - Use cases eller andre høy-nivå beskrivelser av prosjektets systemoppførsel og interaksjoner med OS/systemressurser
- Testobjekter:
  - Hele det integrerte systemet
  - Brukermanualer/Operasjonsmanualer
  - Systemkonfigurasjoner og relatert informasjon/data
- Testmiljøet burde samsvare med produksjonsmiljøet
  - Kombinasjon av black-box og white-box
- Stor grad av uavhengighet helst - men kan gjøres internt avhengig av systemtype (sikkerhet, kritiske funksjoner, hemmelighetsstempling osv.)

Akseptansetesting:

- Spørsmål som krever svar:
  - Er systemet klart for publikasjon
  - Hva er gjenstående risikoer?
  - Har utviklingen nådd kravene?
- Målet er å etablere tillit til systemet selv, og dets ikke-funksjonelle karakteristikker.
- Testbasis:
  - Kravspesifikasjon (brukskrav og systemkrav)
  - Brukerhistorier & Use-cases
  - Businessprosesser
  - Risikoanalyse
- Typer:
  - Bruksakseptanse - validere at systemet er "fit for use"
  - Operasjonell testing
    - Testing av backup/restore
    - Krisegjennoppretting(disaster recovery)
    - Brukerhåndtering
    - Vedlikeholdsoppgaver
    - Periodisk sjekk av sikkerhetssårbarheter
  - Kontrakt og reguleringsakseptanse - dokumentfestede akseptansekriterier
  - Alpha- og betatesting:
    - Alphatesting skjer hos organisasjonen
    - Betatesting (field testing) blir utfør av personer på egne lokasjoner (f.eks. jobb eller hjemme)
    - Begge blir gjort av potensielle brukere, ikke utviklere

Testtyper

Funksjonell testing

- Objektiver: Teste systemets evne til å løse problemet det har som mål å løse, og softwarens eksterne oppførsel.
- Testnivåer: Alle
- Testbasis: Alle beskrivende dokumenter (kravspecc, businessprosesser, use cases) og eventuelle udokumenterte forventede oppførsler

Ikke-funksjonell testing:

- Objektiver: Måle karakteristikker av software som kan kvantifiseres
- Testnivåer: Alle
- Testbasis: ISO - Maintainability, Efficiency, Portability, Reliability, Functionality, Usability

Strukturell testing:

- Objektiver: Måle grundigheten av testing - dekningsgrader
- Testnivåer: Alle, men vanligst på komponent og integrasjon
- Testbasis: Kodestruktur og systemarkitektur
- Metode: Spesifikasjonsbaserte
- Verktøy: De som måler dekningsgrad eller lignende.

Endringsrelatert testing:

- Objektiver: Å verifisere at modifikasjoner til softwaren eller miljøet ikke har uintenderte side-effekter, og at systemet fortsatt møter kravene sine.
- Testnivåer: Alle. Knyttes også til funksjonell- ikke-funksjonell- og strukturell testing.
- Typer:
  - **Bekreftelsestesting:**
    - Etter en defekt er funnet og fikset burde softwaren re-testes for å bekrefte at defekten er suksessfullt fjernet.
  - **Regresjonstesting:**
    - Gjentatt testing av et allerede testet program, etter modifisering, for å oppdage introduserte defekter eller oppdage eksisterende som ikke var funnet.
- Metode: Til hvilken grad man gjør endringsrelatert testing er basert på risikoen for å finne defekter i software som fungerte tidligere. Regresjonstest-suiter blir kjørt mange ganger, og utvikler seg generelt sakte, så dette er en god kandidat for automatisering. Viktig å tenke skala, så testsuiten ikke blir for stor, evnt bare kjøre et subset av suiten.

Vedlikeholdstesting:

- Objektiver: Verifisere at modifikasjoner, migrasjon eller pensjonering av software eller systemer har gått etter plan.
- Testnivåer: Alle nivåer og alle typer, ved behov.
- Typer:
  - Modifikasjoner: Planlagte utvidelser. Korrigerende og krise endringer(patches). Miljøendringer (OS, database)
  - Migrasjoner: Operasjonell test av det nye miljøet. Testing på den endrede hardwaren.
  - Pensjonering: Testing av datamigrasjon eller arkivering, hvis lang data-retention kreves.
- Scope: Basert på endringens risiko og størrelse, samt størrelsen på det eksisterende systemet.

---

3. Statiske Metoder

Statisk testing vs Dynamisk testing

- Statisk: Eksaminering av kode uten å eksekvere/kjøre. Kan gjøres med andre produkter enn bare kode. Både manuell gjennomgang og automatiserte analyser.
- Dynamisk: Krever eksekvering av kildekode.
  Statisk og dynamisk testing har de samme objektivene: å finne defekter.
  Komplementerer hverandre
  Statisk testing finner årsaker til feil, dynamisk testing finner feil.

Statisk analyse med verktøy:

- Undersøker programkode, i tillegg til outputs som HTML, XML, osv.
- Objektiver: Finne defekter og øke kvaliteten på kildekoden og software modellene.
- Typiske defekter funnet av statiske analyseverktøy:
  - Referering av variabler med udefinerte verdier.
  - Inkonsistent interfacing mellom moduler og komponenter
  - Variabler som aldri blir brukt
  - Unåelig (død) kode
  - Syntaks feil i kode og software modeller
  - Programmeringsstandarder som brytes
  - Sikkerhetssårbarheter

Utviklere bruker statisk analyse før og under komponent- og integrasjonstesting.
Designere bruker statisk analyse under software modellering.

Verdien av statisk analyse:

- Tidlig oppdagelse av defekter, i forkant av test gjennomføring
- Tidlig advarsel om mistenkelige deler av kode eller design.
- Identifisering av defekter som ikke blir lett funnet av dynamisk testing.
- Oppdage avhengigheter og ukonsistent oppførsel i software modeller, f.eks. lenker.
- Oppgradert vedlikeholdbarhet av kode og design.
- Forebyggelse av defekter.

Målbare faktorer:

- Kompleksitet kan måles på forskjellige måter, f.eks. basert på antall valg i programmet (antallet binære valg.)
- Kodestruktur:
  - Kontrollflytstruktur: Sekvensen instruksjoner gjennomføres i.
  - Dataflytstruktur: Følger "sporene" til data, slik den blir aksessert og modifisert gjennom koden.
  - Datastruktur: Organiseringen av selve dataen, uavhengighetsgrad av programmet (array, list, stack, queue, tree, graph)
- Kodestandarder:
  - Navnekonvensjoner
  - Adgangskonvensjoner - public/private
  - Layout - indents f.eks.
  - Satte programmeringsregler - f.eks. alltid sjekk array boundaries ved bruk
- Code metrics:
  - Kommentarfrekvens
  - Nesting dybde - Antall logiske operasjoner uten konklusjon (nøstede for-løkker f.eks.)

- Kompleksitet
  Verktøy for statisk analyse kan gi en stor mengde advarsler, som må håndteres godt for effektiv bruk.

Eksempler på verktøy:

- SonarCloud

Reviews

Reviewprosessen

- På skala fra uformellt til formellt:
  - Uformelt review, Gjennomgang, Teknisk review, Inspeksjon.
  - Grad av formalitet beskriver i hvor stor grad det er strukturert og regulert i forkant. De mest uformelle vil ikke ha noen skrevne instruksjoner, mens en inspeksjon vil ha tydelige mål og retningslinjer.
- Formaliteten på en reviewprosess henger sammen med:
  - Risiko, størrelse på prosjektet, krav fra lover og reguleringer, behov for revisjonsspor
- Faser i et formelt review:
  1. Planlegging
     1. Velg persjonale
     2. Fordel roller
     3. Definer entry og exit kriterier hvis det er formelt.
     4. Velg hvilke deler av hva som skal gjennomgåes
  2. Initiering av review (Kickoff)
     1. Distribuer dokumenter
     2. Forklar objektiver/mål
     3. Forklar dokumentene til deltakerne
     4. Kontroller og diskuter entry/exit kriterier
  3. Individuellt review (med individuelle forberedelser)
     1. Dokumentasjon av potensielle defekter, uklarheter og kommentarer.
     2. Foreslå/anta alvorlighetsgrad for defekter - Kritisk, stor, liten
     - Denne fasen er forberedelse til review meeting
  4. Problem-formidling og analyse (review meetings)
     - Loggføring og diskusjon
     - Alt fra bevisstgjøring om defekter, til løsningsforslag og avgjørelser.
     - Avgjørelser baseres på exit kriterier.
     - Undersøkelse, evaluering og dokumentering(recording)
  5. Fiksing og rapportering (Rework og oppfølgning)
     - Fiksing av funnede defekter, som oftest av "forfatter"
     - Valider er defektene er tatt hånd om
     - Samle inn målbare verdier
       - Antall defekter funnet
       - Antall defekter per side
       - Tid brukt per side
       - Total innsats (review effort)
       - osv.
- Roller og ansvar - En person kan ha flere roller:
  - Forfatter: Hovedansvarlig for dokumentene som skal reviewes og reworks etter reviews.
  - Administrering: Bestemmer hvordan og hva som skal reviewes. Fordeler ressurser og avgjør hvorvidt review målene (objectives) er oppnådd.
  - Review leder: Har overordnet ansvar for reviewet og avgjør hvem som skal involveres. Jobber tett med administratorer og fasilitator

    - Fasilitator/moderator: Leder og planlegger reviewet av dokumentet, review møtene og eventuelle follow-ups. Ved behov den som "mekler" mellom motstridende synspunkter, og er often den som påvirker reviewets suksess i størst grad.
    - Reviewere: Personer med spesfikk teknisk eller business kompetanse. Identifiserer og beskriver funnene i produktet som reviewes. Bør være med på review meetings.
        - Viktig å her tilstrebe et utvalg av personer med forskjellige perspektiver og kompetanse.
    - Skribent(recorder): Dokumenterer alle problemer, feil og åpne punkter som blir identifisert gjennom møtene.

Hvorfor?:

- Billigere/mer effektivt å fjerne defekter som blir funnet tidlig i livssyklusen.
- Reviews kan finne mangler, for eksempel i krav, som er usannsynlige å finne gjennom dynamisk testing.
- Tidlig defekt detektering og korreksjon
- Økt produktivitet
- Færre defekter
- Bedre kommunikasjon

Den største manuelle aktiviteten i reviews er å undersøke ett arbeids-produkt, og kommentere på det. (viktig prinsipp at kostnaden av å finne og fikse defekter øker over tid)

Hva?:
Alle software arbeidsprodukter kan bli reviewed

- Kravspesifikasjoner
- Designspesifikasjoner
- Kode
- Testplaner, testspesifikasjoner, testcases og testscripts.
- Bruker guides
- Web pages

Typiske defekter som er lettere å finne med statiske metoder:

- Fravik fra standard
- Kravdefekter
- Designdefekter
- Manglende vedlikeholdbarhet
- Feil grensesnittsspesifikasjoner
- Uklarheter, motsigelser, mangler, overflødigheter osv.

1. Uformelt review

- Formål: Lite kostbar måte å skape litt verdi.
- Metode: Pair review, f.eks. parprogrammering eller en teknisk lead som gjennomgår design og kode.
- Ingen formell prosess, men kan dokumenters hvis man velger det.

2. Walkthrough

- Formål: Læring, skape forståelse, finne defekter, feedback.
- Metode: Møter leder av forfatter. Kan variere i praksis fra ganske uformelt til veldig formelt. Stakeholder kan delta.

3. Teknisk review

- Formål: Diskuter, ta avgjørelser, evaluer alternativer, finn defekter, løs tekniske problemer, undersøk samsvar med spesifikasjoner og standarder.
- Metode:

    - Varierer fra veldig formelt, til uformelt peer review uten administratordeltakelse.
    - Ideelt sett ledet av en trent fasilitator eller moderator
    - Dokumentert og definert prosess for å finne defekter. Inkluderer likemenn (peers) og tekniske eksperter.
    - Møteforberedelser
    - Valgfritt bruk av sjekklister, review rapport, liste av funn og administrator.

4. Inspeksjon

- Formål: Finne defekter
- Metode:
    - Vanligvis peer examination ledet av en trent fasilitator eller moderator (ikke forfatter!)
    - Formell prosess basert på regler og sjekklister, men entry og exit kriterier.
    - Møteforberedelser
    - Definerte roller
    - Inkluderer målinger(metrics)
    - Inspeksjonsrapport, liste av funn

Review teknikker

- Ad hoc reviewing
- Sjekklistebasert reviewing
- Scenario-basert reviewing og "dry runs"
- Rolle-basert reviewing
- Perspektivbasert reviewing

Suksessfaktorer for reviews:

- Organisasjonelle:
    - Ha et tydelig mål
    - Velg den riktige typen og teknikken
    - Review materialet må holdes oppdatert
    - Begrens scopet!
    - Sett av nok tid!
    - Støtte fra administrasjon er kritisk!
- Person-relaterte:
    - Velg de riktige personene
    - Bruk testere
    - Hver reviewer må gjøre arbeidet sitt gjennomført og ordentlig
    - Funn av defekter må være velkommene
    - Reviewmøter er vel-administrerte
    - Tillit er kritisk
    - Kommunikasjon er viktig - både hva og hvordan
    - Følg reglene
    - Opplæring!
    - Kontinuerlig fokus på prosesser og verktøy
- Metode-relaterte:
    - Funn av defekter er velkommne, og blir uttrykt og dokumentert objektivt
    - Implementer passende teknikker for typen og nivået av software produktet.
    - Bruk sjekklister og roller hvis passende, for å øke effektiviteten på defekt identifisering.
    - Administrasjon støtter en god review prosess, f.eks. ved å gi nok tid
- Opplæring og læring:
    - Opplæring blir gitt i review metoder, spesielt de mer formelle.

- Det legges fokus på læring og prosessforbedring!

---

4. Test Design Techniques

Før vi starter med testing:

- Hva er det vi prøver å teste?
- Hva er inputene?
- Hva er resultatene disse inputene ideelt bør produsere?
- Hvordan forberede vi testene?
- Hvordan kjører vi testene?

Svarene ligger i:

- Testforhold
- Testcases
- Testprosedyrer

Design har også forskjellige grader av formalitet, gjerne samsvarende med ønsket grad av testformalitet

Testutvikling

1. Testanalyse

- Analyse av testdokumentasjon - identifisere testforhold og bestemme hva som skal testes
- Testforhold (test condition): Et item eller event som kan verifiseres gjennom ett eller flere test cases
- Eksempler på testforhold:
  - En funksjon
  - En transaksjon
  - Et kvalitetsattributt
  - Andre strukturelle elementer (menyer på nettsider f.eks.)
- Testmuligheter:
- Først: Identfisier så mange testforhold som mulig
- Deretter: Velg hvilke å utvikle i mer detalj
- Kan ikke test alt, må velge et subsett
- Passende testdesignteknikker fører til riktige valg og god prioritering av testforhold

2. Testdesign

- Gjennom testdesign blir testcases og testdata laget og spesifisert.
- Et testcase er et sett av forhånds-tilstander (pre-conditions), inputs, forventede resultater og etterkants-tilstander(post-conditions), utviklet for å dekke spesifikke testforhold.
- Testorakel:
  - Informasjonskilde om korrekt oppførsel for systemet.
  - Forventede resultater - Endring av data og tilstander, outputs, osv.
  - Hvis man ikke definerer forventede resultater kan troverdige feilresultater bli feilaktig antatt som riktig. Disse bør derfor defineres i forkant av testgjennomføring.

3. Testimplementasjon

- Under testimplementasjon organiseres testcases i testprosedyrene (Developed -> Implemented -> Prioritized -> Organized)
- En manuell testprosedyre spesifiserer sekvensen av handling for gjennomføring av en test
- En automatisert testprosedyre kjøres med et test execution tool, og sekvensen av handlinger defineres i et test script.
- Testgjennomføringsplanen definerer:

    - Rekkefølge av gjennomføring av testprosedyrer, og potensielt automatiserte test scripts.
    - Når de skal gjennomføres.
    - Av hvem de skal gjennomføres.
- Testgjennomføringsplanen skal ta hensyn til faktorer som:
    - Risiko
    - Regresjonstesting
    - Prioritering
    - Tekniske og logiske avhengigheter
- Skrivingen av en testprosedyre er en mulighet til å prioritere testene, spesielt med tanke på tid.
- "Finn de skumle tingene først." Hva som er skummelt varierer med business, system, prosjekt og risikoene i prosjektet.

Kategorier av testdesignteknikker

- Dynamisk -> Spesifikasjonsbasert (Black-box):
    - Bruker modeller (formelle eller uformelle) for å spesifisere problemet som skal løses, i tillegg til softwaren eller dens komponenter.
    - Vi utleder systematisk testcaser fra disse modellene.
- Dynamisk -> Strukturbasert(white-box):
    - Testkasene utledes fra informasjon om hvordan softwaren er byg opp, f.eks. kode og design.
    - For eksisterende test cases kan vi måle testdekningen til softwaren.
    - Videre testcases kan utledes systematisk for å øke testdekningen.
- Dynamisk -> Erfaringsbasert
    - Testcasene utledes fra kunnskapen og erfaringen til personer.
    - Kunnskapen til testere, utviklere og andre stakeholders om softwaren, dens bruk og dens miljø.
    - Kunnskap om sannsynlige defekter og deres distribusjon (fordeling)

Dynamiske -> Spesifikasjonsbaserte

- **Equivalence partitioning:**
    - Den grunnleggende idéen er å dele opp et sett av testforhold til partisjoner (subsett), hvor elementene i hver partisjon kan regnes som ekvivalente (like, samme, tilsvarende)
    - Det er viktig at forskjellige partisjoner ikke deler elementer
    - Vi trenger bare å teste ett forhold fra hver partisjon, fordi alle elementene i samme partisjon vil behandles på samme måte av softwaren.
    - Given a set A, and the subsets A1 and A2 where
      $A1 \cup A2 = A$
      $A1 \cap A2 = \emptyset$ (det tomme settet)
      i.e. that A1 and A2 together constitute the whole of A, and that A1 and A2 are disjunct sets without common elements. We then say an A1 and A2 constitute one partition of A
    - Metode: Inputs/Outputs/Interne verdier i softwaren deles opp i grupper som forventes å dele oppførsel. (summer innenfor samme rentenivå i en bank f.eks.)
        - Deler opp i gyldig og ugyldig data, verdier som kan brukes og verdier som skal avvises.
        - Tester kan designes til å dekke mer enn en partisjon
    - Testnivåer: Alle
    - EP kan brukes for å oppnå Input og Output coverage
- **Boundary Value Analysis:**
    - Oppførsel i grensen av hver ekvivalens partisjon har høyere sannsynlighet for feil enn oppførsel innen partisjonene. Grenseverdianalyse ses derfor ofte på som en utvidelse av ekvivalenspartisjonering.
    - Grenser er områder hvor testing har økt sannsynlighet for å produsere defekter
    - Maksimum- og minimumsverdiene til en partisjon er grenseverdiene.

- Grenseverdiene til en gyldig partisjon er gyldige grenseverdier, grenseverdiene til en ugyldig partisjon er ugyldige grenseverdier. Tester kan designes for å dekke begge.
- Testnivåer: Alle
- Relativt lett å gjennomføre.
- **Tommelfingerregel:** Velg to grenseverdier (min og max), og én verdi fra midten av partisjonen for testing.

- **Decision table testing:**
  - Avgjørelses tabeller er en god måte å finne systemkrav som inneholder logiske forhold (conditions), dokumentere internt systemdesign og å bokføre (record) komplekse business regler som et system skal implementere. (business regler er hva'et, der business logikken i koden er hvordan)
  - Hva er avgjørelsestabeller?:
    - Ligner på sannhetstabeller fra tidligere fag
    - Årsak - effekt tabeller
    - Brukes når inputs og handlinger kan uttrykkes som Boolske verdier.
    - Systematisk måte å uttrykke komplekse business regler

Example

- Betingelser:
  GSK: Generell studiekompetanse
  R1: Bestått matematikk R1 fra videregående skole
  KP: Konkurransepoeng over årets grense

- Aksjoner:
  SP: Tilbud om studieplass
  V: Settes på venteliste
  FK: Tilbud om forberedende kurs i R1.
  A: Avslag, dvs. avslag på studieplass og forkurs, samt heller ikke på venteliste.

	Regel 1	Regel 2	Regel 3	Regel 4	Regel 5	Regel 6	Regel 7	Regel 8
GSK	true	true	true	true	false	false	false	false
R1	true	true	false	false	true	true	false	false
KP	true	false	true	false	true	false	true	false
Aksjon	**SP**	**V**	**FK**	**A**	**A**	**A**	**A**	**A**

    - Tar høyde for alle kombinasjoner av betingelser, og uttrykker hvilket utfall hver kombinasjon får gjennom aksjon.
  - Hvorfor?:
    - Hjelper testere å identifisere effektene av kombinasjoner av forskjellige input.
    - Viser ALLE kombinasjoner av input (hvis brukt riktig)
    - Effektiv metode for å finne feil i krav
    - Dokumenterer internt systemdesign

- **State transition testing:**
  - Et system kan være i et begrenset antall forskjellige tilstander. Denne egenskapen til et system kan beskrives som en "*finite state machine*", et tilstandsdiagram.
  - Etthvert system der du får et forskjellig output for et gitt input, avhengig av hva som har skjedd før, et er begrenset tilstandssystem
  - Overgangen fra en tilstand til en annen defineres av "maskinens" regler
  - Komponenter i state transition models:
    - Tilstander: softwarens tilstand. Open/Closed, Active/inactive, etc.
    - Overganger: Fra en tilstand til en annen (ikke alle er lovlige/mulige)
    - Hendelser(events): Forårsaker overganger. Lukke en fil, ta ut penger, etc.
    - Handlinger(actions): Resultater av overganger. F.eks. feilmelding
  - En begrenset tilstandsmaskin blir ofte vist som en tilstandsdiagram

    - Tilstandene til et system under test er separate, identifiserbare og begrensede i antall
    - Hvorfor?
        - Systemet kan gi variert respons basert på tilstander og historikk.
        - State transition testing gir testeren innsyn i softwarens tilstander, overganger, inputs og events som forårsaker tilstandsoverganger (transitions) og handlingene(actions) som er resultat av disse overgangene.
    - Testene kan designe for flere formål:
        - Dekke en typisk sekvens av states
        - For å eksekvere en spesifikk sekvens av overganger
        - For å dekke alle tilstander
        - For å eksekvere alle overganger
        - For å teste ugyldige overganger
    - Brukes mye i software industrien og i teknisk automatisering generellt
- **Use case testing:**
    - Beskriver interaksjoner mellom aktører (brukere og system) som produserer et resultat av verdi for en systembruker
    - Identifiser test cases som exercise(eksekverer?) hele systemet, transaksjon for transaksjon, fra start til slutt.
    - Beskriver interaksjoner mellom aktør og system for å opppnå en spesifikk handling (achieve a specific task) som produserer en verdi for bruker
    - Defineres gjennom aktøren, ikke systemet. Basert på faktisk bruk.
    - Kan identifisere integrasjonsdefekter
    - Pre conditions: conditions that need to be met for a use case to work successfully.
    - Post conditions: terminerer use-caset. Er observerbare resultater og slutt-tilstander for systemet, etter use-caset er fullført.
    - Mainstream: Mest sannsynlige scenario/programflyt.
    - Alternative branches: Alternativ flyt
    - Testnivåer: Mye brukt for akseptansetesting, da use-case testing er best på å avdekke prosessflyt gjennom ekte bruk av systemet.

Dynamiske -> Strukturbaserte testteknikker (White box):

- Strukturbaserte teknikker fyller to formål:
    - Måling av testdekning
    - Strukturell testcase design - kan generere flere test cases med mål om å øke dekning
- Dekning = (number of coverage items exercised / total number of coverage items) * 100 (vanlig prosentregning)
- Et "coverage item" er alt vi har fått til å få oversikt over hvorvidt en test har eksekvert eller brukt objektet.
- 100% coverage != 100% testing
- Strukturbasert testing baseres på en identifisert struktur i softwaren.
    - Komponentnivå
    - Integrasjonsnivå
    - Systemnivå
- Vi kan måle dekning av hver av de spesifikasjonsbaserte teknikkene:
    - EP: prosent av ekvivalens partisjoner exercised
    - BVA: prosent av partisjonsgrenser exercised
    - Avgjørelses tabeller: prosent av business regler eller tabell-kolonner testet.
    - Tilstandsoverganger: Prosent av tilstander besøkt, overganger (delt opp i gyldige og ugyldige) exercised
- Steg i dekningsmålings-prosessen:
    - Velg strukturellt element (metoder, linjer, logiske elementer (if/else, etc.)) å bruke, altså deknings items som skal telles
    - Tell de strukturelle elementene/items
    - Instrumenter koden - få koden til å "markere" elementer som blir testet

    - Kjør tester for koden som trenger dekningsmåling.
    - Gjennom output fra den instrumenterte koden kan man måle prosenten av elementer/items som exercises
- **Statement testing:**
    - I komponenttesting:
        - Statement coverage er prosenten av eksekverbare statements som har blitt exercised av en test case suite
    - Statement testing teknikken utleder test cases for å eksekvere spesifikke statements, normalt for å øke statementdekning. Målet er altså å kjøre så mye av koden som mulig.
    - Statement testing blir blant annet brukt for å finne "død kode"
- **Decision testing:**
    - Decision coverage er oversikten over decision outcomes. Hvor mange av utfallene fra valg i if, when og lignende sjekker blir testet.
    - Decision coverage er "sterkere" enn statement coverage - 100% decision coverage garanterer 100% statement coverage, men det samme gjelder ikke den andre veien.

Dynamiske -> Erfaringsbaserte teknikker:
Tester utledes fra testerens ferdigheter og intuisjon, og deres erfaring med lignende software og teknologier.
Veldig variable utfall, avhengig av testers erfaring og ekspertise.
Gjøres gjerne etter formelle metoder.

- **Error guessing:**
    - Testere antar defekter basert på erfaring
    - En strukturert fremgangsmåte for error guessing er å utlede en liste over mulige feil, og designe tester som angriper disse.
        - Dette kalles "fault attack"
- **Exploratory testing:**
    - Samtidig test-design, -gjennomføring, -logging og læring.
    - Baseres på et testskjema som inneholder testmål, og som utføres innenfor tidsbokser.
    - Er mest brukt:
        - Der det er få eller manglende spesifikasjoner
        - Under sterkt tidspress
        - For å komplementere mer formell testing
        - Som et ekstra tiltak, for å øke sannsynligheten for å finne alvorlige defekter.

Valg av testteknikker:

- Type system
- Tid og budsjett
- Utviklings livssyklusen
- Reguleringsstandarder
- Brukerkrav
- Kontrakt
- Testmål
- Risikonivå og type risiko
- Tilgjengelig dokumentasjon
- Use case modeller
- Testernes kunnskap
- Tidligere erfaring med defektene som finnes

---

5. Test Management and Risk

Test organisering og uavhengighet:

- Testing av software og utvikling av software er IKKE det samme
  - Forskjellige oppgaver involvert
  - Forskjellige mindset å utvikle og å teste
- Testing er en vurdering av kvalitet
  - vurderinger er ikke alltid positive
- Separer testerne fra utviklerne
  - Forbedret defekt lokalisering gjennom bruk av uavhengige testere
  - Unngår forfatter bias -> Objektive vurderinger
- Nivåer av uavhengighet:
  1. Ingen uavhengighet. Utviklere tester egen kode
  2. Testuviklere fra samme team lager tester
  3. Eget test team/gruppe innad i organisasjonen. Rapporterer til prosjekt administrasjonen eller høyere.
  4. Uavhengige testere from business organisasjonen eller brukergrupper (user community)
  5. Uavhegige testspesialister for spesifikke testmål, som f.eks. brukervennlighetstestere, sikkerhetstestere eller sertifiseringstestere (som også gir sertifiseringer)
- For store, komplekse eller sikkerhetskritiske prosjekter er det vanligivs best å ha flere nivåer med testing, med noen eller alle av nivåene gjort av uavhengige testere.
- Utviklingspersonale kan delta i testing, spesielt på lavere nivåer.
- - Fordeler:
  - Uavhengige testere ser andre og forskjellige defekter, og er upartiske
  - En uavhengig tester kan bekrefte antakelser gjort gjennom spesifisering og implementering av systemet.
- - Ulemper:
  - Isolasjon fra utviklingsteamet (hvis totalt uavhengig)
  - Uavhengige testere kan være et bottleneck, som det siste checkpointet
  - Utviklere kan miste ansvarsfølelsen for kvaliteten
- Testing må ikke gjøres av person med kun tester rolle - kan også gjøres av prosjekt manager, kvalitets manager, utviklere, domeneeksperter, businesseksperter, infrastruktur og IT personale. Osv. osv. Dette kan være både positivt og negativ.
- Roller:
  - Test leader = Test manager / koordinator.
    - Koordinerer strategi og plan med prosjektmanagere
    - Planlegging: Forstå testmål og risikoer. Velge testmetoder, estimere tid og kostnad, skaffe ressurser, definere testnivåer og sykluser, planlegge hendelseshåndtering
    - Initiere testspesifikasjon, forberedelser, implementasjon og eksekvering.
    - Monitorere test resultater og exit kriterier.
    - Tilpass planer i henhold til progresjon og eventuelle utfordringer.
    - Oppsett av adekvat konfigurasjons management av testware for sporbarhet
    - Håndtering og valg av test metrics - hva skal måles og hvordan?
    - Velg hva som skal automatiseres, til hvilken grad og hvordan.
    - Valg av verktøy og planlegging av opplæring i bruk av disse.
    - Velge testmiljø, og hvordan det skal implementeres.
    - Skrive test oppsummering rapporter
  - Testere:
    - Reviewer og bidrar til testplaner, analyser og design.
    - Analyse, review og vurdering av brukerkrav, spesifikasjoner og modeller for testbarhet.
    - Lage testspesifikasjoner

        - Oppsett av testmiljø (ofte i koordinasjon med administrasjon og management)
        - Forbered egen og hent inn ekstern testdata
        - Forbereder, designer, implementerer og eksekverer tester på alle nivå.
        - Loggføre testing, evaluere resultater og dokumenter avvik fra forventede resultater.
        - Bruk verktøy i henhold til plan
        - Automatiser tester i henhold til plan
        - Mål prestasjon av komponenter og system (hvis ønskelig/applicable)
        - Review tester skrevet av andre
- Ferdigheter krevd av de som er involvert i testingen:
    - Applikasjon eller business domene: En tester må forstå den intenderte oppførselen til systemet, og problemet systemet ønsker å løse, for å kunne finne uintendert oppførsel.
    - Teknologi: En tester må vite om problemer, begrensninger og muligheter i den valgte implementasjonen av teknologi, for å kunne lokalisere problemer og funksjoner og features med "sannsynlige feil."
    - Testing: En tester må ha kunnskap om testing, for å kunne følge valgt metodikk.
    - Grunnleggende profesjonelle og sosiale ferdigheter: Forberedelse og levering av både skrevne og muntlige rapporter, kommunisere effektivt, bidra til et positivt forhold mellom utviklere og testere.

Testplanlegging og estimering:

- ***Hvem*** skal gjøre ***hva, når*** og ***hvordan***
- Formål med en testplan:
    - Tvinge frem en konfrontasjon med utfordringene som står foran teamet, og fokusere tenkning på viktige temaer.
    - Planleggingsprosessen og planen selv fungerer som verktøy for kommunikasjon med andre medlemmer i prosjektteamet, testere, likemenn og andre stakeholdere.
    - Testplanen hjelper også med endringshåndtering - når vi får mer informasjon revideres planene.
- Planlegging kan dokumenteres i:
    - Et prosjekt eller "master test plan"
    - Separate testplaner for testnivåer
- Testplanlegging er en kontinuerlig aktivitet, som skjer i alle deler av prosjektlivssyklusen.
- Tilbakemeldinger fra testaktiviteter kan brukes for å anerkjenne endring i risiko, slik an planene kan justeres.
- Aktiviteter i testplanlegging:
    - Avgjør scope og risiko
    - Identifiser testmål
    - Definer testmetoder.
    - Definer testnivåer
    - Definer entry og exit kriterier
    - Integrer og koordiner testaktivitetene inn i software lifecycle aktivitetene.
    - Strategi: Gjør valg om hva som skal testes, hvilke roller skal gjennomføre testaktivitetene, hvordan testaktivitetene burde gjennomføres, og hvordan testresultatene skal evalueres.
    - Schedule: Lag timeplan for testanalyse- og testdesign-aktiviteter, og for implementering, eksekvering og evaluering.
    - Fordel ressurser til de definerte aktivitetene
    - Velg metrics for monitorering og kontroll av testforberedelser, eksekvering, defekt løsning og risikoproblemer.
- **Entry kriterier** - definerer når man skal starte testing.
    - Er testmiljøet satt opp og klart?
    - Er vektøyene klare i miljøet?
    - Er koden testbar og tilgjengelig?
    - Er testdata tilgjengelig?
- **Exit kriterier** - definerer når man skal slutte testing. F.eks. slutten på et testnivå, slutten av et prosjekt, eller måloppnåelse for testene.

- Grundighetsmålinger: Dekningsgrader, risiko
- Estimater: Defekttetthet, pålitelighetsmålinger
- Kostnad
- Gjenstående risiko: Defekter som ikke er fikset, mangel på dekning i visse områder
- Schedule: "Time to market"

Test-estimering:

- Testarbeidet er vanligvis et subprosjekt innenfor et større prosjekt
- Et testprosjekt kan brytes ned til faser:
  - Planlegging og kontroll
  - Analyse og design
  - Implementasjon og eksekvering
  - Evaluere exit kriterier og rapportering
  - Test closure
- Innenfor hver fase kan vi identifisere aktiviteter, og innen hver aktivitet kan vi identifisere tasks, og muligens subtasks.
- For å forsikre deg om presise estimater del opp arbeid i oppgaver som er korte i varighet - typ en til tredager.
- Større tasks har risiko for å inneholde skjulte komplekse og store "sub-tasks"
- To metoder i pensum:
  - Metrics-based approach: Estimerer testinnsatsen basert på metrics fra tidligere eller lignende prosjekter, eller på typiske verdier.
  - Expert-based approach: Estimerer oppgavene etter eieren av oppgaven, eller gjennom eksperter.
- En god løsning er å kombinere de to metodene:
  - Først lag en "work-breakdown" struktur, og en detaljert bottom-up estimate.
  - Deretter bruk modeller og "tommelfingerregler" for å undersøke og justere de estimerte bottom-up og top-down'ene, gjennom bruk av tidligere historikk.
  - Denne fremgangsmåten gir et estimat som er mer presist og mer "defensible"
  - Bottom-up = Dele opp prosjektet i de minste mulige komponenter/oppgaver, og summe disse oppover for å finne total effort.
  - Top-down = Regne ut helhetlig krevd effort gjennom historikk fra lignende prosjekter, industri benchmarks eller modeller, og deretter fordele de estimerte ressursene nedover i prosjektet.
- Testinnsatsen er avhengig av flere faktorer:
  - Produktfaktorer:
    - Kvalitet på spesifikasjonen(test basis)
    - Størrelse på produktet
    - Kompleksitet
    - Viktigheten av ikke-funksjonelle kvaliteter - brukervennlighet, ytelse, sikkerhet, osv.
  - Prosessfaktorer:
    - Utviklingsmodellen
    - Tilgjengeligheten av testverktøy
    - Ferdighetene til de involverte
    - Tid/tidspress
  - Utfallet av testingen:
    - Antall defekter
    - Mengden arbeid som må gjøres på nytt

Testmetode/tilnærming (test approach) og strategier:

- Testmetoden er implementasjonen av teststrategien for et spesifikt prosjekt.
- Siden testmetoden er spesifikk til et prosjekt bør den dokumenteres i testplanen.

- Tidskategorier:
  - Preventative metoder - tester designes så tidlig som mulig
  - Reaktive metoder - Test design kommer etter software eller system er produsert
- Testmetoder og strategier:
  - Analytiske metoder: f.eks. risikobasert testing - testing blir fokusert på områder med størst risiko, requirement-based testing.
  - Model-baserte metoder: f.eks. testing som bruker statistisk informasjon om feil-rater (som reliability growth modeller)
  - Metodiske metoder: f.eks. feil-basert (som error-guessing og faul-attacks), erfarings-basert, sjekkliste-basert og kvalitetskarakteristikk basert.
  - Prosess- eller standardkompatible metoder: f.eks. spesifisert av industrispesifikke standarder eller smidige metodikker.
  - Dynamiske og heuristiske metoder: f.eks. undersøkende (exploratory) testing
  - Konsulterende metoder: f.eks. få testdekning evaluert av en domeneekspert fra utenfor teamet.
  - Regresjons-uvillige metoder: f.eks. gjenbruk av eksisterende testmateriale, bred automatisering av funksjonelle regresjonstester.
- Valg av testmetoder og strategier er en viktig faktor for suksessen av innsatsen, og presisheten til planene og estimatene.
- Valg av test strategier bør ta hensyn til:
  - Risiko
  - Kunnskapsnivå og ferdigheter i teamet
  - Mål
  - Reguleringer
  - Produktet
  - Businessen

Test progress monitoring:

- Hensikten med testmonitorering er å gi feedback og synlighet rundt testaktivitet.
- Informasjonen som skal monitoreres kan samles manuelt eller automatisk, og kan brukes til å måle exit kritererier, som f.eks. dekningsgrad.
- Metrics kan også brukes til å vurdere fremgang sammenliknet med planer og budsjett.
- Test log template:
  - Test log identifier: ...
  - Description: Hva som blir testet, hvilket miljø det gjøres i
  - Aktivitets og hendelses entries: Eksekverings beskrivelse, prosedyrens resultat, miljøinformasjon, utilsiktede (anomalous) hendelser, hendelsesrapport identifikatorer)
- Vanlige metoder / hjelpemidler:
  - Test case summary spreadsheets
  - Grafer med informasjon om antall defekter over tid, dekningsgrad, osv.

Test reporting:

- Test rapportering gjøres for å summere informasjon om testinnsatsen, inkludert:
  - Hva som skjedde gjennom en testperiode
  - Analyserte metrics for å støtte valg om fremtidige handlinger
- Metrics samles inn på slutten av et testnivå for å vurdere:
  - Om testmålene for nivået er oppnådd (test objectives adequacy)
  - Om testmetodene var gode for å nå målene (test approaches adequacy)
  - Effektiviteten til testene i henhold til målene
- Kan f.eks. rapporteres i skjemaer med oversikter over faktorer som uløste defekter, antall planlagte tester mot faktisk antall, osv.

- Test summary report template:
  - Test summary report identifier: ...
  - Summary
  - Variances
  - Comprehensive assessment
  - Summary of results
  - Evaluation
  - Summary of activities
  - Approvals

Test control:

- Prosjekter går ikke alltid i henhold til planen. Forskjellige faktorer kan lede til avvik:
  - Nye risikoer
  - Nye behov
  - Testfunn
  - Eksterne hendelser
  - Problemer med testmiljøer
- Testkontroll beskriver ethvert veiledende (guiding) eller korrigerende tiltak gjort som et resultat av innhentet og rapportert informasjon og metrics.
- Eksempler på testkontroll tiltak:
  - Ta valg basert på informasjon fra testmonitorering
  - Re-prioritere tester når en identifisert risiko inntreffer
  - Endre test timeplanen (schedule) grunnet tilgjengeligheten til et testmiljø

Configuration Management:

- Hensikten med configuration management(konfigurasjonsstyring) er å etablere og vedlikeholde integriteten til softwaren og relaterte produkter gjennom produktets livssyklus.
- Konfigurasjonsstyringen skal sørge for at all deler av testwaren er identifiserte, versjonskontrollerte og sporer endring. Dette for å opprettеholde sporbarhet gjennom test prosessen.
- Konfigurasjonsstyring hjelper med å unikt identifisere (og reprodusere):
  - Testobjekter -> Testdokumenter -> Testene -> Testpakken (drivere, mocks, testdata. Alt som trengs rundt for å kjøre testen)
- Konfigurasjonsstyrings prosedyrer og verktøy burde velges i planleggingsstadiet av prosjektet.

Risiko og testing:

- Risiko er muligheten for et negativt eller uønsket utfall, problemene som kan sette stakeholderne sine mål i "fare."
- Risiko hører til produktet og prosjektet
- Risikoanalyse og risikohåndtering (risk management) kan hjelpe oss med å ta gode valg for testingsprosessen.
  - Risiko brukes til å velge hvor vi starter testing, og hvor vi trenger mer testing.
- Risikonivå defineres av to faktorer:
  - Sannsynligheten for at en negativ hendelse skal skje
  - Hvor stor skade hendelsen vil påføre
- For enhver risiko har man fire valg:
  - Mitigate (dempe) - forebygge
  - Contigency (beredskap) - Plan B, vite hva man skal gjøre hvis noe skjer
  - Transfer (flytte) - Forsikring, skytjenester som eier risiko
  - Ignorere - Akseptere mulig tap, vanlig ved veldig liten risiko
- Prosjektrisiko - risikoene som påvirker prosjektets evne til å levere sine mål:

    - Organisasjonelle faktorer - Ferdighets- og personalmangel, manglende opplæring, utfordringer i personalet, kommunikasjonsproblemer, dårlig innstilling rundt testing.
    - Tekniske faktorer - Problemer med å definere de riktige kravene. Graden krav/mål kan oppnåes gitt eksisterende begrensninger. Kvalitet på design, kode og tester.
    - Supplier(leverandør) problemer: Tredjeparti feil, kontraktsproblemer
- Produktrisiko - en risiko direkte relatert til test objektet, potensielle feilområder i softwaren. Muligheten for at systemet eller softwaren ikke vil tilfredstille rimelige kunde-, bruker- eller stakeholderforventninger.
    - Leveranse av failure-prone(ofte feilende) software.
    - Produktet påfører skade/tap til brukere/selskaper.
    - Svake softwarekarakteristikker (funksjonalitet, pålitelighet, osv.)
    - Softwaren utfører ikke funksjonen den var tiltenkt
- Gjennom testing ønsker man å redusere risiko for at negative hendelser skal skje, og redusere den negative effekten av de som skjer allikevel.

Risikoanalyse:

- Identifiserer risikoobjekter (risk items)
- Bestemmer sannsynlighet og skadeomfang(impact) for hvert objekt
- Skala 1-10
- Prioriterer risikoobjektene i henhold til skala
- Risikoanalyse er informert gjetting!(educated guesses)
- Viktig å oppdatere/følge opp analysen ved viktige milestones i prosjekter
    - F.eks. ved alle V-modellens stadieoverganger
    - Nye risikoer kan dukke opp, prioriteringer kan endres
- Risikobasert testing krever også måling av funn av- og løsing av defekter
- Målet i risiko-basert testing burde ikke være - og kan ikke være - et risikofritt prosjekt.
- Beste praksis i risikohåndtering er å oppnå et prosjektutfall som balanserer risiko med kvalitet, features, budsjett og tid.

Defect Management:

- Defekt - Forskjellen på faktisk og forventet testutfall
- Defekthåndtering (defect management) - Prosessen av å anerkjenne(recognizing), undersøke, handle og fjerne (dispose of) hendelser.
- Defektrapport - Et rapportdokument som rapporterer på enhver hendelse som har skjedd og som krever undersøkelse.
    - Gir utviklere og andre feedback om et problem for å muliggjøre identifisering, isolering og korreksjon etter behov.
    - Gir testledere en måte å spore kvaliteten til et system under test og testprogresjon.
- Innhold i defektrapport:
    - Beskrivelse - situasjon, oppførsel eller hendelse
    - Et par skjermer/skjermbilder med informasjon samlet inn av defektsporingsverktøy
    - Beskrivelse av steg tatt for å reprodusere og isolere hendelsen
    - Skadeomfang
    - Klassifiseringsinformasjon - Scope, alvorlighetsgrad og prioritet bl.a.
    - Et prioritetsnivå, definert av testledelsen(test managers)
    - Risiko, kost, muligheter og fordeler knyttet til å fikse eller ikke fikse defekten.
    - Root cause - identifisert (captured) av en utvikler. Når ble defekten introdusert, og når ble den fjernet?
    - Konklusjon og anbefalinger

Ytelsestester:

- Krever et spesielt miljø, som ligner på produksjon, for å måle responstid og ressursbruk.

- Krever spesielle verktøy for måling

---

6. Tool Support for Testing

Potensielle gevinster og risikoer:

- Risikoer:
  - Å kjøpe eller leie et verktøy garanterer ikke suksess med det/verdi fra det!
  - Kan kreve mye innsats for å skape reell verdi!
  - Urealistiske forventninger - tid, kost, innsats
  - Over-reliance (avhengighet)
- Gevinster:
  - Høyere grad av forutsigbarhet og gjentakbarhet (repeatability)
    - Samme handling vil gjøres like hver gang - mennesker kan ikke garantere det samme
  - Objektive tolkninger - maskiner har ikke meninger
  - Data legges frem i lett forståelige format - grafer, skjemaer, visuelle representasjoner osv.
  - Mindre repetitivt arbeid.

Begrensninger:

- Verktøy gjør jobben sin svært godt, men bredden er ofte begrenset - de gjør det de er designet for.
- Testere må derfor fokusere på:
  - Hva som skal testet
  - Hva skal test casene være?
  - Hvordan prioritere?
- Da kan verktøy-brukerne (automatiseringspersonalet f.eks.) fokusere på:
  - Hvordan få verktøyet til å gjøre jobben sin effektivt
  - Hvordan skape økt verdi fra verktøybruken

Typer verktøy:

- Bruksområder (litt mer info om subtypene i slides):
  - Direkte bruk i testing - f.eks. test-eksekveringsverktøy, testdatagenererings-verktøy, resultatsammenlignings-verktøy.
  - Test management hjelpere
    - Apllikasjons-livssyklus management tool (ALM)
    - Verktøy for formalisering av testresultater, testkrav, hendelser og defekter.
    - Verktøy for monitorering og rapportering av testeksekvering.
    - Verktøy for konfigurasjonshåndtering.
    - Verktøy for kontinuerlig integrasjon.
  - Verktøy for exploration - Monitorerer fil-aktiviteten for en applikasjon mens man kjører den
- Klassifisering - Verktøy klassifiseres i henhold til testingsaktivitetene de støtter. Hvis de støtter mer enn en aktivitet, klassifiseres de under "hovedaktiviteteten."
- Noen verktøy er "intrusive" - verktøyet kan påvirke utfallet av teksten. F.eks. hvis forskjellige verktøy skal måle tid, men gjøre det på forskjellige metrics/med forskjellige verktøy)
  - Disse konsekvensene kalles "the probe effect"
  - Noen verktøy er mer passende for utviklere enn for testere

Formål med verktøy:

- Forbedrer effektivitet i testaktiviteter
- Automatisering av tester som er ressurskrevende hvis de gjøres manuelt (f.eks. statisk testing)
- Automatisering av tester som ikke kan gjøres manuelt (f.eks. storskala ytelsestesting)
- Økt pålitelighet i testing

Verktøy for statisk testing:

- Hjelper med å forbedre koden/arbeidsproduktet, uten å eksekvere det.
- Kategorier:
  - Review tools - mest brukt for dokumentering av mer formelle reviews
  - Statisk-analyse verktøy - støtter kodeeksaminering (mest brukt av utviklere - komponenttesting)
    - En utvidelse av compiler teknologi
    - Kildekoden er "input" til programmet
    - Brukes for å forstå og forbedre kode
    - Finne defekter før dynamisk testing
  - Modelleringsverktøy - validerer modellers av system/software
    - Mest brukt i design
    - Identifiserer inconsistencies og defekter
    - Identifiserer risikoområder
- Veldig kosteffektive! Finne ting tidlig er billigere!

Verktøy for testdesign og spesifikasjon:

- Kategorier:
  - Testdesign-verktøy
    - Generere testinput-verdier
    - Generere forventede resultater (hvis et orakel er tilgjengelig i verktøyet)
  - Modellbaserte testverktøy
    - Generer inputs og/eller test cases from lagret informasjon som beskriver en modell av systemet
  - Testdata forberedelses verktøy
    - Lager og forbereder data for bruk i testing
  - TDD - Test Driven Development verktøy (for utviklere)
    - ATDD - Acceptance test-driven development
    - BDD - Behavior-driven development
    - "Naturlig språksyntaks" - Given <"some condition">, When <"something is done">, Then <"result should happen">

Verktøy for testeksekvering og logging:

- Kategorier:
  - Testeksekveringsverktøy - Muliggjør automatisk kjøring av tester basert på lagrede inputs og forventede resultater.
    - Nivåer av skripting: Linear scripts, Structured scripts, Shared scripts, Data-driven scripts, Keyword-driven scripts
  - Coverage tools (dekningsverktøy) - Instrumenterer koden (identifiserer dekningsobjekter), kalkulere testdekningsgrad, osv.
  - Testharnesses and unit test framework tools (utviklere) - Harness = stubs og drivers, små programmer som interagerer med softwaren. Unit test framework tools = støtte for objektorientert programvare
    - Tester individuelle komponenter
    - Supplye inputs til tester og motta outputs fra tester
    - Eksekvere sett av tester innenfor rammeverket
    - Bokføre pass/fails for tester(framework tools)
    - Lagre tester (framework tools)
    - Support for debugging (framework tools)
    - Dekningsgrad på kodenivå (framework tools)
  - Test comparators (testsammenlikning)
    - Brukes når en eksekvert test genererer store outputs

        - Dynamisk sammenlikning -> Gjøres under testeksekvering
            - Gjøres best av verktøy
            - Kan endre kjøremønster midt i eksekvering hvis feil produseres
        - Post-execution sammenlikning -> Gjøres etter testen er ferdig

Verktøy for ytelse og monitorering:

- Datatyper for ytelsestesting:
    - Ekte data - fra brukere
    - Load - Store mengder produsert data
    - Maintenance (vedlikehold) - Testdata fra produksjonsmiljø
- Viktig at tester reflekterer realistiske scenarier.
- Oppsett av data kan kreve markant innsats
    - Testdata forberedelses-verktøy kan hjelpe her!
        - Kan hente data fra eksisterende database-sett
        - Kan lages, genereres og endres for bruk i test
        - Kan gi volum!
        - Nyttig for ytelses- og pålitelighet
- Monitoreringsverktøy:
    - Identifiserer og informerer om problemer
    - Logger real-time og historikk
    - Finner optimale settings
    - Vanlige funn:
        - Bruksstatistikk
        - Finne minnelekasjer
        - Funn av feil-logikk som null-peking

Verktøy for bruk i spesifikke apllikasjonsområder:

- Typer:
    - Datakvalitetsvurdering
    - Data konversjon (conversion) og migrering
    - Brukbarhet og Accesibility testing
    - Lokaliseringstesting
    - Sikkerhets testing
    - Portabilitets testing
- Eksempler:
    - Ytelsestestingsverktøy spesifikt for web-applikasjoner
    - Dynamiske analyseverktøy spesifikt for sikkerhet

Ekstra hensyn - Test eksekveringsverktøy:

- Disse krever ofte betydelig høy innsats for å skape betydelig verdi.
- Capture & replay skalerer dårlig
- Data-dreven skripting separerer inputs fra testen, og kan dermed gjøre samme test med forskjellig data.
- Keyword-dreven skripting bruk spreadsheets med nøkkelord som definerer handlinger (action words) sammen med test data. Testere kan endre mønster ved å velge keywords

Ekstra hensyn - Ytelsestestingsverktøy:

- Designet av byrden generert av verktøyet
- Timing aspekter - probe effekten
- Hvordan tolke dataen

- Krever ekspertise - både til design og tolkning

Ekstra hensyn - Statiske analyseverktøy:

- Ved bruk for å teste gammel kode mot nye standarder kan det oppstå uintenderte konsekvenser.
- Kan gi overveldende feedback
  - Kan derfor være lurt å implementere i inkrementer - enten på små deler av koden, eller med filtere.

Ekstra hensyn - Test management tools:

- Må interface med andre verktøy eller spreadsheets for å operere optimalt.

Introduksjon av et verktøy i en organisasjon:

- Vurder organisasjonens styrker, svakheter og "maturity" (modenhet).
- Evaluer mot tydelige krav og objektive kriterier.
- Bruk proof-of-concept
- Evaluer selgeren/vendor (inkludert opplæring, støtte og komersielle aspekter)
- Identifiser interne krav for opplæring og oppfølgning i bruk av verktøyet.
- Pilotprosjekt! - Lære, evaluere, velge standarder for bruk og vedlikehold, vurder kost-verdi.
- Suksessfaktorer:
  - Roll out inkrementelt
  - Tilpass og oppdater prosesser tilknyttet verktøyet
  - Tilby trening og oppfølgning
  - Definer guidelines for bruk
  - Monitorer bruk og nytteverdi

Testautomatisering

Hva er testautomatisering?:

- Testautomatisering refererer til bruken og utviklingen av spesialisert software, separert fra softwaren som skal testes, for å automatisk eksekvere tester på software applikasjoner, i motsetning til å gjøre det manuelt.

Hvorfor automatisere?:

- Økt effektivitet - automatiserte tester kjøres fortere enn manuelle tester, og kan derfor kjøres oftere også. Dette gir raskere feedback om softwarens kvalitet.
- Økt presisjon: Automatisering minsker risikoen for menneskelig feil som kan oppstå i manuell testing.
- Gjenbrukbarhet: Når de er lagd kan automatiserte tester bli gjenbrukt over forskjelllige versjoner av applikasjonen og i forskjellige miljøer (f.eks. browsere)
- Kontinuerlig testing: Automatisering fasiliterer kontinuerlig integrasjons og leveranse praksiser gjennom å muliggjøre frekvent testing med minimal innsats.
- Dokumentasjon: Automatisering tillater dokumentasjon av krav og generering av test resultat rapporter.

Hvorfor ikke automatisere?:

- Krever en markant oppstartsinvestering
- Ekstra kostnader
- Ekstra kunnskap nødvendig
- Ekstra vedlikehold
- Kan distrahere fra testmålene
- Kan gi falske alarmer

Begrensninger av automatisering:

- Ikke alle tester kan automatiseres
- Ikke alle tester *burde* automatiseres
- Kan ikke erstatte utforskende testing
- Ikke realistisk å automatisere systemer som:
  - Endres ofte og drastisk
  - Snart skal pensjoneres
  - Små i størrelse
- Hvis implementasjon av en automatisert test tar for lang tid, kan det være man ikke spare noe på den, og at det er bedre å teste det spesifikke tilfellet manuelt!

Testautomatisering - metoder:

- Tre hovedmetoder:
  - Capture & playback
    - Verktøy "tar opp" handlinger og interaksjoner fra en bruker
    - Lagrer "opptaket" som en alenestående test-session
    - Verktøyet "playbacker" (spiller av) handlingene og interaksjonene fra opptaket. Pixel for pixel, avhengig av skjermresolusjon.
      - Lett å sette opp og bruke
      - Mangler abstraksjon
      - Øker vedlikehold
  - Structured scripting
    - Starter med manuelle testprosedyrer
    - Transformerer disse til automatiserte test scripts

        - Har et skript bibliotek (funksjoner, klasser, etc.)
        - Har abstraksjon
        - Øker tilpasningsevne og vedlikeholdbarhet
        - Krever startinvestering for å lage skriptbiblioteket
        - Krever programmeringsskils
    - Model-based scripting
        - Automatisk generering av test cases & test scripts, basert på systemet modell
        - Genererer tester for forskjellige systemer
        - Teknologi uavhengig
        - Krever modelleringsekspertise
        - Modellbaserte test verktøy er enda ikke mainstream

Utfordringer for testautomatisering innen smidig utviklings life-cycle:

- Raske endringer i krav
- Opprettеholdelse av progresjon:
    - Ressursallokering
    - Oppstartsinvestering
- Balansere kundeverdi mot adekvat testdekning
- Tester blir mer komplekse når features vokser
- Samarbeid mellom team

Utfordringer med test data:

- Hva er syntetisk test data?
    - Data som blir kunstig skapt og etterlikner real-world data
    - Syntetisk test data blir skapt for å brukes i testing i non-production miljøer.
- Hvor kan vi hente testdata fra?
    - Lage det selv
    - Hente det fra andre team
    - Hente det fra utvendige systemer (Tenor testdata f.eks.)

Hvordan jobbe med automatisering innen smidig:

- Kontinuerlig prosess
- Opprettehold kvalitet og fart gjennom frekvente releases
- Bevissthet rundt hva som bør automatiseres og ikke kan fasilitere en god, smidig prosess
- Minimer oppstarts ressursene, og bygg i iterasjoner

Hvem burde jobbe med testautomatisering?:

- Team setup vil variere fra prosjekt til prosjekt
- Design, implementer og vedlikehold er teknisk av natur
- Krever domeneekspertise og testing skills
- Alle burde delta! Samarbeid mellom utviklere og "test ingeniører"

Hvordan velge et rammeverk?:

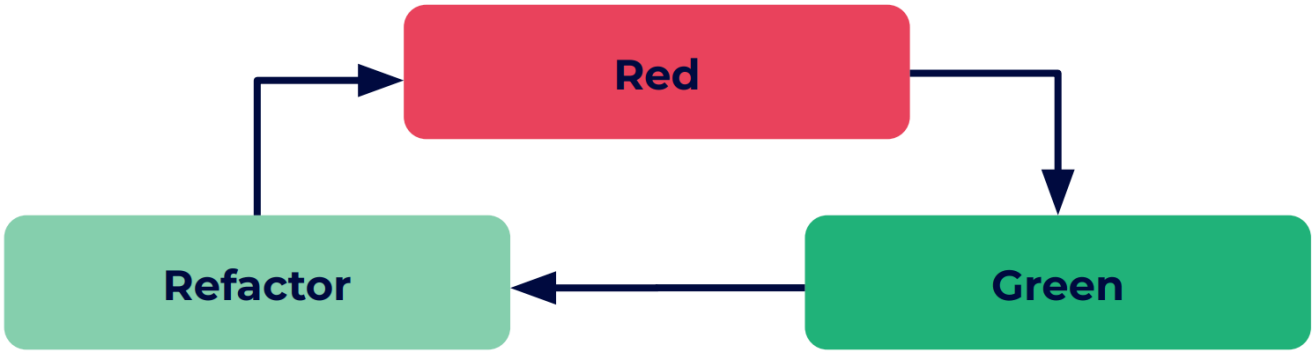
- Ingen "sølvkule" - ingen løsning er perfekt
- Betalt eller Open Source?
- Har rammeverket fortsatt support?
- Er community'et aktivt?
- Vurder å bruke et testautomatiserings rammeverk som ble skapt for ditt valgte utviklings rammeverk

- Hvilke rammeverk har teamet ditt erfaring med?

Playwright

- Pensumprogrammet for testautomatisering
- Javascript/Typescript basert
- Front-end testing verktøy for Web og API testing
- Gratis og Open Source
- Features:
  - Cross browser support
  - Screenshots og video
  - Browser contexts
  - Støtter flere språk
  - Parallel eksekvering
  - Stubs (mocks) og nettverksmanipulering
  - Kontinuerlig integrasjon
- Capture & Playback
- Integrert i VScode

TDD - Test Driven Development



- **Refaktoreringsstøtte:**
  - Tilrettelegger for endringer
- **Designverifisering:**
  - Koden gjør det du tror den gjør
- **Design fordelder:**
  - Høy kohesjon, lav kobling
- **Jobb i inkrementer:**
  - Del endringer ofte, flyt

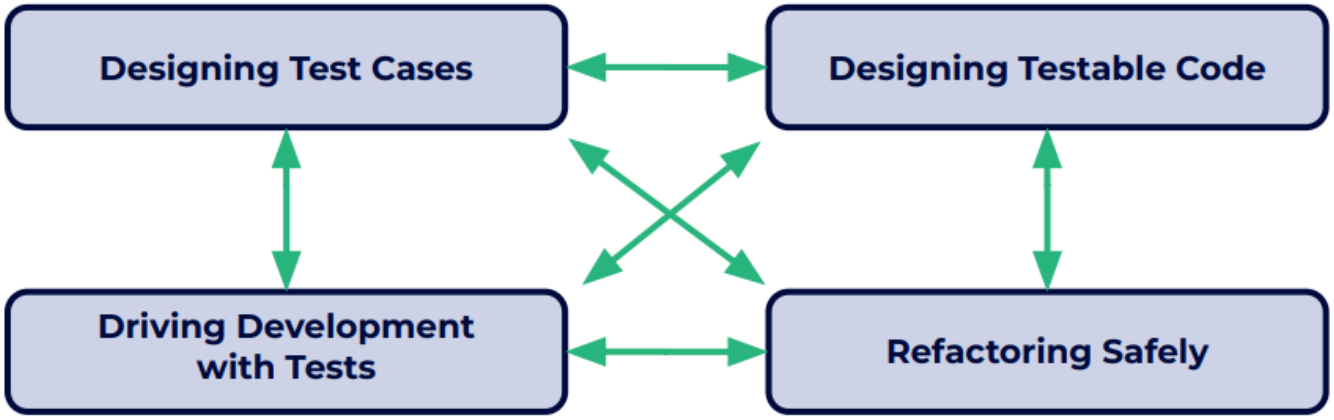
Dojo - Sted for å lære

- Dojo prinsipper:
  - Du kan ikke diskutere teknikker uten kode
  - Du kan ikke vise koder uten tester
  - **Kode uten tester eksisterer ikke!**
- *Kata* - sekvens med “moves” som du lærer

To praksiser

- Incidental
  - Repeter noe du allerede har kunnskap om
    - Gjør og bli bedre på det
- Deliberate
  - Prøv noe du ikke har kunnskap om
    - Bryt ned en ferdighet i komponenter som du øver på separat
    - Krever trygghet og motivasjon

TDD SKILLS



Unit test design - *Readability er key*

Selvtestende kode

- *”Du har selvtestende kode når du kan **kjøre en serie automatiserte tester** mot kodebasen, og være **sikker** at testene dersom **består** er koden din **fri for større defekter**”*
- Fordeler med unit tester:
  - Tester gir tillit til kode.
  - Tester git feedback om design.
  - Readability er viktig
    - Optimaliser for personen som trenger å forstår feilen
  - Test navn burde reflektere feilen.
    - Bruk CamelCase eller snake_case.
- **Tommelfingerregel: én assert per test**
- Unngå duplisering
- Tester gir støtte for rekatorering
  - Test via API, black-box style
- Test private metoder
  - Test via en public method som kaller på den private metoden.
  - Endre til private/protected
  - Flytt til en annen klasse der en er public
  - Bruk verktøy som modifiserer en kjørte koden (fysiske enheter som påvirker kode)
- Unittester burde være **raske**
  - Kjører på hvert build
  - Er ikke en unit test hvis den:
    - Rører filsystem
    - Rører nettverk
    - Rører databasen
    - Krever custom hardware
- Robusthet:
  - Isolering:
    - Feil skal ikke ha en ringvirkning
    - Rekkefølgen for tester er irelevant
    - Repeter tester
  - Feil oppstår ikke tilfeldig
    - Ingen Threading conditions: Feil skal ikke oppstår pga samtidighetsproblemer (f.eks to tråer om 1 ressurs)
    - Ingen nettverkshastighet problemer: Tester skal ikke feile pga nettverk.
- Begrensninger:
  - Unit tester finner ikke integrasjonsfeil
  - Testing finne aldri alle feil

Exploratory Testing

Tester for å sørge for at noe **fungerer**, fungerer som **forventet**, og at det fungerer **godt**.

Prinsipper for testing

- Testing viser **tilstedeværelse av defekter, ikke fravær** (altså: at man ikke finner feil betyr ikke at det ikke er feil der!)
- **Fullstendig (uttømmende) testing er umulig**
- **Tidlig testing** sparer tid og penger
- **Feil hoper seg opp** (defects cluster together, snowball-effekt)
- **Sprøyteparadokset** (pesticide paradox): De samme testene vil etter hvert ikke finne nye feil, man må variere
- **Kontekst er viktig** (context matters)
- **Fravær av feil er en feilslutning** (absence of errors fallacy): At det ikke er funnet feil, betyr ikke at programmet er korrekt eller tilfredsstiller behov

Mindsets :

- **Profesjonell optimisme**:
  - "Hvordan kan jeg bygge det?" (Utviklerperspektiv)
- **Profesjonell pessimisme**:
  - "Hvordan kan jeg ødelegge det?" (Testerperspektiv)

Kontekst er viktig - Testing er kontekstbasert

- **Programvaresystemer er ikke like**
  - Laget for ulike brukere
  - Brukt på ulike måter
  - Ulike typer risiko
- **Faktorer som påvirker testarbeidet/testinnsatsen**
  - **Type teknologi** (gammelt, nytt, hva slags teknologi)
  - **Type prosjekt** (stort, lite, smidig/agilt, fossefall)
  - **Type produkt** (eksperimentelt, livskritisk)
- Dette gjør at:
  - Tester må ha **forskjellig fokus**
  - Kan **ikke teste alle systemer likt**
  - Trenger informasjon om systemet
  - Test må **skreddersys til konteksten**

Hva er explorative testing?

- **Samtidig læring, testdesign og testutførelse**
- Fokuserer på oppdagelse, og styres av testeren selv
- Oppdager feil som kan være vanskelige å finne med tradisjonelle metoder
- Basert på erfaring, intuisjon og domene-kunnskap

Minneteknikker (mnemonics) og heuristikker

- **Mnemonic**: En minneteknikk for å huske informasjon
- **Heuristikk**: En praktisk, intuitiv tilnærming til problemløsning, ikke en rigid oppskrift

COUNT

En metode for å sikre at du tester alle relevante kvantiteter og mengder – spesielt for søk, lister og databehandling.

Verdi	Forklaring	Eksempel/testcase
zero	Søk etter noe som ikke eksisterer	Søk etter en bruker som ikke finnes
one	Søk etter én spesifikk ting	Søk etter én bestemt bruker
many	Søk som gir flere resultater	Søk etter brukere med felles etternavn
too many	Søk som gir *flere* resultater enn forventet	Søk som returnerer over 1000 brukere
too few	Søk som gir *færre* resultater enn forventet	Søk med et veldig snevert filter

Bruk: Sikre at systemet håndterer alle mengder – fra ingen, via én, til mange (og ekstremer).

GOLDILOCKS

Test ulike "størrelser" eller verdier for å finne grenseverdier og typiske input-feil.

Tilstand	Forklaring	Eksempel/testcase
too big	Overbelaste et felt/parameter	Lim inn 1000 tegn i et navn-felt
too small	La feltet stå tomt/for lite	Ikke fyll inn e-post-feltet, eller bruk 1 tegn
just right	Riktig input	Fyll ut alle felt riktig og passende

Bruk: Finn ut hvordan systemet reagerer på for mye, for lite og akkurat passe input.

CRUD

Sikrer at du tester alle hovedhandlingene mot data (Create, Read, Update, Delete).

Handling	Forklaring	Eksempel/testcase
Create	Lag ny entitet/bruke	Opprett ny bruker / opprett bruker med samme e-post
Read	Les/hent data	Hent bruker / forsøk å hente ikke-eksisterende bruker
Update	Oppdater eksisterende/noe som ikke finnes	Endre e-post for en bruker / prøv å endre ikke-eksisterende bruker
Delete	Slett eksisterende/noe som ikke finnes	Slett en bruker / prøv å slette ikke-eksisterende bruker

Bruk: Sikre at alle dataoperasjoner håndteres korrekt – også feiltilfeller!

RCRCRC

Gir et fokus for *hvor* du bør teste grundigst, basert på hvor feil sannsynligvis oppstår.

Fokusområde	Forklaring	Eksempel/testcase
recent	Nye funksjoner/kode (kan inneholde ferske feil)	Test nylig lansert modul
core	Kjernefunksjonalitet (må alltid virke)	Test innlogging, betaling, lagring

Fokusområde	Forklaring	Eksempel/testcase
risky	Områder avhengig av andre systemer/komponenter	Test integrasjon mot eksternt API
configuration	Områder påvirket av oppsett/miljø	Test funksjon etter endret konfigurasjon
repaired	Kode som nylig er rettet (kan introdusere nye feil)	Test funksjonalitet etter bugfix
chronic	Områder som ofte har hatt feil	Test gammel rapporteringsmodul som alltid feiler

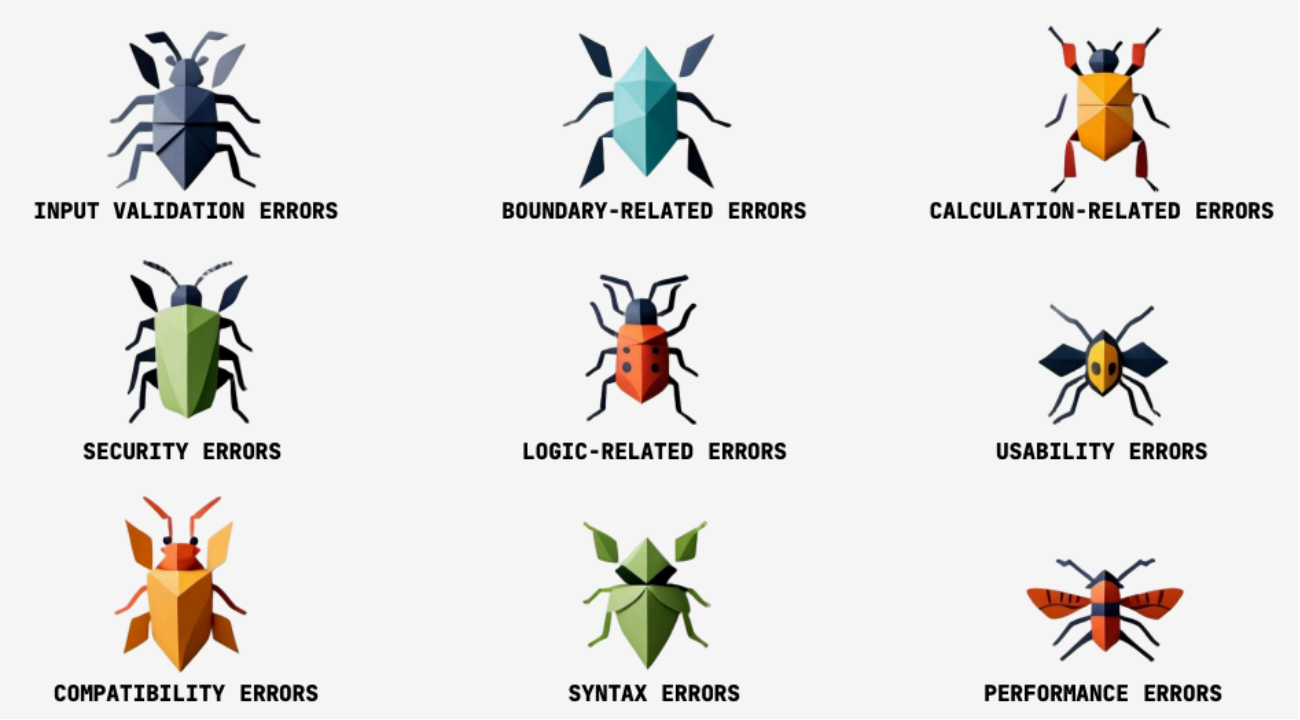
Bruk: Prioriter testing i områder hvor sannsynligheten for feil er størst, og hvor feil har alvorlig konsekvens.

Triggere og følelser

- Hendelser eller følelser kan trigge feil eller problemer (f.eks. frustrasjon over dårlig brukergrensesnitt)

Hvordan utføre explorative testing? (Steg-for-steg)

1. **Klassifiser feil** (fra tidligere prosjekter, rotårsaksanalyse, identifisere typiske risiki)



2. **Forstå systemet** (hva gjør det, hvem bruker det, hvordan fungerer det)
3. **Velg heuristikk** (bruk det som passer best, lag egne over tid)
4. **Lag en test charter** (plan for økten, hva skal testes, hvorfor, hvordan)
5. **Vurder funnene kontinuerlig** (lær av det du finner, tilpass videre testing)

Fordeler og ulemper med explorative testing

Fordeler:

- Avdekker ulike feil på grunn av den “tilfeldige” karakteren
- Rask feedback, krever lite forberedelse
- Oppdager nye testscenarier underveis
- Egner seg når det ikke finnes spesifikasjoner

Ulemper:

- Kvaliteten avhenger av testeren sin erfaring og kreativitet
- Vanskelig å reprodusere feil

- Kan gi dårlig dokumentasjon (“ad hoc”)
- Krever kreativitet

Utvikle en tester-mindset

- **Spill** kan bidra til å utvikle kreativitet, problemløsning, og tilpasningsevne paralleller til test-arbeid

---

Usability Testing

Absence of error fallacy → Finne og fikse feil hjelper ikke hvis software systemet ikke møter brukernes behov og forventninger.

Flere brukere	Flere situasjoner	Flere devices
- Ulike preferanser - Alder - Kultur - Funksjonelle evne - Digital competanse	- Tid - Sted - Vær - Følelser	- PC - Mobil - Osv…

HCI - Human Computer Interaction

- *I hvilken grad et produkt kan brukes av spesifiserte brukere for å oppnå spesifiserte mål med effektivitet og brukskvalitet i en spesifisert brukskontekst.*
- HCI består av:
  - Usability - Brukskvalitet
  - User friendliness - Brukervennlighet
  - Interaksjonsdesign (IxD)
  - User experience (UX)
- Formålet med HCI er å gjøre et software system:
  - Forståelig
  - Lett å lære
  - Effektivt å bruke
  - Lett å huske
  - Tilfredsstillende å bruke
- “Screenshot 2025-06-01 at 16.50.11.png” could not be found.
- Systemer burde utvikles med fokus på mangfold, og kan i verste fall medføre store bøter:
  - E.g SAS 150.000 kr dagen for dårlig system.

HCI Rammeverket: Øverst: 1 → 4: Nederst

1. Estetikk
   - Ansvarlig for førsteinntryket
   - Moderne, fresht, appellerende design
   - Identifisering av selskapets applikasjoner
   - Et selskaps grafiske profil
2. Grensesnitt dynamikk
   - Responsivitet
   - Tilpasningsdyktighet til brukernes behov og kontekst
   - Gir brukeren makt
   - Fangende
3. Brukskvalitet
   - Effektivitet
   - Tilfredshet
4. Grensesnitt standarder
   - Beste praksiser
   - Konsekvent adferd og design
   - Senker arbeidsbelastning

- Raskere development

Brukersentrisk designprosess

"Screenshot 2025-06-01 at 17.13.16.png" could not be found.

Hovedprinsipper

- Forstå hvordan brukere tenker og oppfører seg
- Samle fakta og data
- Studer, design og test med brukere før implementering
- Iterer!

De fire hovedstegene i prosessen

1. **Forstå brukerens kontekst**
   - Intervjuer
   - Spørreundersøkelser
   - Brukerobservasjon
   - Analyse og ekspertevaluering
   - Domenekunnskap og statistikk
2. **Spesifisere krav/behov**
   - Personas (fiktive brukerprofiler)
   - Brukerscenarier og caser
3. **Utvikle konsept**
   - Lavoppløselig prototyping (enkel papir-/skisseprototyp)
   - Høyoppløselig prototyping (mer likt virkelig produkt)
4. **Evaluere konseptet**
   - Brukertesting
   - Ekspertevaluering
   - Sammenlignende tester (A/B-testing, etc.)

Low-fidelity prototyping

- Enkelt, grovt, ofte på papir eller med enkle digitale verktøy
- Raskt og billig å endre
- Gjør det lett å teste mange ideer tidlig uten å bruke mye tid og penger på detaljert design
- Fokuserer på funksjon og flyt, ikke detaljer

Eksempel:
Tegn en app på papir og test med brukere ved å flytte rundt på "skjermbilder" og la dem peke.

Personas

- Fiktive, men realistiske brukerprofiler basert på data
- Hjelper utviklere og designere til å *forstå* og *huske* ulike brukeres behov, mål og utfordringer
- Gjør det lettere å designe for virkelige brukssituasjoner

Eksempel:
"Anna, 32, småbarnsmor, handler ofte på mobilen, ønsker en rask og enkel checkout."

Brukerobservasjon

Hvorfor?

- For å forstå hvordan brukerne faktisk jobber og hvilke utfordringer de har
- Avdekker behov og problemer som brukerne ofte ikke kan uttrykke selv

Spørsmål å kartlegge:

- Hvem er brukerne?
- I hvilken situasjon bruker de produktet?
- Hvilke mål skal produktet hjelpe dem å nå?
- Hvilke krav må oppfylles?

Fremgangsmåte:

1. **Kontakt brukerne** (e-post, telefon, forklar formål og roller)
2. **Forbered bakgrunnsspørsmål:**
   - Hvilke daglige oppgaver har du?
   - Hvilke oppgaver gjør du i *programmet*?
   - Hvor ofte og hvor lenge har du brukt det?
   - Har du fått opplæring?
3. **Lag observasjonsguide/fokusområder:**
   - Logg inn
   - Finn informasjon
   - Legg til/fjern elementer
   - Gjør arbeidsflyt

Under observasjonen:

- Besøk brukernes arbeidsplass
- Bruk “think-aloud” (be brukeren si hva han/hun tenker og gjør)
- Noter og/eller ta videoopptak (med samtykke)

Observasjonsverktøy:

- Silverback (Mac)
- Techsmith Morae (Windows)
- Funksjoner: opptak av skjerm, lyd og bruker, måling av tid, klikk, etc.

Tips:

- Start å finne brukere tidlig
- Vær realistisk på hvor mange du rekker å observere
- Gi brukeren nok tid – ikke avbryt eller hjelp!
- Du tester **ikke** brukeren, men programmet – vær hyggelig!
- Spør etter brukerens generelle inntrykk til slutt

Evaluering av designkonsept (brukertesting)

- Test prototypen med faktiske brukere
- Observer hva de forstår, får til, og hvor de stopper opp
- Samle inn både kvantitative og kvalitative data
- Juster designet basert på funnene

“Screenshot 2025-06-01 at 17.39.10.png” could not be found.

HCI-retningslinjer (Human-Computer Interaction Guidelines)

Hva er HCI-guidelines?

- Samlede standarder og beste praksiser for brukergrensesnitt, fra operativsystemer (f.eks. Windows, macOS) og web (f.eks. W3C).
- Gir utviklere og testere felles spilleregler for design og funksjonalitet.

Fordeler:

- **Konsistent oppførsel og design** på tvers av programvare
- **Reduserer brukerens arbeidsbelastning**
- **Raskere utviklingssyklus** – mindre tid på å finne opp hjulet på nytt

Grunnleggende brukervennlighetselementer å teste mot guidelines

Test alltid disse elementene mot gitte spesifikasjoner og retningslinjer:

- **Arbeidsflyt (Workflows):** Er stegene logiske og naturlige?
- **Navigasjon:** Er det lett å finne frem?
- **Søk og filtrering:** Lett å finne og avgrense informasjon?
- **Grid-systemer:** Ryddig, gir oversikt?
- **Sideflyt:** Viktig info starter øverst til venstre, slutter nederst til høyre (i vestlige kulturer)
- **Plassering av viktige knapper:** F.eks. "Lagre" og "Avbryt" på standard steder (Windows/macOS-standard)
- **Tab-rekkefølge:** Følger leseretningen, kolonner før rader, logisk innenfor grupper
- **Justering (Alignment):** Er elementer pent plassert?
- **Gruppering:** God bruk av grupper/bokser – unngå for mye!
- **Aktive/inaktive elementer:** Inaktive knapper skal alltid være synlig deaktivert
- **Destruktive handlinger:** F.eks. ikke plassér "Lagre" og "Slett" ved siden av hverandre; alltid be om bekreftelse
- **Plassering av avkrysningsbokser og radioknapper:** Følg OS-standard

Systemmeldinger (feil, advarsler, info, spørsmål)

- **Skal forklare problemet og gi løsning på brukerspråk**
- **Aldri bruk feilkoder, sjargong eller tekniske termer**
- **Aldri bare store bokstaver eller utropstegn – det føles som skriking**
- **Riktig meldingstype:**
  - **Error:** Noe har skjedd, brukeren må gjøre noe
  - **Warning:** Potensielt problem
  - **Information:** Bare til informasjon
  - **Question:** Brukeren må ta et valg
- **Kortfattet!** Folk leser ikke lange meldinger
- **Riktig handling på knapper:**
  - Feil og advarsler: ikke bruk "OK", men f.eks. "Lukk"
  - Unngå å ha "Ja", "Nei" og "Avbryt" sammen (forvirrende)

Accessibility Testing

Definisjoner:

Disability: Utfallet av interaksjonen mellom en person og miljø-, og holdningsbarrierer en person kan møte.
Usability: Effektiviteten og tilfredsheten en spesifikk gruppe brukere kan oppnå et spesifikt sett med oppgaver i et gitt miljø.
Accessibility: Brukskvaliteten av et produkt, tjeneste miljø, eller fasilitet for alle mennesker uavhengig av evne. (Graden et produkt, tjeneste eller miljø er tilgjengelig for så mange mennesker som mulig.)

Barrierer:

Barriere prioritet	Omfang
Kritisk	Barrierer som stopper noen fra å bruke en tjeneste/produkt, eller noen tjenester
Seriøst	Problemer som skaper frustrasjon, senker effektiviten eller krever work-arounds
Irriterende (moderat)	Ting som er frustrerende, men stopper ikke noen fra å bruke tjenesten
Støy	Mindre problemer som kan skape et problem. Kan dog skade troverdighet eller renomé

Tilgjengelighet glemmes ofte i HCI arbeid

- Usynlig, gjemt, misforstått
- **Tilgjengelighet (Accessibility) og Lovgivning**
- **Hensikt:**
  Sikre lik tilgang til samfunns-, politisk og økonomisk liv for alle, inkludert personer med nedsatt funksjonsevne.
  - Handler om tilgang til både fysiske steder **og** digitale verktøy, tjenester, informasjon og opplevelser.

Internasjonale avtaler:

- **FN:**
  - *Artikkel 9 i FNs konvensjon om rettighetene til mennesker med nedsatt funksjonsevne* (CRPD)
  - Forplikter alle medlemsland (192 land) til å sikre full tilgjengelighet i samfunnet, inkludert digitale tjenester.

EU:

- **Europeisk strategi for funksjonshemmede (2010–2020):**
  - Fokus på å bekjempe fattigdom, spesielt blant funksjonshemmede
  - Felles “disability card” for lik behandling i arbeid, bolig og reise i hele EU
  - Standarder for tilgjengelighet i valg-lokaler og kampanjemateriell
  - Ta hensyn til rettighetene til personer med funksjonsnedsettelser i utviklingsarbeid og for nye medlemsland

Personas - og tilgjengelighet

Hvorfor bruke personas?

- **Gir et realistisk bilde** av de menneskene vi designer for.
- Hjelper teamet å **ta hensyn til ulike brukertyper** (personas forteller en historie vi kan forstå).
- **Organiserer og dokumenterer** store mengder data og våre antakelser om brukerne.
- **Bygger en felles forståelse** i teamet om hvilke tilgjengelighetsbehov som må løses.

Personas – hva inneholder de?

- **Beskrivelse** (navn, alder, livssituasjon, hverdag)
- **Kjennetegn/egenskaper** (fysiske, kognitive, sosiale)

- **Ferdigheter** (aptitude)
- **Holdning** (attitude)
- **Bruk av hjelpemidler** (assistive technology, f.eks. skjermleser, talegjenkjenning)

Eksempler på personas med spesifikke behov:

- **Person med autismespekterforstyrrelse** (behov for forutsigbarhet og lavt stress)
- **Person med cerebral parese** (motoriske utfordringer, kanskje behov for alternativ input)
- **Blind med noe lysoppfatning** (avhengig av skjermleser og lyd)
- **Person med fibromyalgi (utmattelse)** (behov for enkle, ikke-utmattende brukerreiser)
- **Døvstum** (kan bruke teksting, tegnspråkvideo eller tekstbasert kommunikasjon)
- **Synshemmet** (behov for kontrast, forstørrelse, enkel navigasjon)
- **ARMD (aldersrelatert makuladegenerasjon)** (behov for stor tekst og høy kontrast)
- **Ikke-engelskspråklig bruker** (klare symboler, flerspråklig støtte)

Prinsipper for tilgjengelig design

1. **Mennesket først:**
   Design for variasjon og forskjeller – ta hensyn til ulike behov.
2. **Solid struktur:**
   Bygg løsningen etter etablerte standarder (f.eks. WCAG, W3C).
3. **Enkel interaksjon:**
   Alt skal fungere, være responsivt og tilgjengelig for alle inputmetoder.
4. **God veifinning:**
   Gi brukeren tydelig navigasjon og hjelp til å finne frem.
5. **Tydelig presentasjon:**
   Informasjonen skal være lett å oppfatte og tolke (farger, kontrast, layout).
6. **Klart språk:**
   Bruk enkelt og forståelig språk – unngå unødvendig sjargong.
7. **Tilgjengelig media:**
   Alle typer innhold (bilder, video, lyd) må ha alternativer for ulike sanser (f.eks. teksting, bildetekst).

De fire prinsippene i webtilgjengelighet (WCAG 2.0)

1. **Perceivable (Oppfattbar):**
   Informasjon og brukergrensesnitt må kunne oppfattes av alle sanser (syn, hørsel, etc.).
2. **Operable (Betjenbar):**
   Brukergrensesnitt og navigasjon må kunne brukes av alle, f.eks. via tastatur.
3. **Understandable (Forståelig):**
   Informasjon og betjening må være lett å forstå og forutsi.
4. **Robust (Robust):**
   Innholdet må være robust nok til å kunne tolkes pålitelig av ulike hjelpemidler og nettlesere.

---

1. Perceivable – Kjennetegn og tiltak

- **Tekstalternativer:**
  Alt ikke-tekstinnhold (bilder, video, grafikk) skal ha tekstbeskrivelser.
- **Tidsbasert media:**
  Gi alternativer (tekst, undertekster, beskrivelser) for video og lyd.
- **Tilpassbart:**
  Innhold skal kunne presenteres på ulike måter uten å miste mening (f.eks. enklere layout)

- **Tydelig og distinkt:**
  Sørg for høy kontrast, lesbar tekst, og at informasjon ikke kun formidles med farge.

---

2. Operable – Kjennetegn og tiltak

- **Tastaturtilgjengelig:**
  All funksjonalitet skal kunne brukes kun med tastatur.
- **Tilstrekkelig tid:**
  Brukeren må ha nok tid til å lese og bruke innholdet.
- **Unngå anfall:**
  Unngå innhold som kan forårsake anfall (blinkende lys, animasjoner).
- **Navigerbarhet:**
  Hjelp brukeren å finne frem, forstå hvor de er, og navigere effektivt.

---

3. Understandable – Kjennetegn og tiltak

- **Lesbarhet:**
  Teksten skal være lett å lese og forstå, på klart og enkelt språk.
- **Forutsigbarhet:**
  Oppførselen til siden skal være forutsigbar, ingen overraskelser.
- **Input-hjelp:**
  Hjelp brukeren å unngå og rette feil, f.eks. ved validering og forklaringer.

---

4. Robust – Kjennetegn og tiltak

- **Kompatibilitet:**
  Innholdet skal fungere med ulike nettlesere, hjelpemidler og fremtidige teknologier.
  - Unngå utdaterte teknologier og sørg for at skjemaelementer har riktige etiketter.

Hovedpoeng

- **WCAG** er standarden for universell utforming på nett.
- Prinsippene **POUR**: Perceivable, Operable, Understandable, Robust.
- Følg WCAG for å sikre at ALLE kan bruke og forstå innholdet ditt, uavhengig av funksjonsevne.

Assisterende teknologi for ulike funksjonsnedsettelser

Kommunikasjonsvansker

- Elektronisk talesyntese (speech synthesizer)

Hørselshemming

- Hodetelefoner/øreppropper
- Teksting i sanntid (real-time closed captioning)
- Teletypewriter (TTY/TDD)

Bevegelseshemming

- Sidevendingsapparat (page-turning device)
- Tilpassede tastatur og mus (trackball, vertikal mus, fotmus, pedal osv.)

Fysiske eller psykiske lærevansker

- Talegjenkjenning (voice recognition software)
- Opplesende lærebøker (talking textbooks)

Synshemning eller lærevansker

- Tilpasset skjerm, forstørrelsesprogram (magnification devices)
- Opplesningstjeneste, E-tekst
- Punktskriftnotatblokk (Braille note-taker)
- Punktskriftskriver (Braille printer)
- Skjermforstørrer (screen magnifier)
- Optisk skanner